Technical University of Denmark

The ePNK: A generic Petri net tool
Users' and developers' guide
for Version 1.2

Ekkart Kindler
Technical University of Denmark
DTU Compute
DK-2800 Kgs. Lyngby
Denmark
ekki@dtu.dk

DTU Compute Technical Report YYYY-XX

Revised, updated, and extended version of
IMM-Technical Report-2012-14

This is a half-way updated version of the ePNK manual for version 1.2
Draft version: June 16, 2019

Technical University of Denmark
Department of Applied Mathematics and Computer Science (DTU Compute)
Richard Petersens Plads, Building 324
DK-2800 Kgs. Lyngby, Denmark
Phone: +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

DTU Compute Technical Reports: ISSN 1601-2321

Abstract

The *ePNK* is an Eclipse based framework and platform for developing and integrating Petri net tools and applications. One of its core features is that new *Petri net types* can be plugged in, which does not require any programming. A new Petri net type can be defined by providing a model of its concepts, the so-called *Petri net type definition* (*PNTD*). In addition, the ePNK allows adding new *applications* on Petri nets to the ePNK.

The ePNK builds on the concepts and models of the *Petri Net Markup Language* (*PNML*), which is an XML based interchange format for all kinds of Petri nets, which was published as International Standard ISO/IEC 15909-2 in February 2011. Technically, ISO/IEC 15909-2 is defining an interchange format for three different kinds of high-level Petri nets and a simple version of Place/Transition systems only. But, one of the objectives of PNML was to provide a means for exchanging any kind of Petri net [10, 30, 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 [7], 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 its 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 for Petri nets.

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 create a class diagram defining the concepts of the new Petri net type, along with a mapping of these concepts to XML syntax. This type can then be plugged into the ePNK, and the graphical editor of the ePNK will be able to edit nets of this new type with

all its features. Likewise, the ePNK allows to plug in new applications for the analysis, verification or simulation of Petri nets. Moreover, it is possible to customize the graphical representation of Petri nets and their specific features, and applications can visualize their results and interact with the user on top of the graphic representation of the net in the graphical editor of the ePNK.

Actually, this was the idea when we started the development of the *Petri Net Kernel* (*PNK*) about 20 years ago [18, 12, 21]. At that time, however, we had to implement the complete IDE functionality of the PNK ourselves. The ePNK is based on Eclipse [28], so in the ePNK, we could focus on the Petri net specific parts of such a tool. We get all the functionality of a nice IDE, basically, for free. Therefore, we named the tool *ePNK* for *Eclipse-based Petri Net Kernel*. But, it is only the spirit and their 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 the ePNK is not compatible with the PNK.

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.

This manual focuses on how to use the ePNK as an end user, and on 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 add new applications to the ePNK.

A first version of this manual has been published in February 2011 as IMM-Technical Report-2011-03 already, which referred to version 0.9.1 of the ePNK. The second version of this document (IMM-Technical Reprot-2012-14) referred to version 1.0 of the ePNK, which was released in October 2012. The current version of this report refers to version 1.2 of the ePNK, which was released in August 2017. It was updated with respect to the new features of the ePNK and extended by a detailed tutorial, which discusses a complete example in all technical details.

Contents

Contents **v**

1 Introduction **1**
1.1 Motivation 1
1.2 The Petri Net Markup Language 2
1.2.1 The PNML core model 2
1.2.2 Petri net type definitions 4
1.2.3 Mapping to XML 5
1.3 ePNK: Objective 6
1.4 How to read this manual 6

2 Users' guide **9**
2.1 Eclipse as an IDE 9
2.2 Creating Petri net files 12
2.3 The tree editor 14
2.3.1 The tree editor: Overview 14
2.3.2 Creating elements 15
2.3.3 Saving the document 16
2.3.4 Validating and correcting the document 17
2.3.5 Other Petri net information 18
2.4 The graphical editor 19
2.4.1 Overview of the graphical editor 19
2.4.2 Labels 20
2.4.3 Attributes 22
2.4.4 Pages 24
2.4.5 Graphical features 25
2.5 Petri net types 27
2.5.1 PTNet 28
2.5.2 HLPNG 29

2.5.3 Other types . . . 37
2.6 Functions and Applications . . . 37
2.6.1 A simple model checker for EN-systems . . . 38
2.6.2 Applications view . . . 41
2.6.3 A simulator for high-level nets . . . 44
2.7 Limitations and pitfalls . . . 51
2.7.1 Saving files: Tree editor . . . 52
2.7.2 Reset an attribute . . . 52
2.7.3 Graphical features . . . 52
2.7.4 Petri net type: Should not be changed in an existing net . . . 54
2.7.5 Line-breaks in labels . . . 54
2.7.6 Graceful PNML interpretation . . . 54
2.7.7 Deviation from PNML . . . 55

3 Developers' guide 57
3.1 Eclipse: A development platform for the ePNK . . . 58
3.1.1 Importing ePNK projects to the workspace . . . 58
3.1.2 Installing the EMF and Ecore Tools SDK . . . 60
3.2 The PNML core model in the ePNK . . . 61
3.3 Adding functions . . . 63
3.3.1 Accessing a PNML file and its contents: A file overview 65
3.3.2 Writing PNML files: Generating multi-agent mutex . 71
3.3.3 Long-running functions: A model checker . . . 76
3.3.4 Overview of the ePNK API . . . 86
3.4 Implementing applications . . . 93
3.4.1 Overview . . . 94
3.4.2 Annotations . . . 94
3.4.3 Handlers . . . 96
3.4.4 Application . . . 101
3.5 Adding Petri net types . . . 105
3.5.1 Simple Petri net type definitions: PTNet . . . 105
3.5.2 Petri net type definitions with attributes: SE-nets . . 115
3.5.3 Petri net type definitions in general: HLPNG . . . 119
3.5.4 Petri net type definitions: Summary and overview . . 137
3.6 Defining the graphical appearance . . . 138
3.7 Adding tool specific information . . . 146
3.8 Overview of the ePNK and its projects . . . 149
3.9 Deploying extensions . . . 156

4 Tutorial: Net type and application **157**
4.1 The tool . . . 158
4.1.1 The technical net type . . . 158
4.1.2 The application . . . 161
4.2 Conceptual steps . . . 165
4.2.1 Petri net type . . . 165
4.2.2 Graphics of the Petri net type definition . . . 169
4.2.3 The simulator application . . . 171
4.3 Technical steps . . . 174
4.3.1 Installation . . . 174
4.3.2 PNTD . . . 177
4.3.3 Constraints . . . 189
4.3.4 Graphical extensions . . . 197
4.3.5 Simulator application . . . 205

5 Installation **231**
5.1 Prerequisites . . . 231
5.2 Installing the ePNK in Eclipse . . . 232

6 Experience and outlook **235**
6.1 Experiences with MBSE . . . 235
6.2 Future plans . . . 237

Bibliography **241**

Index **245**

Chapter 1

Introduction

The *ePNK* is an Eclipse based framework and platform for developing and integrating Petri net tools and applications. One of its core features is that new *Petri net types* can be plugged in, which does not require any programming. A new Petri net type can be defined by providing a model of its concepts, the so-called *Petri net type definition* (*PNTD*). In addition, the ePNK allows adding new *applications* on Petri nets to the ePNK.

The ePNK builds on the concepts and models of the *Petri Net Markup Language* (*PNML*) [8, 10, 10, 30, 1]. Therefore, we start with a brief overview of the PNML, and then discuss 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.

1.1 Motivation

The PNML is an XML-based interchange format for all kinds of Petri nets, which allows different tools to exchange Petri net models among each other. One of PNML's main features is that it is generic, which means that it provides a mechanism for defining 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 *tool specific information* to PNML documents, which is information that can be safely ignored by other tools.

Before the ePNK, there was no tool that fully supported these ideas in such a way that the tool would allow a developer to define and plug in new

Petri net types and additional tool specific extensions. And there was no generic editor supporting all Petri net types, once they are plugged in.

The lack of such a generic tool support was the starting point for developing the *ePNK*.

1.2 The Petri Net Markup Language

In order to better understand the ideas of the ePNK, we briefly discuss the main concepts and ideas of the PNML here. For more information on the PNML and on ISO/IEC 15909-2, we refer to [13, 6] or to the International Standard ISO/IEC 15909-2:2011 itself [8].

1.2.1 The PNML core model

As stated above, extensibility and genericity were two of the main objectives behind the PNML [11]. This is achieved by identifying the concepts that are common to all kinds of Petri nets in the so-called *PNML core model*. The common concepts are mainly *places*, *transitions* and *arcs*, and that these *objects* can have some kind of *label*. The PNML core model also provides means for splitting up larger Petri nets into *pages*; connections between nodes on different pages can be established by *reference places* or *reference transitions*. And PNML defines all kinds of graphical information that can be attached to the different elements, such as position, size, font-type, and font-size.

Note that in graphical editors, a label would typically be shown as an annotation attached to (or close to) the object it belongs to. Labels that should be shown as annotations are, therefore, called *Annotations*. Some labels, however, are not supposed to be shown as annotations; for example, if there are different kinds of arcs, the kind or arc might be defined as an attribute of the arc in the properties view of a tool. And in some graphical tools, the graphical representation of the arc itself might change dependent on the value of this attribute; an arc of kind "read", might be graphically represented as a line without any arrow heads or with arrow heads at both ends. An arc of kind "inhibitor" might be shown as a lollipop (see Sect. 3.6 for an example). Therefore, these kind of labels are called *attributes*.

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 1.1 shows the PNML core model of ISO/IEC 15909-2 as a UML diagram. Note that the PNML core model of the ePNK is

slightly more general than the one defined in ISO/IEC 15909-2; this way, it is possible to capture even more variants of Petri nets. For now, these differences are not so important; the most relevant differences are explained later in Sect. 3.2.

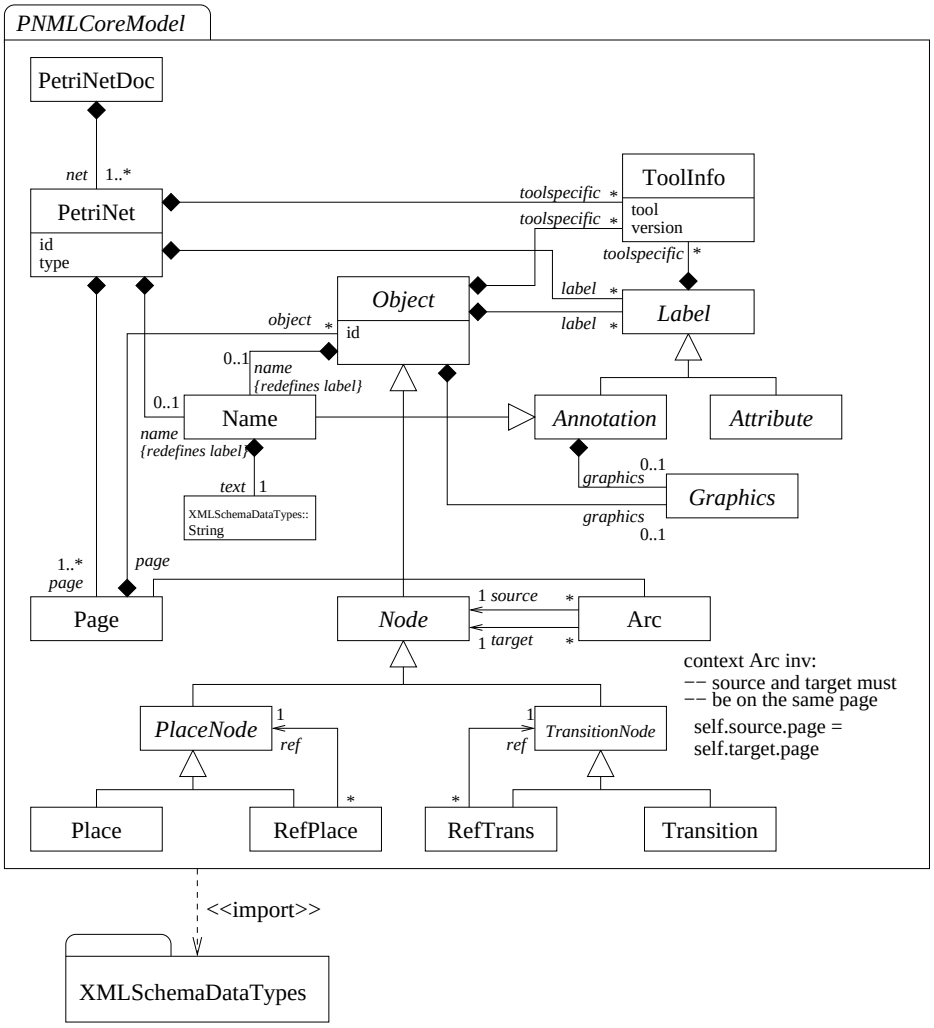

Figure 1.1: The PNML core model of ISO/IEC 15909-2:2011 [8]

Note that there is only one concrete type of label defined in the PNML core model itself, which is the *name* of an element. All the other possible labels are defined in separate Petri net type definitions, which are discussed in Sect. 1.2.2.

In addition to the concepts and relations between them, the PNML core model states also some restrictions on the structure of PNML models. For example, there is a *constraint* stating that arcs can connect only nodes that are on the same page. This constraint if formulated as an *OCL constraint* in Fig. 1.1. Note, however, that there is no constraint in the PNML core model, which states that arcs must run between a place and a transition or the other way round. The reason for not having such a constraint in the PNML core model is that there are some kinds of Petri nets that would allow arcs between places or between transitions. This is why these kind of 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.

1.2.2 Petri net type definitions

As stated above, it is the purpose of a *Petri net type definition* 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 the idea of a Petri net type definition by the help of a simple example: the definition of *Place/Transition-Systems* (*P/T-Systems* in short).

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 1.2 shows the UML model for these concepts and how they are related to the concepts of the PNML core model.

In Fig. 1.2, 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 transitions.

A Petri net type definition, would typically also define how the new concepts from Fig. 1.2 would be mapped to XML. If not stated, the ePNK will uses a default of how the new features are mapped to XML. For the example above, this default mapping is good enough (and actually compatible with ISO/IEC 15909-2).

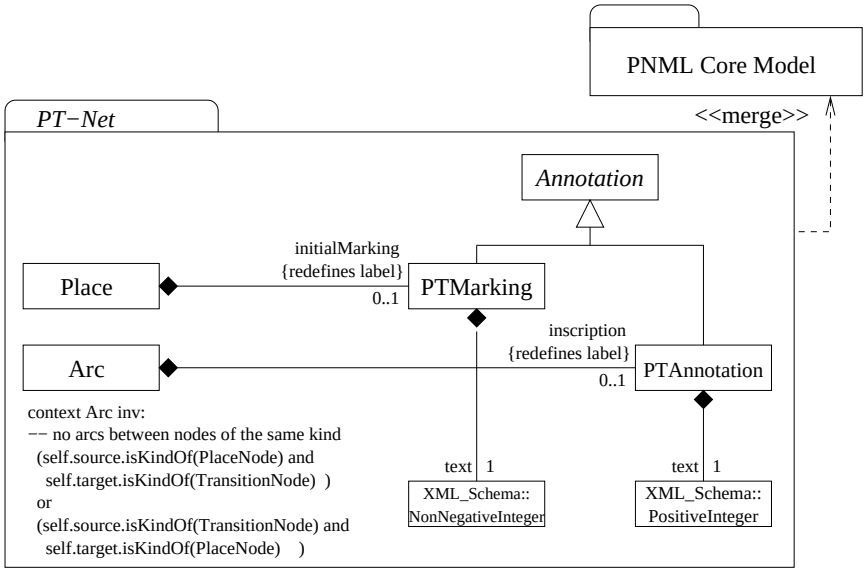

Figure 1.2: The PNTD for PT-Nets

1.2.3 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 an XML transfer format for Petri nets. So, PNML defines how these concepts are saved or represented in XML. This is achieved by mapping every concept or feature of the UML models to some XML construct.

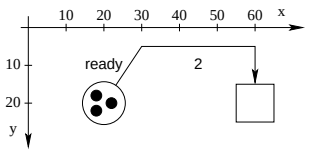

Figure 1.3: A simple P/T-System

Here, we do not give these mappings, but rather show an example (for a detailed discussion of the mappings, see [6]). Figure 1.3 shows a simple example of a P/T-System in its graphical representation (concrete syntax);

Listing 1.1 shows its representation in PNML's XML-syntax[1].

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

1.3 ePNK: Objective

The main objective of the *ePNK—DEF* was to build a tool that fully supports 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 the types defined in ISO/IEC 15909-2:2011.

As soon as such a new Petri net type definition is plugged in, it should be possible to load and save Petri net documents that contain nets of these types. Moreover, there should be a graphical editor that allows us to edit Petri nets of any plugged in Petri net type; and the editor should be fully aware of all the features (annotations and attributes) and the additional constraints of the plugged-in Petri 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 to save them. Moreover, it should be easy to plug in new functionality and *applications* for analysing Petri nets and for visualizing the results, and for manipulating Petri nets or transforming them to other models or to code.

1.4 How to read this manual

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

On the one-hand side, this manual covers 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 call these "end users" just *users*. All the information relevant for users of the ePNK can be found in Chapter 2.

On the other-hand side, this manual covers the information relevant for *developers* who are interested in using the ePNK for their purposes: extending it by defining new Petri net types and their graphical appearance, by defining new tool specific extensions, or by implementing new functionality and applications. Chapter 3 provides the information relevant for developers who want to extend the ePNK.

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

Listing 1.1: PNML code of the example net in Fig. 1.3

```
<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>
```

Chapter 4 is a tutorial, which discusses the concepts and the major steps of defining new Petri net types, their graphical appearance, and a simple simulator application on top of it. The tutorial goes into all the technical details, and is written in such a way that it can be read independently from Chapter 3.

Chapter 5 discusses the installation of Eclipse and the ePNK as well as the major changes of version 1.2 of the ePNK with respect to version 1.0.

Chapter 2

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 and applications. 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. For the particular syntax of some labels of a specific Petri net type, it might be necessary to refer to the documentation of the specific Petri net type. 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.

2.1 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 with Eclipse already can directly read on in Sect. 2.2.

Once you installed and started Eclipse (see Chapter 5), you 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 2.1 shows an example of the Eclipse workbench, with some numbers marking some parts, which we discuss next.

At the top of Fig. 2.1 marked by (1), you can see the *menu bar* and the *toolbar*. 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 your 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, and CNTRL-X does not work properly. In order not to mess up models, by improperly using cut/paste operations, the graphical editor of the ePNK does not support copy-/paste yet.

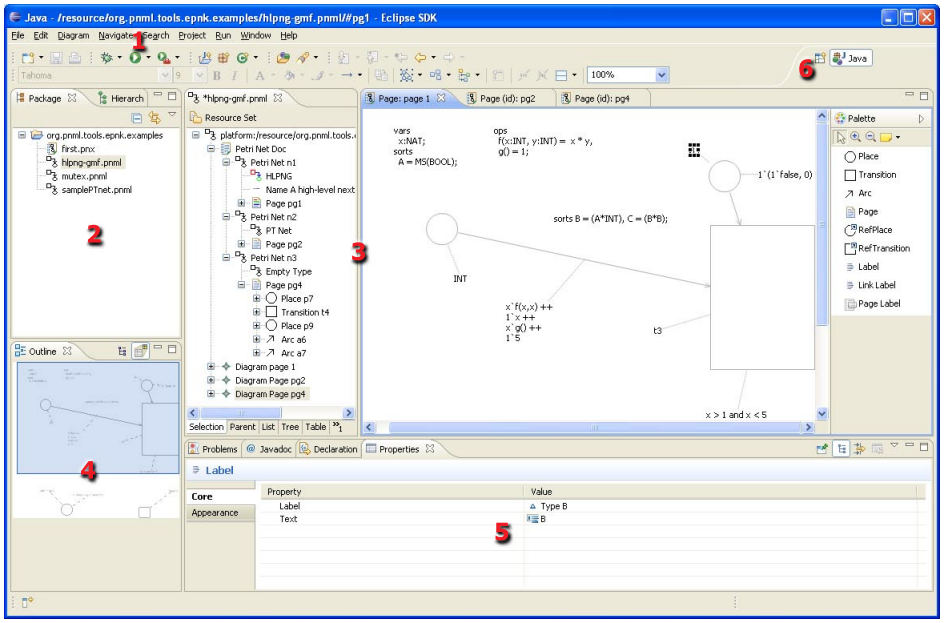

Figure 2.1: The Eclipse workbench

On the left-hands side, marked by (2), you can see the *package explorer*, which gives you access to all the files in your workbench. The package explorer can be used for browsing through the existing files and for manipulating, renaming, copying, moving, and deleting 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 explorers 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 relevant only 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 the explorer and then selecting "New"→"Project". Note that in the dialog, you can create many different kinds of projects; for us the kind "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 that are started in Eclipse will be opened. Note that there can be many editors open at the same time (in our example, there are four editors open). Typically, you can see only 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 in it determines what you see in some other views. For example, you can see the *outline view* (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, typically, double click on the resource you want to open. This will open the *default editor* on the selected resource. You can also 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 editing the contents of a resource depends on the kind of editor; generally, it is straightforward. 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 of the ePNK cannot be initiated directly from the explorer since the resource could have many pages and a page is not the top-level element. When you open a PNML file, a tree editor will be opened that shows the structure of the Petri net. The graphical editors for a page of a Petri net can be opened by a pop-up menu (right mouse button) on pages in the tree editor for Petri nets or by a double click on the page (see Sect. 2.4 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. 2.1). The outline gives an overview of the contents of the currently active editor and, in case of a graphical editor, allows us to quickly move around 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 of the workbench, which are 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.

2.2 Creating Petri net files

This section explains how to create new ePNK files. Note that there are two formats in which the ePNK can save a Petri net. The first and recommended

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

format is PNML. The second format 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 EMF technology used for implementing the ePNK and, therefore, 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 is on PNML.

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.compute.dtu.dk/~ekki/projects/ePNK/`. You can also use a text editor and 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 is started via the "New" menu, which can be either accessed by the "File" menu from the menu bar or via the pop-up menu that opens on a click to the a right mouse button 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 choose "PNML Document" from the "ePNK" category and press "Next". In the next dialog, you must choose a name and, if you want, you can choose a different folder in which this file should be created. Pressing "Finish" will create the file; then, the newly created file will be opened in a tree editor (see Sect. 2.3); 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 choose 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 "PNX Document". When you use this wizard, a PNX file will be created. 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 exactly what you are doing).

2.3 The tree editor

As mentioned earlier, 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 the *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 why there are two different editors in the ePNK. The graphical editor for pages is always started from a selected page – either in the tree editor or in the graphical editor. When opening a PNML document from the resource explorer, it will always be the tree editor that opens. A graphical editor can be opened by a double click on a page element in an already open editor.

2.3.1 The tree editor: Overview

Let us have a closer look at the tree editors first. Figure 2.2 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. 2.2. Therefore, it contains only the Petri net document element without any contents. The other PNML document, which is open on the left-hand side ("hlpng-gmf.pnml"), shows a Petri net document with three nets that have different types.

These documents were opened from the explorer by a double click[2] on the respective file in the workbench's 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 net 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, 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 arcs, which are

[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 is opened with the editor you had selected the last time.

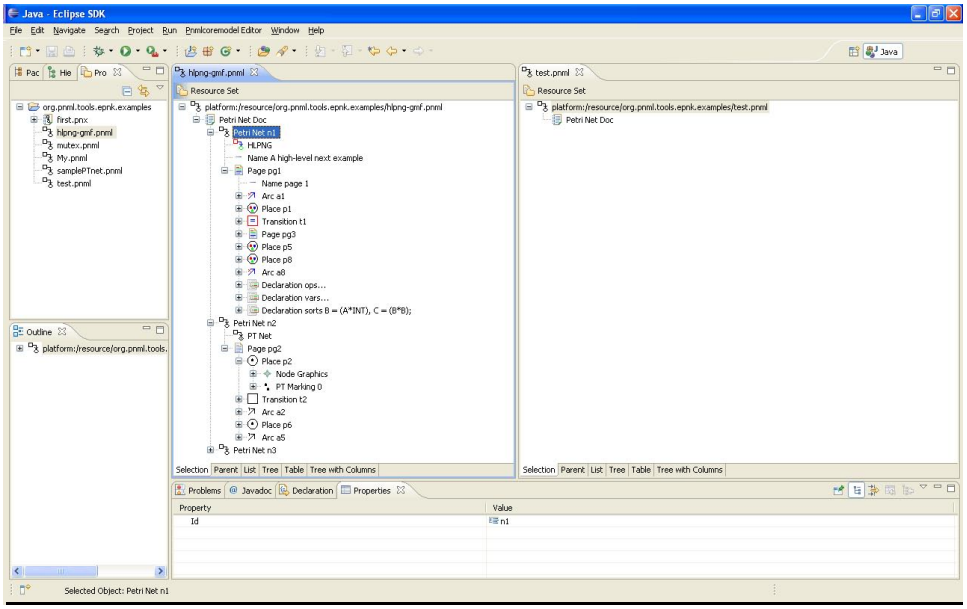

Figure 2.2: Two Petri net documents in tree editors

indicated by corresponding icons. You can also see some pages and subpages. Note that the icons for the places, transitions, and arcs are different for the two different Petri net types, so that it easier to distinguish them on a first glance. In the properties view at the bottom, you can see the properties of the currently selected element, which is Petri net n1; the only property is its id. Note that this net 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".

2.3.2 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 (along with 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*. Inserting a child element is done by right-clicking on the element to which you want to add a child, then selecting "New Child" in the dialog that pops up, and then selecting the appropriate element. Figure 2.3 shows the pop-up dialog when inserting a new Petri net to the Petri net document.

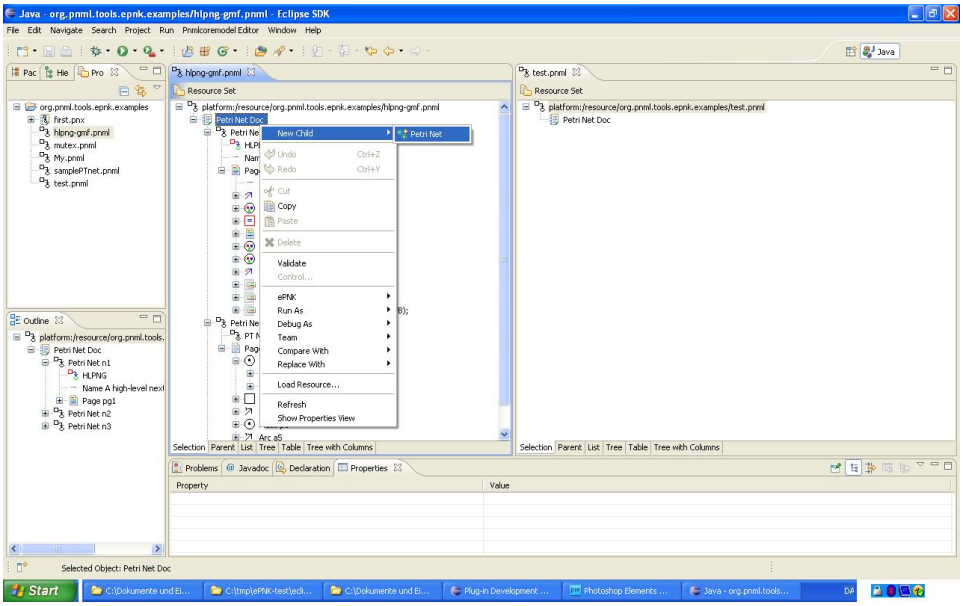

Figure 2.3: Pop-up menu when inserting a new Petri net

Note that this dialog will show all available Petri net types, from which you will need to select. Then a new net of that type will be created. Note that the created net will contain a child element, which represents the type of this net. You should never delete or change this net type manually[3]. You will find some more information on the Petri net types that are deployed together with the ePNK in Sect. 2.5.

After you have created a new Petri net, the name of the net and new pages can be inserted to it by via the "New Child" pop-up menu on the selected Petri net similar to creating the net – just select the kind of element you want to create.

2.3.3 Saving the document

As discussed in Sect. 2.1, you can save the net via the "File" menu or with the CNTRL-S shortcut. Note that saving the net must be done – and can be done only – in the tree editor. Therefore, only the tree editor shows the dirty-flag, when the editor contains unsaved changes.

[3]Unless you know exactly what you are doing.

2.3.4 Validating and correcting the document

Before saving a PNML document, it is a good idea to *validate* the net. This will check whether all the constraints that PNML and the respective Petri net types imposes on the Petri net document 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, and other tools might not be able to load it.

There are many things that can be wrong and need validation on a Petri net document. Most of them are type specific (such as the requirement 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. 2.5. 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 on the right mouse button on the Petri net document; then, in the pop-up menu, select "Validate". The result of the validation will be shown in a dialog; the results of the validation is also visible in the *problems view*[4] after the validation as shown in Fig. 2.4. 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), which means that the same ID is used twice in this document.

Note that you can double-click on the individual problems in the problems view. Then, an editor is opened with the element to which this problem refers to selected. If there is a graphical editor open for the page that contains the element with the error, the ePNK will show the selected element in the graphical editor; if there is not graphical editor open with that element, the element will be shown in the tree editor.

[4] If the problems view is not open, you can open it by "Window"→"Show View" →"Problems".

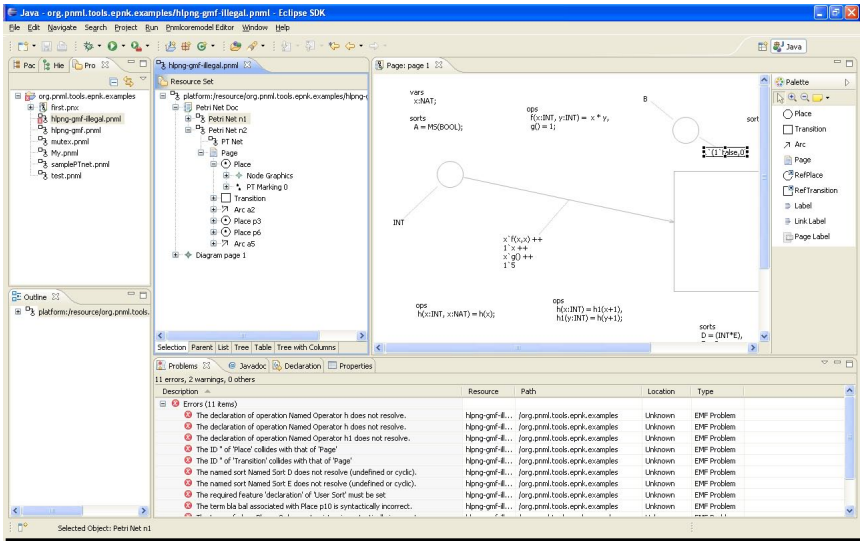

Figure 2.4: The problems view with many constraint violations

In order to reduce the number of errors, you can also do a validation on 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 can be fixed automatically. For example, ids can 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. Actually, there is a shortcut: a double-click on the Petri net document element in the tree editor will automatically add all the missing ids.

Some other errors might need some manual changes, which could typically be made in the graphical editor of pages.

2.3.5 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. 2.3, 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[5].

2.4 The graphical editor

For editing the contents of pages, the graphical editor should be used. The graphical editor can be opened by right-clicking on the respective page in the tre -editor and then, in the pop-up menu, selecting "ePNK"→"Start GMF Editor on Page". A shortcut for this is double-clicking on the page.

When you open a graphical editor on a page for the first time, you will be warned that this change cannot be undone – and no undos will be possible beyond that point. Therefore, you will be asked whether you want to proceed with that operation or not.

Figure 2.4 shows the graphical editor with a page open in a graphical editor; this is a 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 and the graphical editor are visible at the same time.

Figure 2.7 shows another example of a Petri net opened in the graphical editor, which we will discuss in Sect. 2.4.3.

2.4.1 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 shows the page's name (if the page has a name label assigned to it) 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.

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

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.

2.4.2 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 type of net Petri net via a pop-up menu. Figure 2.5 shows the pop-up menu during the creation of a page label for a high-level net, 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.

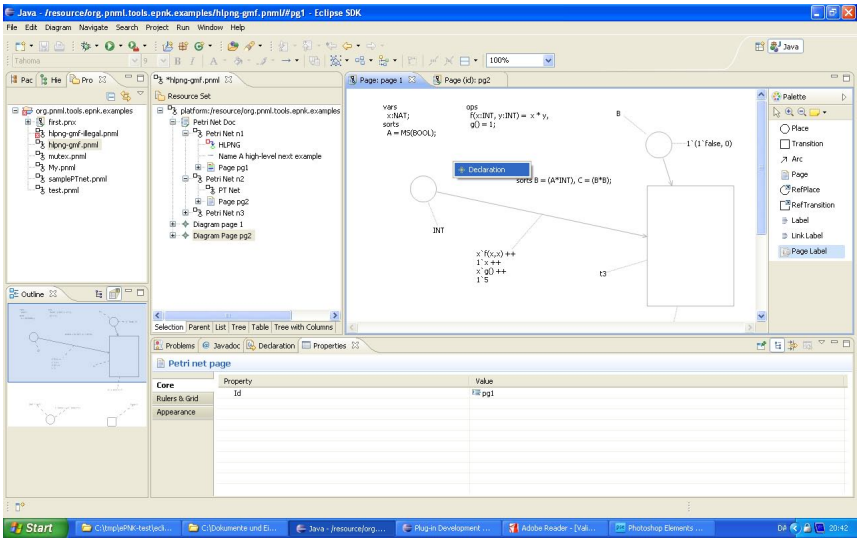

Figure 2.5: Pop-up menu during page label creation

The process for creating a label and attaching it to an object is slightly different. First you must create the label. This new label, however, will not be attached to any object yet, which is indicated by the text "<not connected

label>". A not yet connected label can – and must – be connected to some Petri net object (which could also be a sub-page) by choosing the tool "Link Label", clicking on a label, and then without releasing the mouse button moving it over the object the label should be attached to. Then, a pop-up menu will be opened showing you the possible kinds of labels that could still be attached to the chosen object. This is shown in Figure 2.6 for a label that is attached to a place of a P/T-System. The possible 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.

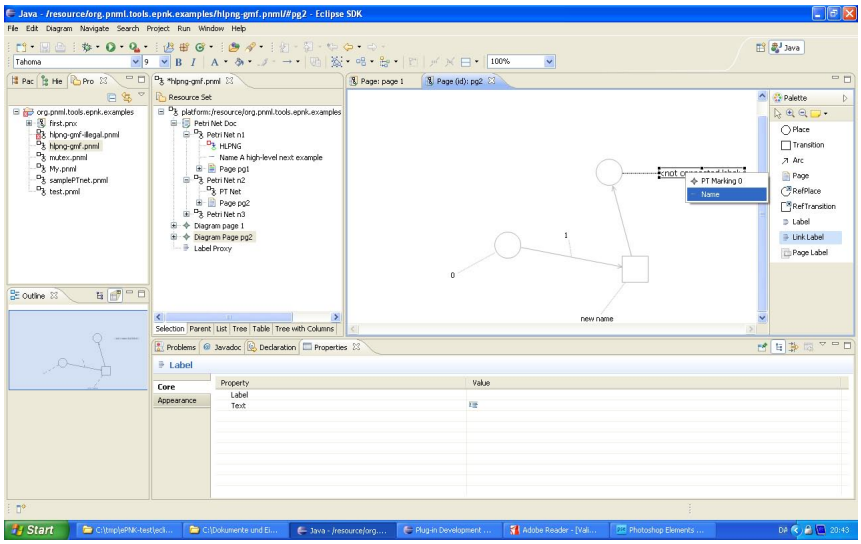

Figure 2.6: Pop-up menu during attaching a unconnected label to an object

After the label has been 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 Petri net type and which kind of label it is. In general, when editing labels and page labels there are two different cases: The first case are *simple labels*, which typically are simple values like "true" or "false", values like numbers, arbitrary strings, or IDs (in general, it will be some form of data types). If such a label is typed in syntactically incorrect, 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

structured 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 for the names are structured). All these labels will be parsed and checked for syntactical correctness; but the entered text will be stored in all cases. If the text is syntactically incorrect, however, the structure is not set and this will very likely result in some validation error later (see Sect. 2.3.4). So this error needs to be fixed, by editing the label again. In case of such an error, the label will be marked with a warning symbol (and when the mouse is moved over the warning symbol, the tool tip will indicate that the label could not be parsed). If, for example, we delete the comma that separates the two sort declarations in the label "sorts B = (A*INT), C = (B*B);" the label will be decorated with a warning symbol. Upon validation, a validation error message will be given and later shown in the problems view.

The documentation of the legal syntax of these 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. 2.5.

Note that labels, in principle[6], 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.

Some Petri net types have quite many labels, and it is quite tedious to create and link all these labels to a Petri net object. Therefore, the graphical editor of the ePNK has a context menu when one of its objects is selected, which will create and attach all missing default labels of that object (and arrange them equally distributed around the object). The menu pops up, when the right mouse button is pressed on the element; then select "ePNK"→"Add default labels".

2.4.3 Attributes

As discussed earlier, some "labels" of a Petri net object are not supposed to be represented as graphical annotations of a Petri net object. These are called *attributes*. Figure 2.7 shows an example of a Petri net which uses attributes for some objects. It is a signal-event net (SE-net) [27], which will be used later in the Developers' Guide of this manual as an example of how to define new Petri net types for the ePNK[7].

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

[7] The reason we need to resort to SE-nets here is that all the Petri net types that are defined in ISO/IEC 15909-2 use annotations only

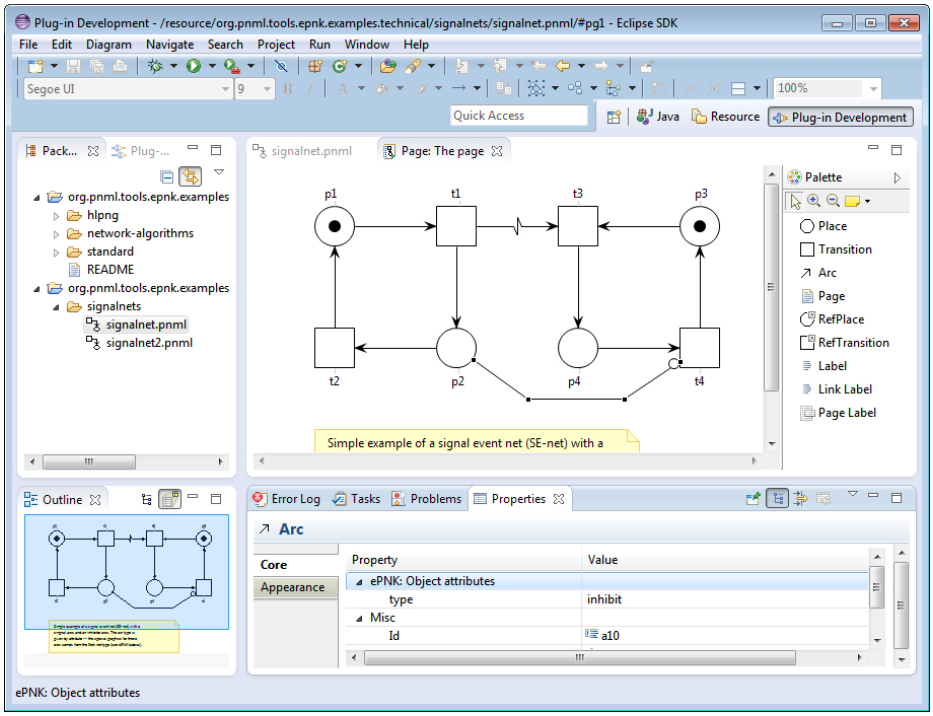

Figure 2.7: A Signal/Event net open in the graphical editor

In this example, arcs have an arc type. If the arc type is not set, it is considered to be a normal arc. The arc that is selected in Fig. 2.7, is of type "inhibit", i. e. it represents an inhibitor arc. The value of the attribute is visible in the properties view, and can also be changed there. In this example, the arc is graphically shown as a lollipop – which comes from the implementation of that Petri net type, which implements a dedicated graphical representation for these kind of arcs. Another example is the signal arc (the one with the "flash" decoration), which is an arc of type "signal". For this net type, we have chosen the marking to be an attribute; therefore, the marking is not shown as a label. It can only be changed by selecting the respective place, and then changing the respective attribute in the properties view. In this net type, the marking is indicated with a special graphics for the place (as black tokens).

As mentioned before, the value of an attribute can be changed by selecting the respective object and then changing the value in the properties view. Depending on the type of the attribute, this can be done by either typing some text into the value column for that attribute or by a drop down menu

(if there are only finitely many possible values). If you want to reset a value to the default value (the default is that the value is not set at all), you can right-click into the property column of that property in the properties view and then select "Restore default value".

2.4.4 Pages

The graphical editor also allows you to create other pages on the page it is showing. In order to avoid confusion, we call such a page a *sub-page*. Creating a sub-page can be done with the "Page" tool, in the very same way in which places or transitions are created on a page. In the graphical editor, a sub-page is graphically represented as a rounded rectangle.

It is possible to open a graphical editor on a sub-page from the graphical editor 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). And also here, a double-click on the page in the graphical editor is a shortcut for opening a graphical editor on this page. Therefore, the tree editor is needed only for creating the top-level pages of the net; all the sub-pages could be created by the graphical editor. But navigation to sub pages might be a bit easier and much faster 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.

The graphical editor indicates by a special decoration when a sub-page is open in some graphical editor: a symbol of an open folder.

What is more important about pages is how to deal with their labels. Typically, all the type-specific labels are represented as *page labels* on the respective sub-page. For HLPNGs, for example, the declarations of a page are shown as page labels on that sub-page. The name, however, will be shown as a label attached to that page on the super-page. Which labels are shown as labels attached to the sub-page on the super page, and which labels are shown as page label on the opened sub-page is up to the Petri net type definition.

Note that some Petri net types, allow labels to be directly defined for the net, in which case they are *net labels*and not *page labels*. This applies for example for all kinds of declarations in High-level Petri nets (HLPNGs).

Though, we do not recommend to use them, ISO/IEC 15909-2 mandates tools to support them. The ePNL allows to add net labels in the tree editor, if needed.

Note that even though, a Petri net can consist of many pages, the net is considered as a single flat net only. *Reference places* and *reference transitions*, are conceptually merged with the places and transitions they refer to. This is what we call *flattening of the net*. In Sect. 3.3.4.4 of the Developers' Guide, we will see that the ePNK provides mechanisms for accessing the flattened net structure in a uniform way.

2.4.5 Graphical features

The graphical editor of the ePNK allows you to make all kinds of changes to the graphical appearance of the Petri net. The features supported are the standard features of GMF editors. Figure 2.8 shows the same net as above again; in order to high-light some GMF features now, the grid and rulers switched on.

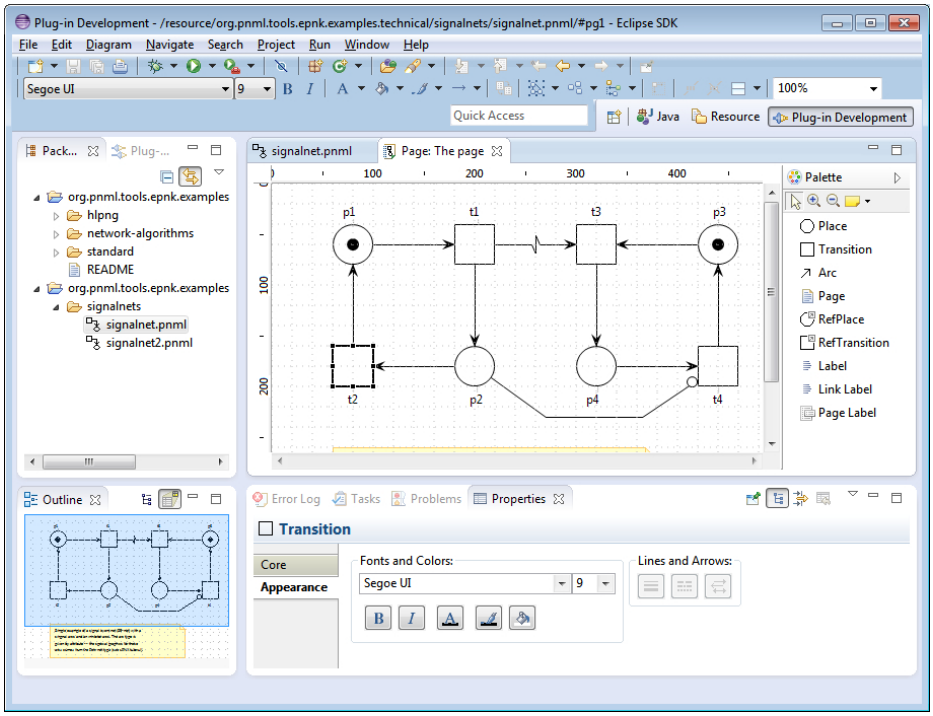

Figure 2.8: The Signal/Event net again: now with grid and rulers

Since the graphical features are pretty straightforward, and are similar to typical graphical editors, we do not explain the details here. The graphical features can be changed in the properties view, when selecting the "Appearance" section (up to now, the properties view had always shown the "Core" section). Figure 2.8 shows the graphical features that can be changed, when a node is selected. The available features vary for the different kinds of objects.

Note that Petri net types can define a special graphical appearance for some nodes or arcs, which depends on attributes or some other information of the resp. element. In that case, some graphical information selected by the user might be overruled by the specific graphical information for that element of the Petri net type. You can see an example of such a specific graphics in Fig. 2.8, where signal arcs and inhibitor arcs are shown in their usual graphical notation. In this example, however, the specific graphical information does not interfere with the user graphics. But, if a net type defines that some specific arcs should be displayed in red colour, for example, this would override the colour chosen by the user.

The ePNK saves the nets in exactly the graphical representation you see them before you save the file. But, the exact graphical representation is saved as a tool specific extension of the ePNK! This means that the exact graphical representation can be reproduced in the ePNK only.

The ePNK also transfers most of the graphical information to the respective features of PNML, so that other tools could reproduce almost the same graphical appearance as you see it in the ePNK. Some features, however, are not supported by PNML (as for example the size of labels) and some others are not yet transferred by the ePNK to PNML. In turn, some graphical features of PNML are not supported by the ePNK editor (e. g. the alignment of text in labels).

In Sect. 2.7.3, you will find a complete list of graphical information that is transferred to PNML elements. Here is a brief list of graphical information from the graphical ePNK editor that is *not* transferred to PNML: font styles (bold or italic for labels); routing and jump link information for arcs. Note that also the comments that you can place on a page will be lost in other tools, since this is tool specific information only in the ePNK (comments are not a concept of PNML).

Note that PNML nodes that do not have a size attached to them, will have the width and height 40pt in the ePNK (which the GMF default).

Arcs can have *intermediate points* or *bend points* in the ePNK as well as in the PNML. These intermediate points will be transferred to the PNML model. But there is some caveat: when a node is moved in the ePNK

editor, the bend points of the attached arcs might also be moved. Due to some quirk in GMF, these changes are not transferred to the PNML model at all. If you want to be sure that the intermediate points of the PNML model corresponds to what you see in the graphical editor of the ePNK, you should move at least one bend point of each arc attached to the node after you have moved the node.

Arcs are either drawn as a *polyline* or as a *bezier curves*, which is defined by the "Smoothness" chosen for the respective arc in the ePNK editor. The ePNK draws an arc as a polyline, if the "Smoothness" is set to "None"; for all other choices of "Smoothness" (i. e. "Normal", "Less", or "More") the arc will be drawn as a quadratic bezier curve, where every second intermediate point is used as a control point as mandated by ISO/IEC 15909-2. The information on whether an arc should be drawn as a straight line or as a bezier curve is transferred to the PNML model.

The graphical editor of the ePNK also supports the *image* feature of PNML: Every node can be assigned an image in the PNML model; instead of the normal shape of the respective node, the image will be shown. The ePNK supports JPEG and PNG – as required by PNML. Actually, the ePNK supports even more image formats[8]. But, since the other formats are not supported by PNML, we recommend to use the JPEG and PNG format only.

The properties view has an image property, in which the path to the image can be set directly when the resp. element is selected in the graphical editor. The paths should be a relative path to the image, staring from the folder that contains the PNML document.

Note that, for efficiency reasons, each image is loaded only once – the first time it needs to be shown in the graphical editor. If you change an image file and you want a graphical editor in an already open document to show the new image, the *image cache* of the open editor must be cleared explicitly. To this end, right-click on the top-level "Petri Net Doc" element in the ePNK tree editor of that document and select "ePNK→"Clear Image Cache".

2.5 Petri net types

In this section, we give an overview of the Petri net types that are deployed together with the ePNK. In the basic version of the ePNK, these are *P/T*-

[8] All image formats supported by the Eclipse SWT `ImageLoader` should work: BMP, ICO, JPEG, GIF, PNG and TIFF.

Systems (*PTNet*) and *high-level Petri nets* (*HLPNG*). Moreover, there is the *empty type* (*Empty*), which, however, does not contain any concepts in addition to the PNML core model; therefore, we do not discuss the empty type 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 are a way of representing P/T-Nets as high-level nets; basically, "dot nets" are high-level nets restricted to the sort "DOT" and a minimal version of operators on them; "*symmetric nets*" are a restricted version of high-level nets that uses some special finite sorts and a limited set of operations only; 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 by selecting the kind attribute in the properties view (identified by the respective URI as defined by ISO/IEC 15909-2). For a detailed discussion of the legal constructs of the different kinds of HLPNG, we refer to the discussion of ISO/IEC 15909-2 [6]. Note that, in contrast to the Petri net type, the kind of a HLPNG can be manually changed anytime, since the kind of HLPNG concerns the validation only. The PNML syntax is the same.

2.5.1 PTNet

We start with explaining the details of *PTNets*. In Sect. 1.2.2 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 a label whether the label is syntactically correct (see Sect. 2.4.2); if it is not correct, 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[9]. The arc-inscription is similar, just that it must represent a positive integer (i. e. must be a number 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[10]. 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.

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

[10] In the tree editor, illegal arcs can be created, but the respective net would not pass validation.

when flattening a net (see Sect. 2.4.4), this amounts to the above condition.

2.5.2 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 [8, 6]. Actually, HLPNG are conceptually quite close to coloured Petri nets [9] or algebraic system nets [19].

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

Declaration A *declaration*is a page label, which is 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 they are attached to a specific page.

Declarations do not need to be contained in a page at all – they can be contained directly in the net. We do not recommend to make use of that; but ISO/IEC 15909-2 mandates this to be possible. So, the ePNK can read nets with declarations which are directly contained in the net and such net labels can be created in the tree editor of the ePNK, if needed.

Type A *type* is a label that is associated with a place. Every place must have exactly one type label which denotes the sort of the tokens on that place. This sort can be built from the predefined sorts of HLPNGs or some user-defined sorts.

HLMarking A *marking* is a label that is attached to a place and defines the place's initial marking. The marking is represented by a ground-term[11], which must denote a multiset over the place's type. Note that this label may be omitted, in which case the initial marking is considered to be empty. 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; if the condition is missing, it is assumed to be true.

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

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[12].

All labels of HLPNGs are structural labels (see Sect. 2.4.2), which means that the user can edit them and leave them syntactically incorrect. Of course, this will not pass validation; but, it is possible to save nets with incorrect labels and load them again, so that the labels can be corrected another time.

PNML does not define or mandate a concrete syntax for declarations and terms. The concrete syntax for the labels is up to each tool; so it might be different in different tools. What matters is the abstract syntax. In order to, get the abstract syntax of a HLPNG net of some net from another tool into the concrete representation of the ePNK, the ePNK provides a pop-up action, which converts the abstract syntax into ePNK's concrete syntax. It is available in the tree editor[13], when a HLPN object is selected "ePNK"*rightarrow*"Serialise HLPNG Labels".

ePNK's concrete syntax for HLPNG labels resembles the one of CPN Tools [9], but it is not identical to the one of CPN Tools! Below, we explain this ePNK's concrete syntax for labels. 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;
```

[12] Actually, 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.

[13] Note that you must have installed the "HLPNG Label Serialisation" feature for this to work.

```
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. The operation f takes two parameters of type INT; the operation g does not have parameters. Note that named operations, basically, are abbreviations and, therefore, do not allow any recursion (see [6] for details). 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 sides 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 used as 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 2.1 shows the part of the grammar for declarations. Listing 2.2 shows the part of the grammar for terms. Note that we have simplified the grammar for making it more readable. The simplification, however, makes the grammar ambiguous (i. e. some texts could be parsed in two or more different ways). The ambiguities can be resolved again by assigning a binding priority to the different operators – moreover all operators are left-associative. Each 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

Listing 2.1: Grammar for declarations

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

SortDecl :
    NamedSort | Partition;

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 2.2: 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;
```

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 2.3 and 2.4 show the part of the grammar for built-in sorts and constants. Note that every number constant will implicitly be assigned the

Listing 2.3: 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)?;
```

tightest fitting sort: INT, NAT, or POS. If a positive integer, say 5 should have the type INT instead, this can be expressed by 5:INT, which works like a type cast in object-oriented programming languages.

In addition to these syntactical constraints, the terms must also be cor-

Listing 2.4: 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 ;
```

rectly 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 (a correct sort in the context of the defined sorts of the net).
2. Every declaration must be syntactically correct and correctly typed.
3. Every declaration must properly resolve (must not be recursive and all symbols it refers to must be defined).
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 attached place's sort.
7. Every condition must be a term of sort BOOL.
8. Every declaration should 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. The parameters of every 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).

As mentioned earlier, PNML and ISO/IEC 15909-2 do not define a concrete (textual) syntax for declarations and terms. The syntax defined here is a syntax specific to the ePNK. In principle, a PNML document with a high-level Petri net in it could leave all the textual parts of the labels empty. In that case, the most important structure and content of these labels would not be visible in the graphical editor at all. The user would not see and would not be able to edit the labels textually. In order to convert this structural information into some text that can be edited by the user in the ePNK, a simple extension to the ePNK is deployed as a separate feature called "HLPNG Label Serialisation". If you have the "HLPNG Label

Serialisation" feature installed, you can serialize all structured labels to the textual syntax of the ePNK. To this end, right-click on the respective HLPN element (the Petri net) in the tree editor; then select "ePNK"→"Serialise HLPNG Labels".Then, you will be able to see and edit the labels in the syntax that we have discussed above (independently from which editor the PNML file came from).

2.5.3 Other types

Note that there are some other Petri net types coming with the ePNK, if if you have the ePNK tutorial installed. Most notably, there are *Signal/Event-systems* (*SE-Nets*) , which we will use as an example later in the Developers' Guide (see Chapt. 3). Moreover, there is a technical Petri net type where pages and arcs are equipped with comments, and arcs can have different types. This type is called *ArcTypes*, but mainly serves as a tutorial for using attributes, and for equipping a net with graphical extensions.

If you have the ECNO extensions installed, you have so-called ECNO nets, which allow to model the life-cycle of elements in the so-called *Event Coordination Notation*. These are a story of their own [17] and are not discussed here.

2.6 Functions and Applications

The ePNK in its basic version does not come with much functionality for analysing, simulating and verifying Petri nets. Its main purpose, is to provide a graphical editor for Petri nets and PNML Documents, and to provide an infrastructure so that new Petri net types and new functions and applications for Petri nets can be plugged in. By and by, some functions have been developed that are now deployed together with the ePNK. And there are some applications of the ePNK, which are projects in their own right – for example the ECNO project, which generates code for so-called ECNO nets [16]. We hope, that over time, more functions and applications of other ePNK developers will be deployed together with the ePNK.

In this section, we explain some of the basic functions and applications that are deployed together with the ePNK. In these examples, we will also explain the *ePNK applications view*, which is used as a general user interface for the end user to control ePNK applications. Later, in the Developers' Guide in Chapt. 3, we explain how you can contribute your own functions and applications to the ePNK.

Generally, the ePNK distinguishes *functions* and *applications*. A *function* is something, which is initiated on some Petri net, possibly asking the user for some extra input, then does some computation, and when the computation is finished, provides some output to the user in form of some dialog. After that, the function and its result are gone. One example of such a function is the verification of some CTL formula for some Petri net by a model checker, which is discussed in Sect. 2.6.1. In particular, the model checker does not show or visualize any result in the Petri net itself. Also an *application* is initiated on some net. In contrast to a function, the application has a longer live-time, it shows some feedback to the user on top of the Petri net in the graphical editor, and also provides means for the user to interact with the application. An example of such an application is a simulator for high-level Petri nets, which is discussed in Sect. 2.6.3. This simulator shows graphically which transitions are currently enabled, and, by clicking on them, the user can determine which transition should fire next.

2.6.1 A simple model checker for EN-systems

In this section, we briefly discuss how to use the *model checker*, which can be initiated on P/T-nets. Note that even though this model checker is initiated on P/T-nets, the model checker interprets the net as an Elementary Net System (EN-systems) [29, 26], which means that at any time on any place, there can be at most one token. A transition that would add another token to a place, would not be able to fire – and all arc-inscriptions are ignored.

Figure 2.9 shows a P/T-net which consists of several pages. Page pg0 (not visible) contains a single place semaphor with one initial token, and pages pg1, pg2, and pg3 model three agents with the same life-cycle, which is shown in the graphical editor in Fig. 2.9. The three agents are competing for the semaphor in order to access their critical section criticalx. Actually, this net was automatically generated by a wizard for creating a net with any number of agents. This wizard can be initiated by "File"→"New"→"Other..." and then selecting "Multi-agent Mutex Net Wizard" from the category "ePNK".

The model checker on this net can be initiated by right-clicking on the Petri net element in the tree editor and then selecting "ePNK"→"Model checker". Then, a dialog like the one in Fig. 2.10 pops up, where two CTL formula, which make sense in any system, are provided as a default input to the model checker:

```
AG EX true, EG EX true # deadlock free, infinite path
```

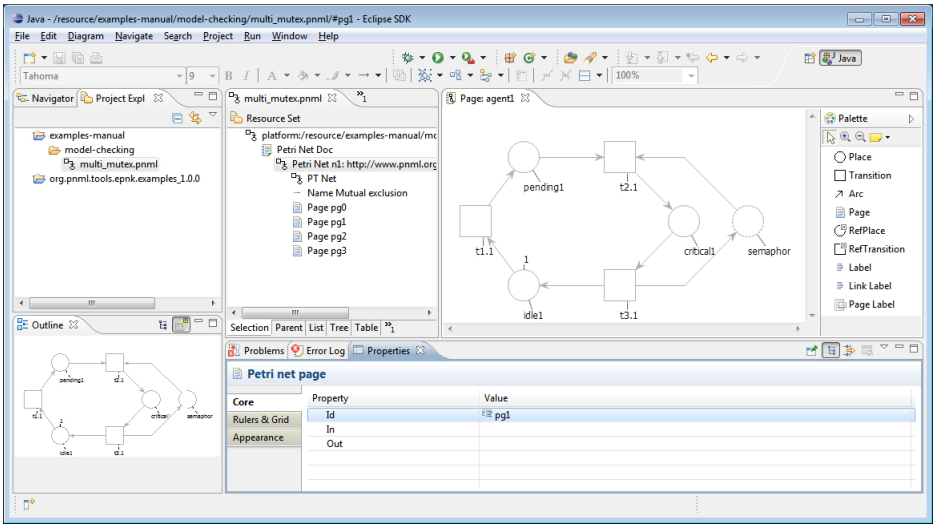

Figure 2.9: Mutex example with multiple agents

As indicated by the comments behind the hash symbols, these two formula will check whether the system is deadlock free and whether there is at least one infinite path. You can of course enter some other formulas, which can

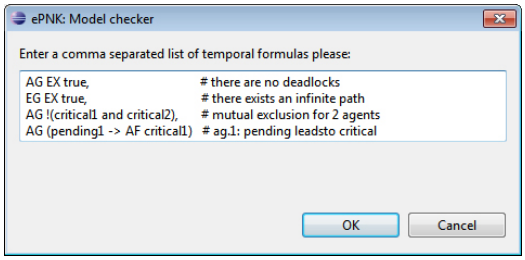

Figure 2.10: Model checking dialog: Input of formulas

use place names in order to formulate some more specific properties like:

```
AG !(critical1 and critical2) # mutual exclusion for 2 agents
AG (pending1 -> AF critical1) # ag.1: pending leadsto critical
```

For the exact syntax of the temporal formulas (CTL formulas), we refer to the documentation of the MCiE library `http://www2.cs.uni-paderborn.de/cs/kindler/Lehre/MCiE/` and its example formulas, and we recommend to have a look into the documentation of MCiE's parser package. Place names will be used as variables in the formula. You need to make

sure that place names of the Petri net are legal MCiE variable names (in particular, there should not be white spaces or special characters in them). One speciality of the syntax of CTL formulas of MCiE is that the binary temporal operators, such as EU and AR, are represented in infix notation like $p1EUp2$ instead of the more common notation $E[p1Up2]$. Moreover, you can use the hash symbol `#` as a line comment – everything following the hash symbol in the same line is ignored by the MCiE parser.

If the formulas entered to the dialog in Fig. 2.10 are syntactically incorrect, the dialog will pop up again, indicating the position of the syntax error. You can either correct the error or abort the dialog.

If the sequence of formulas is syntactically correct and the dialog was not aborted, the model checker will be started on the net and check all the formulas. Since model checking can take quite some time for larger nets, the actual model checking is done in the background, so that the Eclipse GUI is not blocked while the model checking is done. This is actually an Eclipse concept, which is called *jobs*. If a job should take too long, it can be aborted in the Eclipse *progress view*, which can be easily opened while jobs are running in the background by clicking on the *progress indicator* in the bottom line to the right of the Eclipse workbench[14].

When the model checking job is finished, this will be indicated by a symbol in the bottom right corner of the Eclipse workbench. When you click on it, a dialog with the model checking result will pop up. For the net and the CTL formulas from the input dialog above, the result dialog is shown in Fig. 2.11, indicating that the first three formulas evaluate to true, the third evaluates to false[15].

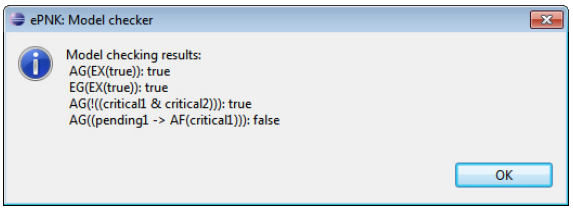

Figure 2.11: Model checking dialog: Result

[14] If the Eclipse progress area and icon are too small for you, you can open the Eclipse progress view explicitly: "Window"→ "Show View"→ "Other..." and then select "Progress" in category "General".

[15] The reasons for this property not being true is that there are not fairness assumptions in this net.

2.6.2 Applications view

As mentioned above, ePNK applications are a bit more involved, since the user can interact with them, and applications can show some visual feedback to the user and interact with the user with visual feedback on top of the Petri net shown in the graphical ePNK editor. One example of such an application is an interactive simulator for high-level nets, which is discussed in Sect. 2.6.3.

In order to explain the *applications view*of the ePNK that is used for controlling the running applications and for choosing which application the user wants to interact with, we discuss a simple application: a simple *simulator* for Place/Transition-nets.

Figure 2.12 shows the ePNK with a P/T-net, two of the net's pages are open in the graphical editor. At the bottom, the *applications view* of the ePNK is shown. Note that, initially, the application view is not open in Eclipse. You can open it in the following way: Choose "Window"→"Show View"→"Other..."; then, in the opened "Show View" dialog select "ePNK: Applications" from the "ePNK" category.

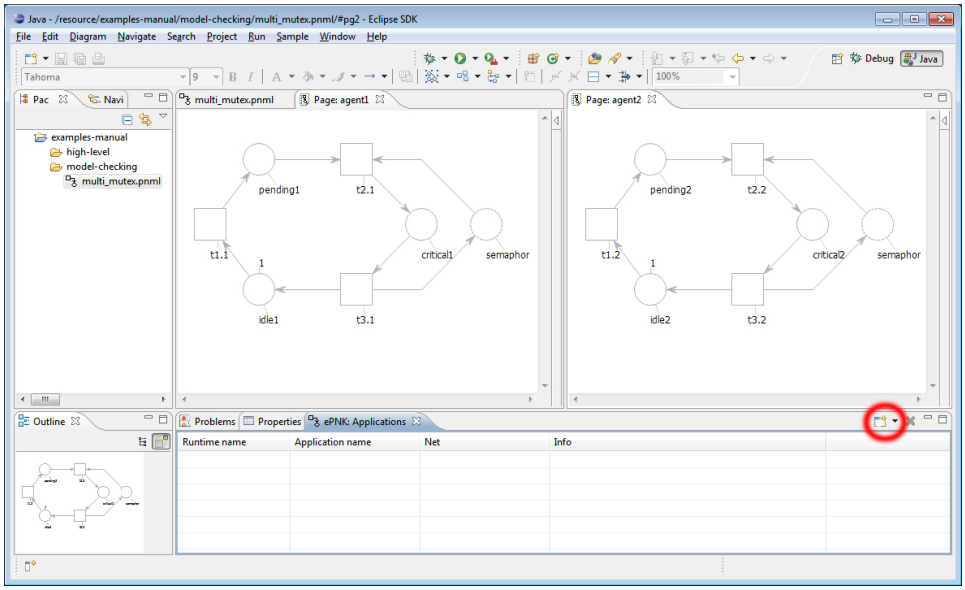

Figure 2.12: Application view a P/T-net in graphical editor

In Fig. 2.12, no applications are running yet. When an editor of a net is selected, you can start an application by selecting an application from the

drop down menu, which is marked by a red circle in Fig. 2.12. This menu will show all registered applications for the selected Petri net type as well as the option to load an application that was saved earlier, which we discuss later.

Once you have started the P/T-net simulator, the started application shows up in the application view, and the net in the graphical editor is decorated with some additional information. In the case of the simulator, it shows the current marking as a blue textual label at the top-right of the respective place; and the enabled transitions are highlighted with a blue overlay. When the user clicks on these overlays, the respective transition fires. Fig. 2.13 shows the simulator after the user fired transitions t1.2 and t2.2. Note that you might not see all graphical feedback, since some pages are not open in the graphical editor or the graphical editor is not on the top. You need to open the pages on which you want to see the feedback yourself. In addition, the application view shows some more tools in its tool

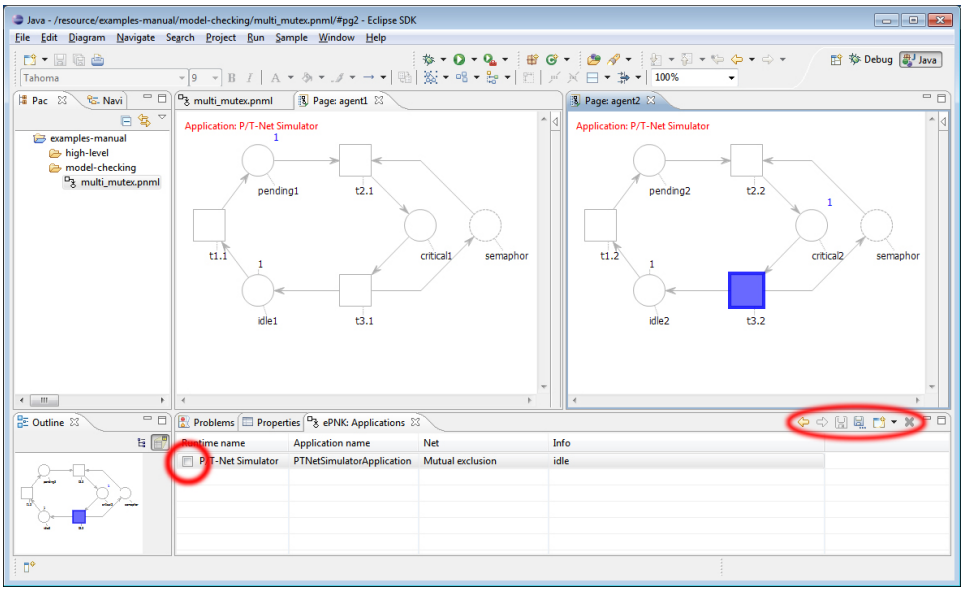

Figure 2.13: Simulator application on P/T-net running

bar (marked in Fig. 2.13) by a red ellipse. The back and forward buttons allow the user to navigate to the previous or next markings, and the save buttons allow to save the state of the simulator. The user can also start further applications by the respective drop down menu. And the user can shut down an application by selecting one or more applications by checking

the boxes to the left of an application, and then clicking on the delete tool. Which tools are shown in the toolbar of the application view depends on the specific application; but the ones shown in Fig. 2.13 are there by default and therefore, most applications will have them.

Note that an application is always started from and associated with a net that is open in an editor. If the editor is closed, all applications on the respective net are shut down. But, as mentioned above, you can save the state of an application, so that it can be restarted in that state later. Fig. 2.14 shows the Eclipse workspace after saving the state of a simulator by pressing the "Save as" button for the simulator application. In that case the user will be prompted for a folder and file name. The default is the same name as the net with file extension ".apnml" for annotated PNML. After the first save, the state of the application can be saved again, by simply pressing the "Save" button. Later the application can be started by the

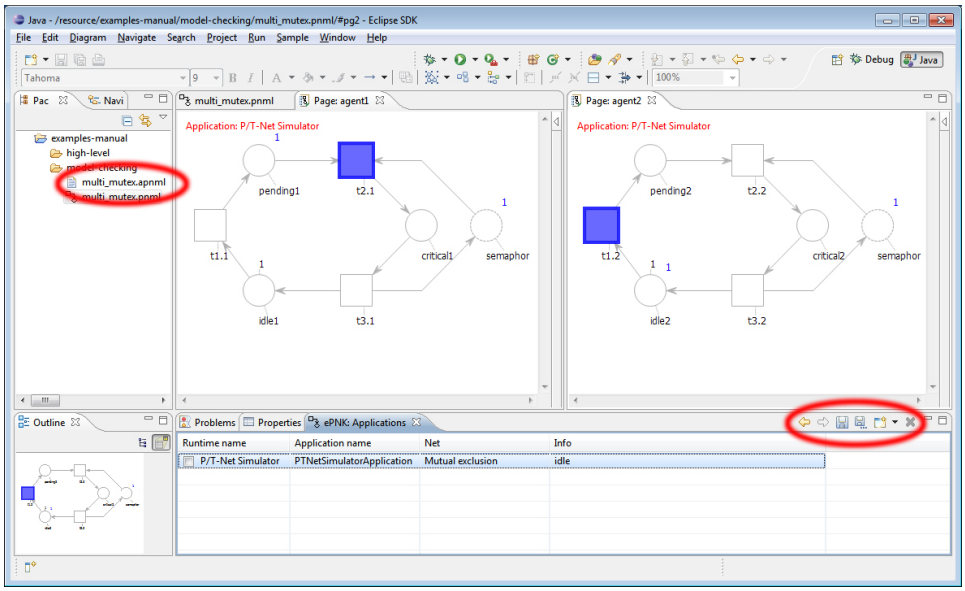

Figure 2.14: Simulator application with a saved state

drop down menu selecting "Load application" and then selecting the file to which the state was saved before.

Note that there is always at most one application *active* in the ePNK, and only the decorations of this active application are shown in the graphical editor. But, there can be many applications running at the same time. All running applications are shown in the applications view, and by selecting

an application there, it becomes active. You can also deselect an active application by clicking on it with the CTRL-button pressed at the same time.

2.6.3 A simulator for high-level nets

In this section, we discuss the simulator for high-level Petri nets, which is deployed together with the ePNK. It was developed by Mindaugas Laganeckas as part of his master's project [22]. The simulator is able to simulate high-level Petri nets as well as so-called *high-level net schemas* [19, 20, 24, 6, 8], which can be instantiated with some communication network in order to simulate a network algorithm on a specific network [25].

Note that all examples that are discussed in this section can be obtained from the ePNK home page together with release 1.2.0 of the ePNK: `http://www2.compute.dtu.dk/~ekki/projects/ePNK/`.

2.6.3.1 The basic simulator for high-level nets

We start with explaining the simulator for normal high-level nets in this subsection and explain the simulator for net schemas later in Sect. 2.6.3.3.

Figure 2.15 shows the simulator application running on a simple high-level net. The high-level net models a simple algorithm that computes the prime numbers according to the principle of the "Sieve of Eratosthenes": It starts with a multiset of all the numbers from 2 up to some upper limit (11 in our example) on the place called numbers. Then, transition t removes a number (the value of x*y) from this place, if this number is a multiple of some other number (the value assigned to x) on that place. When no number on the place is a multiple of another number on that place, the transition cannot fire anymore – and the algorithm terminates. The numbers that are left on place numbers are prime numbers. In Fig. 2.15, there is only one last non-prime number left: 6. The current marking of the place is shown as a blue label at the top-right corner of the place (a long stack of tokens represented as a multiset term in the concrete syntax for HLPNGs of the ePNK).

We assuming that you have obtained and installed the examples from the ePNK home page already. Next, we explain how to start the simulator on these examples and how to open the additional simulator view, which shows the firing sequence up to the latest point in the simulation. Then, we will explain how to interact with the simulator.

If you did not do that already, you need to open the *ePNK applications*

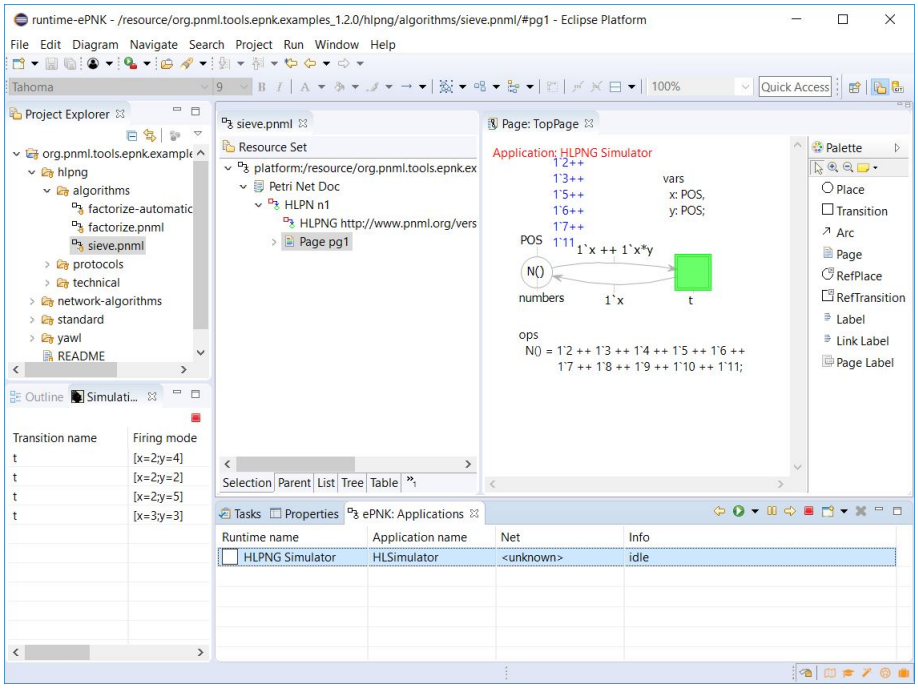

Figure 2.15: ePNK with a simulation application running on a HLPNG

view as described[16] in Sect. 2.6.2. The *Simulation view* can be opened in a similar way: Choose "Window"→"Show View"→"Other..."; then, in the "Show View" dialog, select "Simulation View" from the "HLPNG Simulator Category". By clicking on the tab at the top of the views, you can arrange them in a way similar to Fig. 2.15, since it will be convenient, if you can see the applications view and the simulation view at the same time.

The easiest way to start the simulator on a high-level net is to open the graphical editor on one of the nets pages. Then, right-clicking on the drop-down menu in the ePNK applications view will show all the applications that are available on this type of net. Selecting the The simulator can be started on a high-level net by right-clicking on the HLPNG element in the ePNK tree editor and then selecting "HLPNG Simulator" will start the simulator on this net.

Once the simulator application is started and selected in the ePNK application view, the simulation view shows the firing sequence of all transitions

[16] In short: Choose "Window"→"Show View"→"Other..."; then, select "ePNK: Applications" from the "ePNK" category.

(along with the firing mode) from the initial marking up to the last step of the current simulation. If no simulation application is selected, the simulation view shows the firing sequence of the last active simulation. You can click on the different entries and navigate up and down with the resp. buttons of the keyboard; then the net will show the marking before the selected transition is fired. The current marking for each place is shown as a blue label at the top right of every place – if there is no label, the place's marking is currently empty.

When a simulator application is selected in the applications view, you will find several action buttons on the top right of the applications view, which can be used to control the simulator (as shown in Fig. 2.15). The *back* (left arrow) and *forward* (right arrow) buttons allow you to navigate back and forth in the firing sequence (which had been simulated already). The *play button* (white triangle in green circle), starts the automatic and random firing of some transitions – as long as there are enabled transitions. The *simulation speed* can be selected by a drop down menu on the small triangle right of the play button. It can actually be changed while the automatic simulation is running. The automatic simulation can be stopped – actually "paused" – by the pressing the *pause button*. The automatic simulation can be started any time by pressing the play button again. By pressing the *stop button*, (red box), the simulator is reset to the initial marking as defined by the net. Pressing the cross icon (delete button) to the right will stop and shut down the simulator; note that this is actually a general functionality provided by the ePNK application view, which is available for all running applications.

The automatic simulation will randomly choose any of the currently activated transitions, and randomly choose a firing mode. If the simulation is paused, however, the activated transitions are high-lighted by a green overlay. You can click on these green transitions[17]; then, a menu will pop up, which shows all the possible firing modes for that transition from which you can select. Then, the transition will be fired in the selected mode. Clicking somewhere else or pressing the ESC button, will cancel the selection.

Note that if you go back to some earlier state of the simulation by selecting a transition in the simulator view or by the back button in the simulator application, the marking at that point in the firing sequence will be shown. You will see that one transition is high-lighted by a blue (and darker) overlay. This is the transition to fire next in the firing sequence as shown in the

[17] You will also be able to click on a transition which is high-lighted in grey or blue, as will be discussed later.

simulation view. There might also be some other transitions high-lighted by a green overlay, which would have been alternative choices at that point. Note that also a blue transition might "hide" alternative choices that are not graphically high-lighted, since it might be able to fire it in different firing modes.

If you are in such an intermediate state of a firing sequence, you can still interact with the transitions by clicking on them as discussed above and by selecting a firing mode, which will fire this alternative transition in that marking. Note that, in this case, all the later firing steps of the earlier firing sequence are deleted. From the current point on, a new branch of simulation will be followed. This way, you will be able to explore different branches of the reachability graph of the Petri net.

Note that, in some cases, the simulator is not able to compute the firing modes fully automatically, and is not able to decide whether a transition is enabled. In that case, the respective transition is high-lighted with a grey overlay. This does not happen in the prime factors example, but it will happen all-over in the example "factorize". Once you click on one of the transitions with a grey overlay, you will be prompted for possible values for the different variables. You can enter a semicolon separated list of values for each of the variables; then the simulator will try to compute possible firing modes based on these values. Note that you do not need to provide values for all variables; in many cases, it is enough to provide the value for one variable from which the values for the other variables can be derived. If the simulator can compute some enabled firing modes, the transition will be high-lighted in green, so that you can actually select the mode in which this transition should fire. You can still select "Manual input" for providing more or other possible values. If no modes could be found, the transition will remain high-lighted in grey; only if you provide values for which the transition can fire (or enough information for the simulator to figure that out), the transition will actually become enabled.

2.6.3.2 HLPNG operations supported by the ePNK simulator

High-level nets as defined in ISO/IEC 15909-2 have quite many built-in operations. The simulator of high-level nets for the ePNK does, unfortunately, not yet support all these operations. This, in particular, applies to the sorts and operators for symmetric nets.

The following sorts of ISO/IEC 15909-2 are supported in the current version (1.1.1) of the simulator: DOT, BOOL, NAT, POS, INT, and STRING as well as the generic sorts product, multiset and lists over existing sorts.

The following operators are supported: == and != on all sorts; or and and on BOOL; +, -, *, /, %, <, and > on NAT, POS and INT; concatstring for Strings; ', ++, --, all, and empty on multisets; the tuple operator for products; emptylist, makelist, memberat, sublist, length, appendtolist, and concatlists for lists.

The sorts and operators introduced for symmetric nets are not supported by the simulator at all.

2.6.3.3 The simulator for network algorithms

In this section, we discuss the *simulator for network algorithms*. Before discussing the simulator itself, we discuss the concept of network algorithms and the way they are modelled as algebraic nets schemas or – in the terminology of ISO/IEC 15909 – high-level Petri net schemas (HLPNGS). To this end, we use an example which – except for syntactic sugar – is almost identical to the first publications that used algebraic net schemas for modelling and verifying network algorithms [19, 31, 24]: it is a simple algorithm that, for a given network of computing agents with some distinguished root agents, computes the minimal distance of each agent from a root agent. The Petri net modelling the algorithm is shown in Fig. 2.16. In the example projects

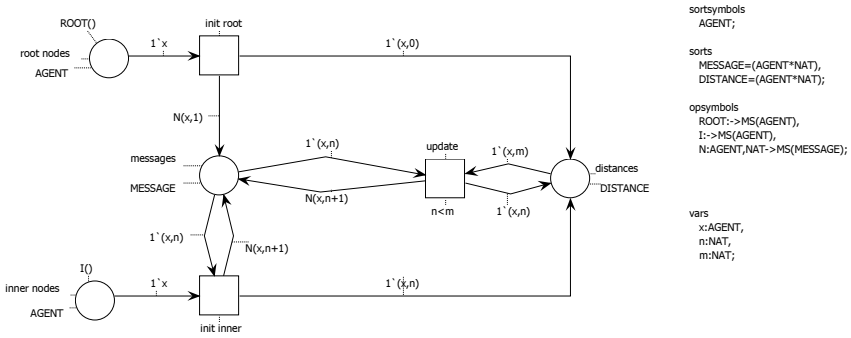

Figure 2.16: Network algorithm computing the minimal distance to a root

deployed for the ePNK 1.2.0, you will find it in subfolder min-distance of folder network-algorithms.

One of the main features of a Petri net schema is that, the Petri net model itself is completely independent from the actual network of agents on which the algorithm is working. In the model from Fig. 2.16, the set of agents of the network is represented by the sort AGENT, but it is just a symbol – which still needs some interpretation. The multiset constant

ROOT represents the set of root nodes, and the constant I represents the set of non-root nodes (or inner nodes).

Moreover, there is an operation N, which takes an AGENT and a natural number as a parameter. This function represents sending a message from one agent to all its neighbours in the network – where the message itself is a natural number. A MESSAGE to an agent is represented by a pair, where the first component is the receiver AGENT and the second component is the actual content of a message. If there is a distance computed for some agent already, this is also represented as a pair of an AGENT and a number NAT – for making the difference clear, we call this pair DISTANCE.

Initially, the place root nodes contains all the root nodes of the network; the place inner nodes contains all the inner nodes. The transition init root models the initial step of a root node x: it sets its own distance to 0, which is represented as a pair $(x, 0)$, and adds a message to all its neighbours that they might have distance 1 from a root node – all these messages are represented by $N(x, 1)$. Transition init inner models the initial step of an inner node: when an inner node x initially receives some message with some distance n, it stores this distance as a potential shortest distance, and sends a message to all its neighbours with distance $n + 1$, which is represented by $N(x, n + 1)$. An inner node x can later receive other messages with another distance n; if the other distance n is less than its current distance m, the agent takes distance n as its new distance, and sends a message with distance $n + 1$ to all its neighbours. This is modelled by transition update.

As said before, the Petri net model from Fig. 2.16 models a minimal distance algorithm for any network. If we want to simulate the algorithm, we need to know on which network it should run. To this end, a very simple network editor is deployed together with the high-level simulator of the ePNK. Figure 2.17 shows the network editor with a simple example network. In this case, it is a network with directed arcs. The network simulator can be started in the same way as the simulator for general high-level nets. Once, a page of the net is open in the graphical editor, right-click the drop-down menu in the ePNK application view, and select "Network algorithm simulator". If there is a network file with the same name as the Petri net model in the same folder, the network simulator chooses this network for the simulation. If there is no such file, the user will be prompted for a file with a network model. Once the network model is selected, the interpretations of the sort AGENT, the constant symbols ROOT and I, as well as the function N (sending a message to all the neighbours of the agent) are defined by this network. For example, for the network of Fig. 2.17, the set associated with the sort AGENT is $\{A, B, C, D, E\}$; the constant ROOT

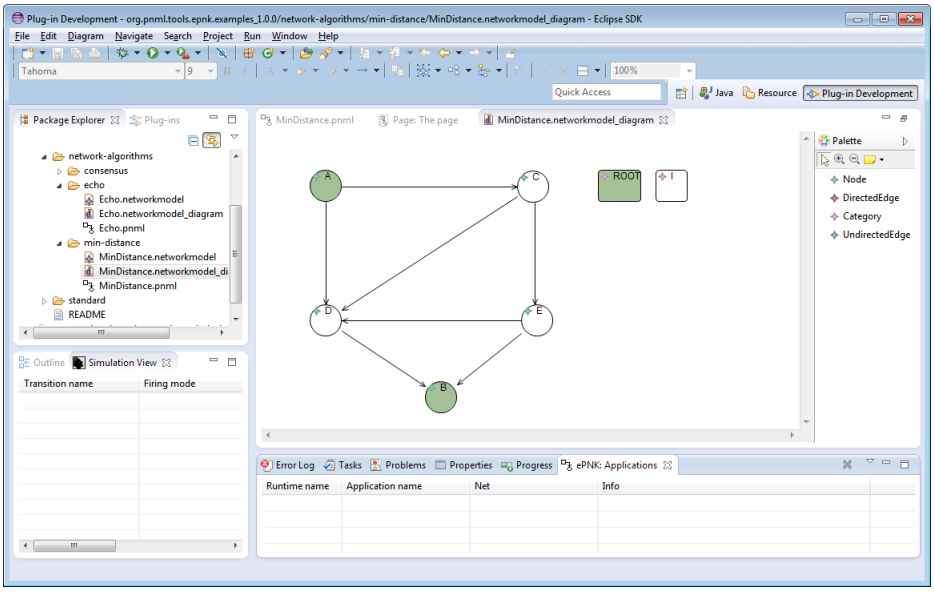

Figure 2.17: A network on which the algorithm could work

denotes the multiset $[A, B]$, the constant I denotes the multiset $[C, D, E]$; for $x = A$ and $n = 5$, the term $\mathsf{N(x,n)}$ will evaluate to the multiset $[(C, 5), (D, 5)]$ – representing the message 5 to each neighbor of agent A. And these will be the interpretations the simulator will be using for simulating the net.

Figure 2.18 shows the network simulator running on the minimal distance algorithm from Fig. 2.16 on the network from Fig. 2.17. The interaction with the simulator and the way to control the simulation is exactly the same as described for the basic simulator in Sect. 2.6.3.1, once the network simulator is properly initialized with a network.

The editor for the *network* is a simple *editor* generated by GMF and follows the GMF philosophy. A new network can be created by "File"→ "New" → "Other..." and then selecting "Network Diagram" from category "Examples". In this diagram, you can add nodes, directed and undirected edges between the nodes, as well as categories for nodes. When a node is selected, each node can be associated with any number of categories by a menu that pops up when clicking in the category property in the properties view. In Fig. 2.17, the association with categories is also shown by using the same colour for the category and the resp. nodes; but the colour itself does not have any meaning in the diagrams, its only purpose is to make the network diagrams easier to grasp on a single glance.

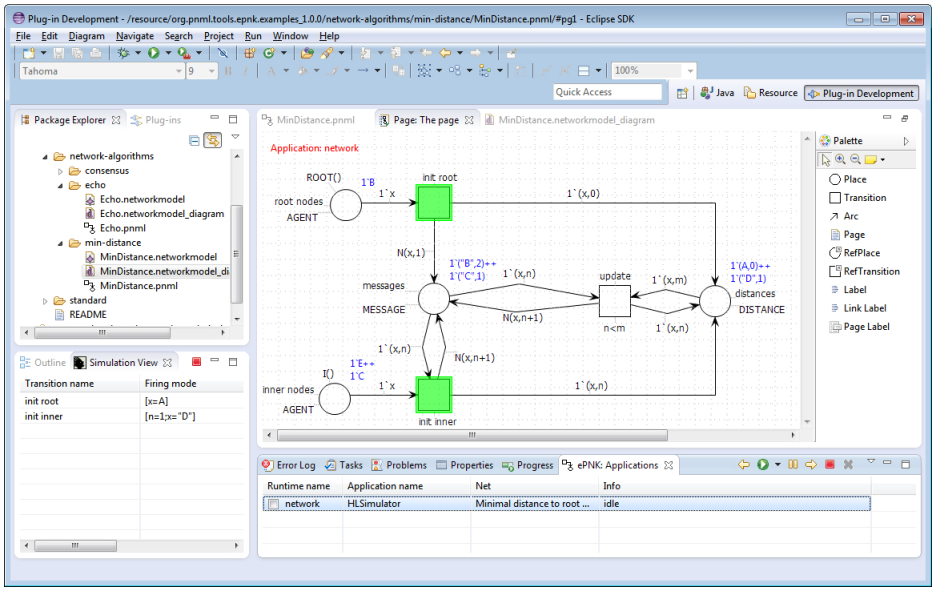

Figure 2.18: Network simulator running on the minimal distance algorithm

For a given network, the sort AGENT is associated with the set of nodes; for every category, the respective constant symbol is associated with the multiset of nodes that are in that category (in our example, these are the root nodes and the inner nodes). For the operation symbol N, $N(x, m)$ denotes a multiset of pairs, where for each outgoing arc from x to y there is a pair (y, m) in the multiset. $S(x)$ is a multiset of pairs over agents, where there is a pair (y, x) in the multiset if there is an undirected arc from x to y; these represent all messages that are sent from agent x to all its neighbours. Likewise $R(x)$ contains all the messages received from all its neighbours.

A Petri net modelling the echo algorithm, which is another example deployed together with the ePNK, makes use of the send and receive operations S and R. But, we do not explain the echo algorithm here – see [31, 20] for details[18].

2.7 Limitations and pitfalls

The current version of the ePNK (1.2.0) still has some minor limitations. Moreover, some functionality of the ePNK's functionality might not be ex-

[18]The operation S is denoted with M and the operation R is denoted with $\overline{M}$ in [20] – and the other way round in [31].

actly where you would expect it. In order to avoid some unpleasant surprises these issues are briefly discussed in this section.

2.7.1 Saving files: Tree editor

Technically, the tree editor and the graphical editor of the ePNK are different editors, they just work on the same underlying model. The graphical editors can be initiated via the tree editor only (via the pop-up menu "Start GMF Editor on Page" or a double click on a page in the tree editor or the graphical editor). The tree editor always serves as the master editor; in particular, saving a PNML document is possible only from the tree editor – and only the tree editor shows a valid "dirty-flag" when the file contains unsaved changes (see Sect. 2.3.3).

2.7.2 Reset an attribute

In the user interface of the ePNK there is no explicit way to create *attributes* for net object, as you can do for labels. The reason is that attributes are not supposed to be shown as labels in the graphical editor at all. They are shown only in the properties view when the respective object is selected. When a value is entered for an attribute for the first time in the properties view, the attribute is created; and the value of the attribute changes whenever the value is changed in the properties view.

The question, however, is how the attribute can be deleted? Entering an empty string is not removing the attribute. To actually remove an existing attribute from an object, you would select the respective object, and then, in the properties view, right-click on the respective property and select "Restore default value" (see Sect. 2.4.3).

2.7.3 Graphical features

The graphical editor of the ePNK supports many graphical features, such as positions and size of nodes, intermediate positions for arcs, fonts, size and colours for labels, colours and line-width for nodes and arcs. Not all of these graphical features, however, are transferred back to the actual PNML document—some of this information cannot not even be represented in PNML.

All graphical information of the ePNK is stored as tool specific information in PNML, so that the net in the graphical editor of ePNK always shows the way as you have edited it even when the PNML does not support the used graphical features.

The graphical information that is used and made available in PNML so that it can be used by other tools that support PNML is the following:

- Position and size of nodes.
- Line colour and background colour for nodes.
- Image information for nodes.
- Position resp. relative position of labels[19].
- Text colour[20] and background colour for labels.
- The font and the font-size of labels.
- Intermediate points of arcs.
- Line colour of arcs.
- The shape attribute of arcs (straight or bezier).

Note that all the graphical information – even the one that is not transferred to PNML features – is saved by the ePNK as tool specific information. You can even put notes on the graphical canvas, which will be saved. The graphical information not covered in the above list, however, will not be available to other tools, and will be gone, when the tool specific information (org.pnml.tools.epnk.diagraminfo) for the respective page is deleted.

The ePNK reads all graphical attributes that are supported by PNML; but, only the ones discussed above are used and shown in the graphical editor. Moreover, the unused graphically attributes are not checked for validity; they will be written in exactly the same way as they were loaded from the PNML file (see also Sect. 2.7.6)—even if they were not valid PNML.

Due to some limitations in the automatically generated GMF editor on which the graphical editor of the ePNK for pages is based, there is another quirk of the graphical ePNK editor: When a node is moved, the intermediate points of the attached arcs are also changing in the diagram; these changes are, however, not propagated to the PNML model. If you want to make sure that the intermediate points of an arc of the diagram and the PNML are exactly the same, you need to explicitly touch and slightly move one intermediate point of each attached arc; only then, the exact positions of the intermediate points are properly propagated to the PNML model.

[19]The size of labels is not supported by PNML

[20]For the text colour, the ePNK is actually abusing the line colour attribute of PNML, since PNML does not support a text colour attribute.

Note that some Petri net type definitions might define some graphical appearance for some of its elements. In that case, this graphical appearance overrides the graphical attributes of PNML.

2.7.4 Petri net type: Should not be changed in an existing net

As mentioned in Sect. 2.3.2, the type of a Petri net should never be changed after a net of that type was created – unless you know exactly what you are doing. Otherwise, it could happen that the produced PNML is invalid.

For HLPNGs, it is no problem to change its kind any time, since the kind has an effect on the validation only, but no effect on the serialisation of the net to a PNML file. Changing the "kind", actually, does not change the Petri net type – just the subsets of features that are supported.

2.7.5 Line-breaks in labels

All labels in ePNK Petri net types can have line-breaks (if the Petri net types syntax allows it. In order to create line-breaks in labels with the graphical editor, you need to press CNTRL and ENTER at the same time; ENTER alone would actually finish editing the label.

2.7.6 Graceful PNML interpretation

PNML files that are produced by the ePNK and which have been successfully validated are conformant to PNML 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 – even if they are not conformant to ISO/IEC 15909-2. But, if a PNML file is created by using the ePNK only and if it validates 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 PNML documents that do not properly validate; and the ePNK is able to load these files again even though other PNML tools might not be able to load them. For example, when some elements do not have ids (as required by PNML), references to these elements cannot be made via their id. In that case, the ePNK uses XPath references to these elements, which is not conformant to PNML. If such an invalid PNML file is loaded later by the ePNK again, and the ids are added, and validation

is successful, saving this file will produce a conformant PNML documents again.

2.7.7 Deviation from PNML

There is one minor deviation of the ePNK from PNML as of ISO/IEC 15909-2:2011: In the ePNK, all declarations of HLPNGs can have a name attribute, which comes from the fact that `Declaration` implements the interface for symbol definitions (`SymbolDef`). As a consequence, also the `Unparsed` declaration can have a name attribute. In ISO/IEC 15909-2, `Unparsed` does not have a name attribute – as the only exception among all declarations of PNML. Therefore, if an `Unparsed` operation declaration with a name attribute would be manually added to a PNML document in the ePNK tree editor, the resulting PNML document would not be conformant to ISO/IEC 15909-2:2011 anymore.

In practice, however, this should not be any problems, since the only way to add an unparsed declaration to a HLPNG net would be to add it manually via the ePNK tree editor. If all declarations are edited in the graphical editor by editing the respective labels, this problem will never arise. If the PNML document would contain an `Unparsed` operation declaration without a name attribute (as it would be according to the standard), the ePNK would show this as an error under validation. But, it would still be able to read and write the PNML document.

In future versions of ISO/IEC 15909-2, this might be resolved by requiring that all declarations – including `Unparsed` – should or, at least, could have a name attribute.

Chapter 3

Developers' guide

In this chapter, we discuss how to extend the ePNK, with new functionality and applications, with new Petri net types, with new graphical appearances, or with 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.

This chapter addresses developers, who want to develop extensions for the ePNK, and it gives a systematic overview over the concepts of the ePNK by discussing different examples for the different concepts. This chapter is organized along the different concepts for extending the ePNK and for using its API. In contrast, Chapter 4 is organized along a single running example, which covers the most relevant concepts and goes through the conceptual steps as well as through the technical details. Both chapters are written in such a way that they can be read independently of each other.

Section 3.3 shows how to add some functionality to the ePNK, which could be a model checker, or some other analysis or verification function for Petri nets, or which 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 PNML format. Section 3.4 shows how to implement *applications* that visualize the results on top of the graphical editor of the ePNK and can also interact with the end user while they are running.

Section 3.5 shows how to add new Petri net types to the ePNK. Simple net types can almost completely be generated from a model; for more com-

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

plex Petri net types, such as high-level Petri nets, a mapping to XML must be defined and parsers need to be implemented.

Section 3.6 shows how to customize the graphical appearance of some features of some types of Petri nets.

Section 3.7 shows how to add tool specific information to the ePNK.

At last, Sect. 3.8 gives an overview of the projects of the ePNK and Sect. 3.9 briefly discusses how to deploy own extensions.

All of these extensions are discussed by the help of some examples, which are deployed together with the ePNK. In these examples, we assume that the reader is familiar with the main principles and ideas of Eclipse, its plug-in architecture, and Eclipse plug-in development. We cannot give a detailed introduction to Eclipse and to Eclipse plug-in development here (when you have the feeling that you need more background on these issues, the "Platform Plug-in Developer Guide" which is part of Eclipse might be a good starting point: "Help" → "Help contents"; and there are many other Eclipse resources [28, 4]). Still, Sect. 3.1 gives a brief overview of Eclipse plug-in development. We go a bit more into the details for 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.

In order to use the ePNK for implementing own extensions, developers would need some understanding of the PNML core model and the API generated from it. Therefore, Sect. 3.2 gives a brief introduction to the PNML core model as it is used in the ePNK and discusses some differences to the PNML core model from ISO/IEC 15909-2.

3.1 Eclipse: A development platform for the ePNK

As briefly discussed in Sect. 2.1 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 have a look into the code of the examples, which are discussed in chapter. For a hands-on experience, you should install Eclipse and the ePNK as explained in Chapter 5.

3.1.1 Importing ePNK projects to the workspace

As a developer, you would probably want to have a look into some parts of the source code of the ePNK. Therefore, the ePNK is deployed in such a way that you can easily import the relevant ePNK plug-in projects to your

workspace with all their source code and all their models, so that you can inspect the code and the models from which major parts of the code were generated. Furthermore, the source code of all ePNK projects is available on GitHub: `https://github.com/ekkart/ePNK`

Here, we explain how to import the ePNK projects into your Eclipse development workspace as source projects – assuming that you have installed the ePNK in your Eclipse already. Initially, it is recommended to import only the basic project `org.pnml.tools.epnk` into the workspace; later during this developers' guide, we will mention several other projects that might be interesting for you to have a look at. In the end, Sect. 3.8 gives an overview of important ePNK projects and their main function and purpose.

In order to import an ePNK project (or any other plug-in project which is running behind the scenes in your Eclipse), you first need to open the Eclipse *Plug-ins view*. To this end, in the Eclipse workbench, select "Window" → "Show view" → "Other..."; then select "Plug-ins" from the category "Plug-in Development". Typically, this view would also open when you switch to the *Plug-in Development perspective* of Eclipse. Once the Plug-ins view is open, browse this view and find the ePNK plug-in `org.pnml.tools.epnk`. Right-click on this plug-in and select "Import As" → "Source Project". After that, you will find the project `org.pnml.tools.epnk` in the Eclipse package explorer or some other open resource browser.

In this project, you can see the source code in the folder "src" and also the model files in the folder "model". But, if you did not install some extra tools yet, you will not be able to open these models with some reasonable editor. Section 3.1.2 briefly discusses which additional tools you need to install to your version of Eclipse so that you are able to inspect – and later create – these kind of models and diagram files.

You can import all ePNK projects to the workbench as discussed above, but only some few projects will be relevant for you. This chapter introduces you to the most relevant ones – one after an other (in the end, Sect. 3.8 gives an overview of the ePNK projects). If you, eventually, are confident in developing functions and applications for the ePNK, you might also want to contribute to the ePNK and make your extensions and changes to the ePNK plug-ins. In that case, you can ask to be given access to the ePNK development repository.

In your workbench, you have now the project `org.pnml.tools.epnk`, which is the basis for developing new plug-ins and, in particular, extensions to the ePNK. We start calling this workbench *development workbench* now. The reason for introducing an additional adjective to the term *workbench* here is that you need to start another workbench from this one, in order

to start Eclipse with the new extensions that you are developing in the development workbench. This additional instance of Eclipse is called the *runtime workbench* since this is where the ePNK with your new extensions is running. It will very much look like the original ePNK as discussed in Chapter 2 – just with the extensions from your development workbench also running. The 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". After you have started the development workbench in this way once, it will be enough to press the "Run" button in the tool bar for starting the runtime workbench again. For now, however, we do not start the runtime workbench since we need to implement some new functionality first.

3.1.2 Installing the EMF and Ecore Tools SDK

As mentioned already, a major part of defining a new Petri net type is creating an Ecore model which captures the concepts of the new Petri net type; most of the code can then be generated from such a model. In order to be able to do that, you need to install the *Eclipse Modeling Tools* (*EMT*) in your Eclipse or install the *Eclipse Modeling Tools* package as your version of your Eclipse.

The details are discussed in Chapter 5. Note that most of the diagrams associated with the Ecore models were created with version 1 of Ecore Tools and have not been updated to version 2 of Ecore Tools yet. These diagrams are not supported by Eclipse newer than Luna. But, if you install the feature for legacy Ecore diagrams as discussed in Chapter 5, you will be able to open them. It is, however, strongly recommended that new models and diagrams are created with the current version of Ecore Tools, since the legacy Ecore Diagram Editor eventually might not work any more for future versions of Eclipse.

You can check if you have installed Eclipse and its EMT extensions correctly: Open the project `org.pnml.tools.epnk` and, in that project, open the folder `model`. The files with extensions ".ecore", ".ecorediag", and ".genmodel" should have special icons. And when you double-click on ".ecore" and ".genmodel" files, a tree-editor should open on them. When you double-click on "PNMLCoreModel.ecorediag", you should see an Ecore model in a class diagram like graphical notation as shown in Fig. 3.1.

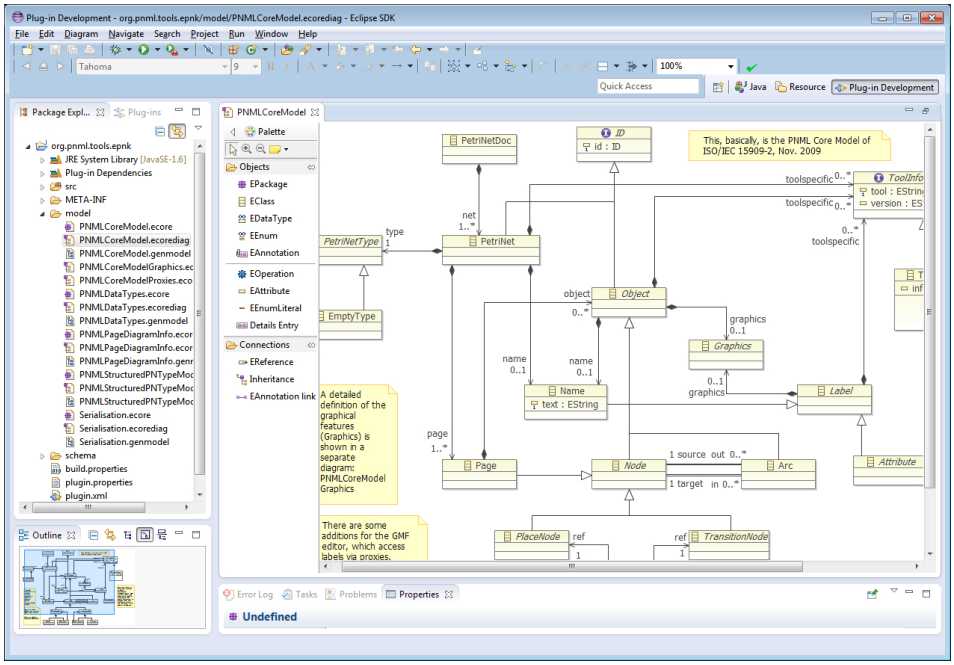

Figure 3.1: Developer's view with the ePNK PNML core model

3.2 The PNML core model in the ePNK

For using the ePNK as a developer it is important to have a more detailed understanding of the *PNML core model* since the API for accessing, navigating and modifying Petri nets is generated from this model. In Sect. 3.1.2, you have seen already were to find and how to open the diagrams of the PNML core model and some related diagrams. And when you start developing with the ePNK, you will probably need to have a look into these diagrams once in a while, in order to understand the details of the relation between the different classes and concepts of PNML.

In this section, we give an overview of the PNML core model of the ePNK and discuss some of the differences to the PNML core model of ISO/IEC 15909-2. As discussed in Sect. 1.2.1 already, the PNML core model of the ePNK is slightly more general than the one of ISO/IEC 15909-2:2011 [8] (see Fig 1.1). One of the main differences is the following: According to ISO/IEC 15909-2, a page is not considered to be a node; in the ePNK, a page is considered to be a node. This way, it is possible to define Petri net types in the ePNK that allow arcs to be connected to pages (e. g. in order

to define a Petri net type that mimics substitution transitions of CPNs [9]). Keeping this in mind, might be particularly important, when defining the constraints for arcs of net types (such as the ones for P/T-nets as defined in Sect. 3.5.1.4), when you do not want to allow to connect pages to other nodes with arcs.

The other differences of the PNML core model are a bit more technical and will be discussed below. You might want to have a look at Fig. 1.1 of the PNML core model of ISO/IEC 15909-2:2011 and at Fig. 3.1, while we discuss the difference. For more details, you can also open the Ecore diagrams in the project `org.pnml.tools.epnk` in your Eclipse workspace, provided you have installed the legacy Ecore Diagram editor.

Most importantly, the `type` of a Petri net is an attribute of the class `PetriNet` according to ISO/IEC 15909-2, whereas, in the ePNK, it is a composition to a class that must inherit from the abstract class `PetriNetType`. Instances of this class represent the Petri net type and will provide some services to the ePNK to generically maintain the features of the respective Petri net type. This is discussed in more detail in Sect. 3.5. The PNML core model of the ePNK also provides one concrete class for a Petri net type, which does not exists in ISO/IEC 15909-2:2011: `EmptyType`, which represents a Petri net without any labels other than `Names`.

In the ePNK, the `source` and `target` reference of the class `Arc` have a corresponding opposite reference from the `Node` to its `out`-going and `in`-coming `Arc`s. These opposite references are not serialized to the XML document, however, since this would not be compliant with the PNML format. The opposite references will, however, be restored when loading the Petri net. These opposite references are very convenient when navigating between different elements of the net.

In turn, the ePNK does not have the class `Annotation` of ISO/IEC 15909-2, which represents those *labels* that should be shown as graphical annotations of the respective node or arc in the graphical editor. There exists a class `Attribute`, however, which represents those labels that should be represented as properties of the respective element only, but not as annotation in the graphical editor[2]. In the ePNK, all *labels* that are not attributes are considered to be annotations. This has historical reasons, since the first version of the ePNK did not support attributes at all; but, it also avoids making mistakes in Petri net type definitions: it is impossible to define labels that are neither attributes nor annotations.

[2]By plugging in some graphical extensions, attributes can still have an effect on the graphical appearance of a Petri net.

At last, the PNML core model of the ePNK defines a separate interface `ID`, which is used to unify all those elements that need a unique identifier in a PNML document. All classes that must have such an identifier, implement the interface `ID` in the PNML core model of the ePNK. The reason for adding the class `ID` to the ePNK PNML core model is that, this way, the ePNK can handle elements with an identifier in a uniform way; with the separate id attribute for all elements as defined in the PNML core model of ISO/IEC 15909-2, this would require much more effort.

The attentive reader might also have noticed that the PNML core model of the ePNK does not contain any OCL constraints, whereas the PNML core model of ISO/IEC 15909-2 does. This, however, does not mean that the ePNK ignores these constraints; in the ePNK, constraints are just added in a different way: they are plugged in on top of the model. We will see examples of how ePNK constraints are plugged in to the ePNK resp. to Eclipse later (e. g. in Sect. 3.5.1.4).

3.3 Adding functions

Next, we discuss how new functionality can be added to the ePNK. As discussed earlier, there are two different ways of new adding new functionality to the ePNK: *Functions* take some net, possibly some user input, do some computing and then report a result – typically via some dialog window. After that, the function is over and done with. The model checker from Sect. 2.6.1 is a typical example of a function. By contrast, *applications* are started on a net; then, they show some feedback on top of the graphical editor, the user can interact with the application, and the visual feedback will be updated accordingly. Typically, an application finishes only when the user explicitly terminates it. The simulator for high-level nets of Sect. 2.6.3 is a typical example of an application.

In this section we, discuss the implementation of functions for the ePNK. Basically, we use the standard mechanism of Eclipse to plug in and start functions – as *views*, *wizards*, *actions*, *command handlers*, *jobs* or *dialogs*. In this manual, we explain the use of these concepts on the side, as far as they are necessary. Since setting up jobs in a proper way can sometimes be a bit tedious, the ePNK provides some utility classes that make programming jobs a bit easier.

The focus of this section is on the use of the API of the ePNK that is generated from the PNML core model and the Petri net type definitions by the *Eclipse Modeling Framework* (*EMF*). On the side, we point out some

of the general principles of EMF and some practical issues on working with EMF. For a more detailed introduction to EMF, we refer to [2]. In this section, we do not only show how to access, navigate and manipulate nets; we discuss also how to open, create and write PNML files programmatically.

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

- Section 3.3.1 shows how to implement a new Eclipse view, which gives an overview of a PNML file that is selected in one of the Eclipse resource explorer views. This section will also discuss how top open and access the contents of a PNML file.

- Section 3.3.2 shows how to implement a *wizard* for creating a PNML file (actually, the coded is based on a wizard that was automatically created by the Eclipse "new plug-in project wizard"). This wizard creates a PNML file with a P/T-System that represents a simple mutual exclusion algorithm for a number of agents – where the number can be selected by the user in one of the dialogs of the wizard. The Petri net is split up to different pages, so that the Petri net for each agent is contained in a separate page. In particular, we will discuss how to create a PNML file, how to fill it with some contents and how to save it.

- Section 3.3.3 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.

In the end, Sect. 3.3.4 gives an overview of the different functions of the ePNK (and the API generated from its EMF model), some hints on how to work with this API in Eclipse and in EMF general, as well as some additional ePNK utilities and helper classes that make it more convenient to handle and access some of the information stored in a PNML document. Experienced Eclipse and EMF developers, might want to start reading the overview in Sect. 3.3.4 first, and come back to Sect. 3.3.1–3.3.3 for some details later.

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

In this section, we discuss how to implement a new (and very simple) Eclipse *view*, that will give an overview of the contents of a PNML file that is selected in the explorer. Figure 3.2 shows a screenshot of the result. For the selected file "hlpng-gmf.pnml" in the "Project Explorer", the "ePNK File Overview" at 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; the type of a net is represented by its unique URI. The name of the first net is "A high-level next example"; the other two nets do not have a name.

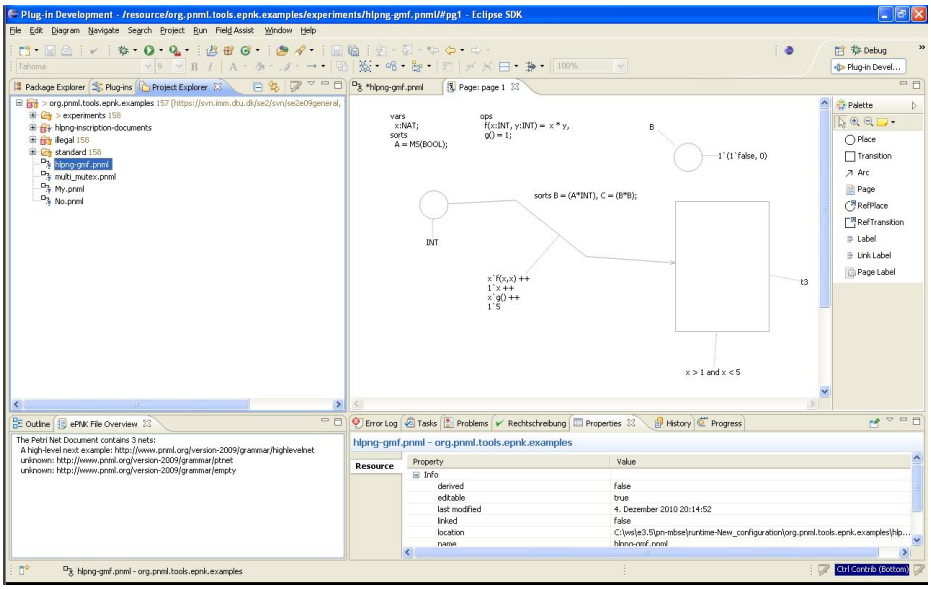

Figure 3.2: The ePNK with the "File Overview" view

This view and its functionality is implemented in the plug-in project `org.pnml.tools.epnk.functions.tutorial`. We go through this project now[3]. In addition to the overview view discussed above, this project also contains the implementation of the wizard for creating a PNML document which will be discussed later in Sect. 3.3.2.

The implementation of the view is contained in a single Java class: `PNMLFileOverviewView` in the package `org.pnml.tools.epnk.functions.`

[3]Remember that you can import the source code of that project to your development workspace from the "Plug-ins" view by selecting the project, right-clicking on it and then choosing "Import As" → "Source Project" (see Sect. 3.1.1.

`tutorial.overviewview`. We briefly explain the general structure of this view class, an extract of which is shown in Listing 3.1; we deleted all imports, an attribute definition, and all comments; if needed this information can be looked up in the source code. The class extends the Eclipse `ViewPart`, which actually makes it an Eclipse view, and it implements the `ISelectionListener`, which allows our view to obtain the information on the object that the user has currently selected in the workbench. Note that this class does not have an 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. Since these are standard Eclipse providers, we 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 by calling the `selectionChanged()` method for the current selection. We will discuss the respective method `selectionChanged()` below. Note that there are two other methods. The `setFocus()` methods 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 when the user decides to close the view).

Once the view has registered itself as a selection listener with the Eclipse workbench, its `selectionChanged()` method is 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 accessing the actual contents of the file and for computing the contents of the overview is called; this produces an array of Strings, which then will be set as the new contents of that view – and, this way, shown to the user.

The `getOverviewInfo()` method 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 here). The implementation of this method is shown in Listing 3.2. Up to line 7, it is checked whether the file extension is either "pnml" or "pnx" (the two file extensions, the ePNK uses for storing Petri net files – "pnml" represents files in PNML format, and "pnx" represents Petri net files in XMI, which we call PNX); then, the path to that file is extracted and a URI of that path is created. Actually, Eclipse

Listing 3.1: 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 3.2: 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;
}
```

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 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 to 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. 3.3.2 for more information) via a resource set. After we successfully got the resource, we can obtain its contents by the `getContents()` method which returns a list of its top-level objects – in case of PNML, this list should contain exactly one `PetriNetDoc` object. 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 *PNML core model*, the ePNK API has corresponding *getter* and *setter methods*. For example, if a class has an attribute `name` of type `String`, the `getName()` method will return the `String` with that name, and with `setName()` we could set it – but we do not change the net in this example. For attributes and features with a multiplicity greater than one, this is slightly different. For example, a `PetriNetDoc` object can contain many nets; therefore, `getNet()` will return a list of nets, which we then can iterate over to obtain the individual nets. And by adding a net to this list, this Petri net would be added to the Petri net document (see 3.3.2 for examples).

As stated above, class `PNMLFileOverviewView` implements the "ePNK File Overview" as we have seen it in Fig. 3.2. But, it is not enough to 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" file. 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 is a bit easier with the XML fragment that is produced by this editor. The fragment relevant for our overview view is shown in Listing 3.3. It says that we contribute our *extension* to the Eclipse *extension*

Listing 3.3: Defining the overview view 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>
```

point `org.eclipse.ui.views`, which is a new Eclipse view. The category defines where and under which category the new view can be found, when the user wants to open it via the Eclipse "Show View" menu. We define a category specifically for the ePNK and then define the actual view referring to this category. The attributes of the view say that a view of this kind can at most be open once, that it uses the above category, and refer to the class which actually implements it: `PNMLFileOverviewView`; and the attributes define 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 such as `Resource`, `ResourceSet`, and some of the ePNK classes like `PetriNetDoc`, `PetriNet`, etc. in the implementation of the view, we would also need to define dependencies to the plug-in projects by which 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 of Eclipse; from there, you could open the "ePNK File Overview" 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. 3.2. Whenever the user selects some PNML or PNX file in the package explorer, this view

will show an overview of the contents of the selected file. Since this plug-in project is deployed with the ePNK 1.0.0 already, it is of course not even necessary to start the runtime workbench – the view is already there in the development workbench, once the ePNK is installed.

3.3.2 Writing PNML files: Generating multi-agent mutex

Next, we discuss how to create new PNML files and how to fill them with some contents. In typical applications, the contents might come from a file in a format of another Petri net tool, which should be converted to PNML. In our example, however, we programmatically generate a Petri net: my favourite semaphore mutex example, which was used in Sect. 2.6.1 as an example for model checking already. To make things 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: ...` . These packages are contained in the same plug-in project, `org.pnml.tools.epnk.functions.tutorials`, as discussed in Sect. 3.3.1 already.

In the rest of this section, we focus on the explanation of the parts of the implementation that are concerned with the creation of the file and its contets. This functionality is implemented in the method `createPNMLFile()` of the class `MultiAgentMutexNetWizard`, which is shown in List. 3.4. 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 the Petri net programmatically is quite straightforward, but code intensive. Therefore, we have split up the creation 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 omit some boring ones in the end (you can find all details in the source code). Listing 3.4 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 contents of the file that we want to write. Then, we convert the path into a URI. Then, we create a resource set – from which the resource (the file) is create. Surprisingly enough, this is already all we need to do. At this point, we can add the contents 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 of the resource so that either a PNML file or a PNX file is written once we save the resource in the end. But, we configured the wizard in such a way that the user can chose only the "pnml" extension.

Adding the contents follows the same principle that we have discussed already. With `getContents()` we get a list of EMF objects (which would be empty, since the resource was newly created right now); 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.

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 3.5. In the second line, a new Petri net document is created. Note that this is not done using Java's `new` construct. Rather, the *factory* for the PNML core model is obtained by `PnmlcoremodelFactory.eINSTANCE`, which is then used for creating a new `PetriNeyDoc` object. It is part of

Listing 3.4: 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.
  }
}
```

Listing 3.5: Method `createPetriNetDoc(int number)`

```
1  public PetriNetDoc createPetriNetDoc(int number) {
     PetriNetDoc doc =
         PnmlcoremodelFactory.eINSTANCE.createPetriNetDoc();
     PetriNet net =
         PetriNetTypeExtensions.getInstance().createPetriNet(
6            "http://www.pnml.org/version-2009/grammar/ptnet");
     if (net == null) {
       return null;
     }
     PetriNetType type = net.getType();
11
     net.setId("n1");
     doc.getNet().add(net);

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

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

     Place semaphor = this.createPlace(
       type, "semaphor", "semaphor", 1, 380, 140);
     page.getObject().add(semaphor);
26
     for (int i=1; i<= number; i++) {
       page = createAgentPage(type, semaphor, i);
       pages.add(page);
     }
31
     return doc;
   }
```

the EMF philosophy that clients using the generated code should not know anything about the actual implementation of classes. And EMF strongly recommends to create new objects only via these factories. Note that also the new net is not directly created; it is created by using the ePNK's type registry `PetriNetTypeExtensions`, which creates the Petri net with the correct type from its URI. Note that the net will be `null`, if no net type is registered for the respective URI.

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 ids are different (if we chose to create the ids programmatically). 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 the `getNet()` method on which we call the `add()` method. There is no way to directly add a net to a document.

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 page. 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 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 is shown in Listing 3.6. This method 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 is straightforward and follows the exact principles of ISO/IEC 15909-2 (see [6]). Note that, if a reference pace is created, the place it refers to needs to be set by `setRef()`; therefore, we need to pass the the place `semaphor` to the `createRefPlace()` method as a parameter.

Once the implementation of the wizard class is finished, it must be made know to Eclipse: it must be plugged in via the "plugin.xml". But, we do not discuss this here since this is similar to plugging in a view (have a look into the "plugin.xml", if you are interested).

In the runtime workbench (or a version of the ePNK in which this plug-in is installed already), you could invoke this function as follows: Go to the

Listing 3.6: 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;
}
```

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 choose a folder[4] ("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 modified in such a way that existing files won't be overwritten accidentally.

3.3.3 Long-running functions: A model checker

In this section, we discuss the implementation of a model checker for P/T-Nets, which, actually, are interpreted as EN-Systems here. The model-checker is based on a simple library for symbolic model checking that was developed for teaching purposes: *Model Checking in Education* (*MCiE*)[5]. This MCiE library is deployed as part of the ePNK tutorials.

In 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 with an *action* on a Petri net that is open in the tree editor[6]. Model checking can actually be quite computation intensive and could take quite some time; therefore, we need to make sure that the actual model checking action does not block the graphical user interface of Eclipse while the model checker is running. Eclipse provides *jobs* for this purpose, which allow to run tasks (or jobs) in the background. The ePNK provides some convenience classes that make it a bit easier to set up and start jobs, which are running in the background – and provide a possibility to show a result in a dialog, once the job is finished.

The model checking functionality is implemented in the plug-in project `org.pnml.epnk.functions.modelchecking`. Like the other projects, you can import the source code of this project to your workspace via the Eclipse "Plug-ins" view and the "Import As" → "Source Project" menu. The actual model checker is implemented in the class `ModelcheckingJob` in package

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

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

[6] Note the extension points for pop-up menus and actions are deprecated since Eclipse 4.2. But, they still work – eventually, this will be adjusted by using Eclipse commands and handlers.

`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 this one first. It is shown in Listing 3.7. This class extends the `AbstractEPNKAction`, which is an

Listing 3.7: 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);
  }

}
```

ePNK convenience class that makes it easy to add a new action, which initiates a job. The class `AbstractEPNKAction` overrides two methods: `isEnabled()` and `createJob()`. The method `isEnabled()` checks whether the action is applicable for the selected Petri net. In our example, it checks whether the Petri net has a type and whether this type is `PTNet`. The other method `createJob()` creates the actual job, which is an instance of class

Listing 3.8: 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>
```

`ModelcheckingJob` with a Petri net and a defaultInput (a default formula for the user dialog in our case). 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 3.8 shows the part of the "plugin.xml" file that defines this extension (with some ellipses).

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 implement (override) the following methods of `AbstractEPNKJob`: `prepare()`, `getInput()`, `run()`, `showResult()`, and `canceling()`. Below we explain the implementation of the constructor and the methods:

Constructor: Sets up all the data structures needed during the job; typically, this will be storing the default input. In our model checker ex-

Listing 3.9: 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;
}
```

ample, 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 3.9.

`prepare()`: This method is handling the user dialogs before the actual job starts. In our case, it asks the user for some CTL-formulas; it 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 3.10. Since this is standard Eclipse programming, we do not go into the details of this part here. 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 11/12 and line 33/34).

In our model checking 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 3.11 shows the part of the `prepare()` method converting the initial marking into a state formula. The basic idea is that, in this formula, a variable corresponding to the place occurs exactly once. It occurs negated, if the place is not marked and it occurs without nega-

Listing 3.10: 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:" + LF +
        e.toString() + LF +
        "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);
```

Listing 3.11: Building the formula for the initial marking (in `prepare()`)

```
FlatAccess flat = FlatAccess.getFlatAccess(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)));
    }
  }
}
```

tion, if the place is marked (with at least one token[7]). All these negated and un-negated variables are connected by boolean and-operations as formulas 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`; its static method `getFlatAccess()` called with a parameter of a Petri net of any type creates a *flat access object* for that net (called `flat` in our case). This flat access object can be used to obtain all places of the net, independently of the pages they occur on. Likewise, the flat access object 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 the code that converts transitions into formulas, which however is not discussed here.

Note that creating a flat access object is quite computation intensive; therefore, the static method `getFlatAccess()` for creating a flat access object[8] will create only one flat access object for each net. But, the flat access object becomes invalid, when the underlying Petri net changes. The flat access object can, therefore, also be used to notify an application that the underlying net has changed. But, we do not discuss the details here.

The last part of the prepare method, is converting the formulas into ROBDD-representation and creating a transition system out of these formulas. This is shown in Listing 3.12. 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.

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

[8] Note that in earlier versions of the ePNK, the flat access object was created by a constructor. This constructor is deprecated now, and should no longer be used; but, it is still there, so that older code should still be working. It is strongly recommended, to replace the use of the constructor, though.

Listing 3.12: Finishing the `prepare()` method

```
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;
```

Listing 3.13: The `run()` method

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

Note that all computations in the `prepare()` method should run fast. Computations that are time-consuming should be implemented in the `run()` method, which will be run in a separate thread in the background.

`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 – and typically contains the computation intensive parts. In our case, this is the actual model checking task. The implementation of the method is actually quite simple (most of the programming work lies in the preparation and the MCiE framework). It is shown in Listing 3.13. Still, it is the most computation intensive part, which is why we are using the job to run it in the background. Note that, at the end of this method, we also prepare the `result` already in a String that will be shown to the user. But, there must not be any user dialog in the `run()` method itself, 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 3.14 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 3.14: Code for showing the final result

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

Listing 3.15: Code for aborting the model checking job

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

`canceling()` This method is a call-back mechanism that allows Eclipse – typically triggered by the end user – to abort a job that is running in the background. In the case of computation intensive jobs, 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()` on the underlying transition system on which the model checking is done. Our implementation of the `canceling()` method invokes this `abort()` method to actually terminate the model checking – possibly with some delay. This is shown in Listing 3.15, where `transitionsystem` is the one that was constructed before in the `prepare` method and on which the model checking is done. This will actually cause the model checker – the thread in which the computation is running – to throw some exception at some point of its computation in the `isValid()` method; this will stop the complete thread, since the exception is not caught.

Together, the classes `ModelcheckingAction` and `ModelcheckingJob`, plugged in via the "plugin.xml" implement a simple, but complete model checker. If the model checking project was not already part of the ePNK, you would start the runtime workbench, and would have the model checker

available there. Section 2.6.1 of the Users' Guide had explained already how to use this model checker.

3.3.4 Overview of the ePNK API

The previous sections have given an idea of how the ePNK and its API can be used to access and modify Petri nets for implementing functions and, as discussed later, applications on Petri nets. They also showed how to plug in these functions to the ePNK – or actually to Eclipse. But, these examples just scratch the surface. In this section, we give an overview of where to find and look up things in the API of the ePNK and how to use this API in the context of Eclipse and EMF. Some of these things are actually not specific to the ePNK, but specific to Eclipse and EMF, and could be read up on in the many Eclipse publications (e. g. [2, 4, 5]). Anyway, we briefly mention or point to some of the relevant concepts here, in order to avoid some unpleasant surprises.

3.3.4.1 Eclipse and EMF

We start with giving an overview of the code that is generated[9] from Ecore models, which was also briefly discussed in Sect. 3.3.1 already. All models used in the ePNK are *Ecore models*, which are an implementation of the *Meta Object Facility* (*MOF*) [23]. For the purpose of this manual[10], an Ecore model can considered to be a simplified version of a UML class diagram. Note that, in Ecore models, the concept that represents a UML association is called *reference* – and in the case of a bi-directional association, a pair of two *opposite* references.

The *Eclipse Modeling Framework* (*EMF*) [2] allows us to generate Java code from these models, which provides *getter* and *setter methods* for all the attributes and references. And there will be a lot of code behind the scenes for loading and saving models, and for notifying some observers when changes are made. Actually, in the generated code, each class of an Ecore model is represented by a Java interface and a Java class implementing this interface; the interfaces and implementations reside in two different Java packages – where typically the package name with the implementing classes ends with a segment called "impl" and also the name of the implementing

[9] EMF provides many features to configure and change the way the code is generated from a model. Here, we discuss only the standard settings, which – with some few exceptions – are used for all models of the ePNK.

[10] We do not bother to go into the details of the MOF-levels and into the motivation behind MOF. See [15] for an brief introduction and overview.

Java class will end with "Impl". Normally, developers that want to access model elements, would use the interfaces only. For attributes and references with multiplicities 1 or 1..0, the generated API and the use of the generated setter and getter methods is straightforward[11]. In case of multiplicity *, you will find that there are no setter methods for the respective attribute or reference at all; there will be a getter method, which returns a collection. In order to add or remove an element to or from the attribute or reference, you would obtain this list by the getter method, and then add or remove something from the collection by the respective methods of collections. Note that this collection is attached to the object, and it is crucial that you do not use it for other purposes.

As explained above, the package with the Java classes that implement the Java interface of the model should typically not be used directly by other developers. This also applies to the constructor (which normally is protected). If a developer wants to create an instance of some class, this should be done via a *factory* for the model, which can be found in the same Java package as the generated Java interfaces of the model. This factory class is typically called `XXXFactory`, where `XXX` is the name of the package; the singleton instance of this class can be accessed by a static attribute `eINSTANCE`. For each class of the model, this Factory provides a method for creating a new instance of the respective class.

Note that all[12] Java classes that are generated as implementations for classes from the Ecore model inherit from the class `EObject` of the EMF Framework. The class `EObject` provides a lot of functionality behind the scenes and also some convenience methods. For example, it allows another object to register with it as a listener, so that the other object is notified about any changes of its attributes and references, and even some other events. But, we do not go into these details here. One of the convenience methods is to obtain an iterator of all its directly and indirectly contained elements (indicated by compositions in the Ecore model): `eAllContents()`. All the methods of the `EObject` start with the letter "e". We cannot discuss all of them here; but, for example, there are methods for reflectively finding out which model class this object represents `eClass()`; there is a method `eContainer()` to obtain the object in which this object is directly contained; there are methods to find out which features this object has, and to change them.

[11] There is a minor, but sometimes confusing twist when an attribute is of type boolean: in that case, the getter method actually starts with "is" instead of "get".

[12] Remember that we discuss the standard configuration of EMF only.

Note that `EObject` has a method `eResource()`, which returns the resource (file) in which the object is contained – if it is associated with a file already. Resources are important, when loading and saving models to a file, and when they are loaded and edited in an editor. Actually, a resource is typically contained in a *resource set*, which is responsible for maintaining different resources that refer to each other – and for loading and saving them together so that the links between them remain consistent. For a resource, the resource set it is contained in can be obtained by method `getResourceSet()`. In turn, resources can and should be created from a resource set, which will make sure that the correct type of resource is used for the respective file type. We have seen two examples of that already: in the file overview (Sect. 3.3.1), the resource set and resource was used to open a selected PNML file; in the "multi-agent mutex wizard" (Sect. 3.3.2), the resource set was used to create a new PNML file. Note that, once you have a resource, its contents can be obtained by the method `getContents()`, which returns a list of `EObjects`, which contains the top-level element of that resource; adding and removing elements to or from this list will add and remove these elements to the top-level of the resource. The `save()` method of the resource can be used to save the current contents of the resource to the file.

Note that the only example where we actually create or change a Petri net model programmatically via the API is the "multi-agent mutex wizard" of Sect. 3.3.2. In the other examples, we access and inspect the contents of a Petri net document only. For making the changes and additions we made, we used the getter and setter methods of the API that was generated from the PNML core model. This, however, was possible only because the resource that we were changing was not under the control of an editor. If a resource is under the control of an editor, the resource and actually the complete resource set would also be under the control of a so-called *editing domain*. In that case, we cannot make changes on the resource with the getter and setter methods of the API directly anymore. Depending on which kind of editing domain it is, changes made via the API might result in exceptions. The reason for this is that changes "on the side" by some other program would ruin the editor's undo and redo mechanism. If a function should make changes to a model that is under the control of an editing domain, these changes need to be encapsulated into commands of the Eclipse *command framework*, which however is beyond the scope of this manual (see Sect. 3.3 of [2] for an overview of these concepts).

Eclipse provides many different ways to plug in extensions to Eclipse itself and to the ePNK. In the examples from Sect. 3.3.1–3.3.3, we used

views, *wizards*, and *pop-up menus* for that purpose, and we used *jobs* for running long-running computations in the background. Eclipse, provides many more possibilities, which are beyond the scope of this manual. You will find more information on that in [4]: Chapter 6 discusses commands[13], actions and handlers; Chapter 7 discusses views and Sect. 21.8 gives a brief overview of Eclipse jobs.

For pop-up actions and handlers, the respective extension points of Eclipse allow us to provide information to which elements the respective actions and commands should apply, and when the respective actions should be visible in pop-up menus, tool-bars etc. Only when the respective element is selected these entries will be shown. This is straightforward when elements are selected in the tree editor – then the respective Java class can be used. In the graphical ePNK editor, this is slightly more tricky, since Eclipse does not always "see" the underlying model elements; Eclipse "sees" only the controllers, which are called *edit parts*. If you want to attach commands and actions to the graphical editor, the actions and handlers need to be registered for these edit parts – depending on which mechanisms you are using. Since the action is then called with an edit part, the action needs a way to obtain the underlying model element. And this might be a bit tricky, for people new to the EMF and GMF framework. Therefore, Listing 3.16 shows how to obtain a `Page` object from its corresponding edit part[14]. The code for other types of net elements is similar, where `Page` would need to replaced with the respective other class. Note that the method `getModel()` is actually not returning the model element behind the edit part. It returns a view of the diagram; only the `getElement()` method of this view returns the underlying model element.

3.3.4.2 ePNK models

In order to implement functions for the ePNK, you would make use of the different packages, classes, and their methods of the ePNK (in short the API of the ePNK). Since there is a standard mapping between the Ecore models and the generated API (see Sect. 3.3.4.1), we do not discuss the API explicitly; we give an overview of the models underlying the ePNK,

[13] Note that this notion of command should not be confused with the notion of command of the EMF command framework!

[14] This code is a snippet from the action that opens a graphical editor on a page, which can be found in the project `org.pnml.tools.epnk.gmf.integration` in the class `InitiateGMFEditorOnPage` of package `org.pnml.tools.epnk.gmf.integration.actions.popup`.

Listing 3.16: Accessing the model element underlying an edit part

```
page = null;
if (selection instanceof IStructuredSelection) {
  IStructuredSelection structuredSelection =
    (IStructuredSelection) selection;
  if (structuredSelection.size() == 1) {
    Object selected = structuredSelection.getFirstElement();
    if (selected instanceof Page) {
      page = (Page) selected;
    } else if ( selected instanceof EditPart ) {
      EditPart part = (EditPart) selected;
      Object model = part.getModel();
      if (model != null && model instanceof View) {
        EObject object = ((View) model).getElement();
        if (object != null && object instanceof Page) {
          page = (Page) object;
} } } } }
```

which serve as a kind of map. The standard mapping together with the auto-completion mechanism of the Eclipse IDE, should make it possible to use the API based on these models.

The ePNK is based on (and generated from) many different models, most of which reside in the plug-in project `org.pnml.tools.epnk`[15] in the "model" folder. Undoubtedly, the most important model is the *PNML Core model*; it contains all the constructs common to all Petri nets (cf. Fig. 1.1 and Fig. 3.1). The PNML core model is actually split up into three diagrams[16]: the diagram `PNMLCoreModel.ecorediag` covers the main concepts of PNML, the diagram `PNMLCoreModelGraphics.ecorediag` covers the graphical features of PNML, and `PNMLCoreModelProxies.ecorediag` covers some features that are volatile (which means that they are not saved to a file) and are responsible for maintaining the relation between the GMF diagram and the PNML information. The corresponding Java package with the interface and the factory for this package is `org.pnml.tools.epnk.pnmlcoremodel`.

Since we had discussed the main idea of the PNML core model al-

[15] Remember that you can make the source code and the models available via the "Import As" → "Source Project" from the Eclipse "Plug-ins view" (see Sect. 3.1.1).

[16] As mentioned already, these diagrams are legacy from an outdated version of Ecore Tools. But, if you install Eclipse as discussed in Chap.5, there will be a legacy editor for these diagrams, so that you still can inspect them.

ready in Sect. 1.2.1, we do not discuss it here any further. Concerning the volatile features of `PNMLCoreModelProxies.ecorediag` and the classes `PageLabelProxy` and `LabelProxy`, we actually recommend not to use them anywhere in your functions and applications.

The model `PNMLDataTypes` defines some of the data types used in the PNML core model (note that this replaces the respective XMLDataTypes that are used in ISO/IEC 15909-2).

The model `PNMLStructuredPNTypeModel` provides some general infrastructure for defining more complex Petri net type definitions with labels that require some parsing and linking, which will be discussed in Sect. 3.5.3.

The other two models `PNMLPageDiagramInfo` models the GMF diagram information for the graphical editor of the ePNK, which is stored as tool specific information of the PNML model. And the model `Serialisation` represents some auxiliary information when loading some models. Both of these models, and the API generated from them are not supposed to be used for ePNK extensions. In particular, messing around with the diagram information might render graphical information inconsistent – and the graphical editor of the ePNK might not be able to start up again, when this is changed manually.

3.3.4.3 ePNK Petri net types and their use

Some of the models that come with the ePNK provide the definition of the two Petri net types of ISO/IEC 15909-2. The model for the Petri net type definition of P/T-Systems resides in the model folder of project `org.pnml.tools.epnk.pntypes`: `PTnet.ecore`. The models for the Petri net type definition of HLPNGs resides in the model folders of projects `org.pnml.tools.epnk.pntypes.hlpngs.datatypes` and `org.pnml.tools.epnk.pntypes.hlpng.pntd`.

Both Petri net type definitions are discussed in more detail in Sect. 3.5.

Here we point out one important aspect of using these Petri net types when adding new Petri net elements like places, transitions, and arcs to the net. Since every Petri net type can define its own kind of extensions of places, transitions and arcs, and actually also of pages, and reference nodes, it is important, that only places of that kind are used in a net of the respective kind. The tree editor as well as the graphical editor of the ePNK guarantee that always the correct type of element is created, which fits the Petri net type. The API, however, would allow to add other kinds of elements, which ultimately might result in problems when serializing and loading the net again. In order to make it easier to create the correct type of element,

the class that defines a Petri net type serves as a factory for creating the respective elements: The interface `PetriNetType` which all Petri net types implement has the methods `createArc()`, `createPage()`, `createPlace()`, `createTransition()`, `createRefPlace()`, and `createRefTransition()`. And it is strongly recommended to use these methods for creating the respective elements (we have seen that in the example of Sect. 3.3.2 already).

Actually the interface `PetriNetType` even serves as a factory for creating the Petri net type itself and for creating a Petri net of the respective type: `createPetriNet(String)`, `createPetriNetType(String)`, and `createPetriNetType()`, where the parameter of type `String` would be the unique URI identifying the type, which is discussed in Sect. 3.5 in more detail.

3.3.4.4 ePNK convenience classes

The main purpose of the PNML core model was to define an interchange format for Petri nets. The concepts and their relation captured in the PNML core model were driven by this purpose. For actually accessing, modifying, and updating the net, the model and the API generated from it are sometimes a bit clumsy and require some extra steps in programming. In order to make up for that, the ePNK provides some convenience classes that should make some programming a bit easier. Some of the convenience classes can be found in the Java package `org.pnml.tools.epnk.helpers` in the plug-in project `org.pnml.tools.epnk`.

The first important class is `FlatAccess`. Instances of `FlatAccess` can be created by calling the static method `getFlatAccess()` with the respective Petri net as a parameter[17]. Once an instance is created, it provides methods to directly get a list of all the places and transitions of that net. And for each node, it gives the set of all in-coming and out-going arcs. And there are two methods that, for a place or a transition, return the list of all reference places resp. reference transitions that refer to that place. And there are methods for the other direction: for a given node (which might be a reference node), the method `resolve()` computes which node it actually refers to (which will be a place or a transition). This actually indicates the main purpose of the convenience class `FlatAccess`. Even though the Petri net model contains

[17] In earlier versions of the ePNK, an instance of `FlatAccess` could be created via a constructor; this constructor is deprecated now and should no longer be used – mostly for efficiency reasons. The ePNK maintains a single valid `FlatAccess` object for each net and takes care of maintaining and invalidating them, when they are created under the control of the ePNK via the `getFlatAccess()` method.

pages and places and transitions distributed among them, with `FlatAccess` it appears to be a flat net. This way, functions and applications that are interested in the Petri net only, can ignore the page structure.

Note that you can also register listeners (in EMF terminology an `Adapter`) with `FlatAccess`, which then will be notified when the `FlatAccess` object becomes invalid. This happens, when the underlying net is changed semantically while an application is running; note that purely graphical changes will not invalidate the `FlatAccess`, since the net does not change semantically. We will see in Sect. 3.4 how this can be used to shut down an application, when the underlying net is changed semantically by the end user.

An other convenience class is `NetFunctions`. It provides several static methods: e. g. there is a method that, for a given object, returns the Petri net to which this object belongs (or `null`, if it does not belong to any Petri net); there is a method that returns the Petri net type of the net an object belongs to. And there are methods that return all pages of a net or all the net's objects.

The convenience classes `AbstractEPNKAction` and `AbstractEPNKJob`, make it easier to start long-running computations on some Petri net. These two classes can be found in the Java package `org.pnml.tools.epnk.actions.framework.jobs` in the plug-in project `org.pnml.tools.epnk.actions`. The use of these two classes is discussed in Sect. 3.3.3.

3.4 Implementing applications

In this section, we discuss the implementation of ePNK *applications*. The simulators for P/T-nets and high-level nets, which we had discussed in Sect. 2.6.2 and 2.6.3, are typical examples of such applications. In contrast to functions, applications – once started – stay in the background, ready to interact with the end user and to show results to the end user. In addition, applications can visualize results by overlays on top of the Petri net in the graphical editor, and they can interact with the end user via these overlays, as we have seen in the simulators.

We discuss how to implement an ePNK application by the help of the simulator application for P/T-nets, which we had discussed in Sect. 2.6.2. You will find the source code for this simulator in project `org.pnml.tools.epnk.tutorials.applications.pt-net-simulator`, which you can import to your Eclipse workspace as discussed before.

Here, we discuss the idea, the major concepts and steps for developing an ePNK application by using this example. In Chapter 4, you will find another

example, which goes through all the technical details of implementing an ePNK application.

3.4.1 Overview

Implementing a new *ePNK application* consists of three steps: First, the runtime information of the application needs to be defined. This is done by defining *annotations*, which is discussed in Sect. 3.4.2. Second, some *handlers* need to be implemented; these handlers define how the annotations of the runtime information are presented to the end user, and which actions should be taken, when the end user interacts with an annotation in the graphical editor. The handlers are discussed in Sect. 3.4.3. Third, the actual application, needs to be implemented, which combines the annotations and handlers; and the application needs to be plugged in to the ePNK. This is discussed in Sect. 3.4.4.

When implementing an application, these three steps are not necessarily done sequentially. In most of our applications, we have chosen to implement the core functionality of the application in the so-called application class, and the handlers will mostly delegate the work to some methods of the application. But, for conceptual clarity, we explain the steps in the order as introduced above.

3.4.2 Annotations

The first step of realizing an ePNK application is to define the *runtime information* of the application. Of course, it depends on the specific application what constitutes its *runtime information*. For our simulator, this runtime information is the current marking of the net along with the sequence of all markings up to the current one. Technically, the runtime information of an ePNK application is defined by *annotations*, which are associated with the net itself and with the net's objects.

Following the idea of model-based software engineering, the annotations of an application are defined by a model, which then is used to automatically generate some code from it. Figure 3.3 shows the class diagram defining the annotations for the P/T-net simulator. This diagram consists of three parts: The class Object on the top-left comes from the *PNML core model* as discussed in Sect. 3.2, which represents the different kinds of Petri net objects. These are the objects that are supposed to be annotated. The other classes at the top (graphically represented in orange), represent the general annotations that are built into the ePNK. These are extended by

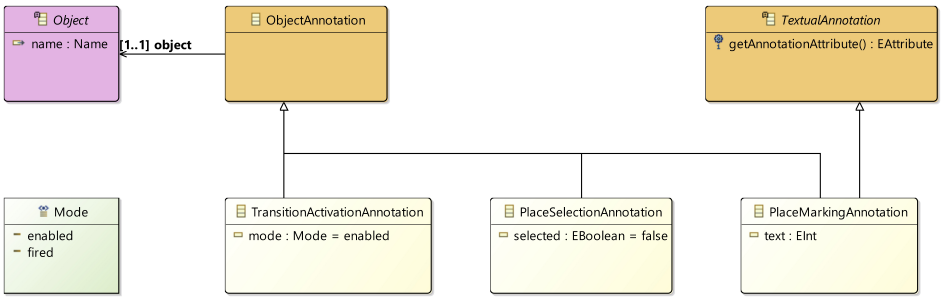

Figure 3.3: Annotations for P/T-nets simulator

the annotations of a specific application, which, in our case, are the three classes at the bottom.

Let us have a closer look at these concepts: The concept Object represents objects of a Petri net: places, transitions, and arcs; but also pages, reference places and reference transitions. As said already, these represent the objects the annotations refer to. The orange classes at the top are ePNK's built-in concepts of annotations. An ObjectAnnotation annotates exactly one object, which is represented by the reference object. Note that a Petri net and its objects do not know anything about their annotations at all, since applications should be detached from the net they are running on.

The three classes at the bottom of Fig. 3.3 define the annotations for the simulator: TransitionActivationAnnotation, PlaceSelectionAnnotation and PlaceMarkingAnnotation, which we have seen representations of in Fig. 2.13 already. The ePNK annotation model has an abstract class TextualAnnotation. An ePNK application, by default, presents annotations inheriting from TextualAnnotation as textual labels at the top-right of the object in the graphical editor, showing the value of its text or value attribute. In our YAWL simulator, PlaceMarkingAnnotation is an example of such a textual annotation. All other annotations are, by default, shown as red overlays of the respective object. But, we will see later that an application can change this. The annotation TransitionActivationAnnotation is used for indicating which transitions are enabled in the current marking. This has actually an attribute mode of the enumeration type Mode, which is also defined in the model. The Mode can have two different values: enabled and fired. This is used to distinguish transitions that are enabled (to be selected by the end user) from transitions that are going to fire in the firing sequence. When navigating back and forth in the firing sequence, the transition firing in the next step will be represented in a different colour: The transition that will

fire according to the current firing history is shown with a red overlay, as will be discussed later. And the places affected by this transition will also have a special annotation, which we call PlaceSelectionAnnotation[18]. These places will be highlighted by a red overlay.

Altogether, the classes from Fig. 3.3 allow a running simulator to store its state. We call this the *runtime information* of the simulator. Note, that this information can be much more than what is currently shown to the end user: in our simulator, only the current marking is shown visually; but the user can navigate back and forth in the complete firing sequence.

Before you can use the annotations of the model, the code for this model needs to be generated as usual for EMF. But, we do not discuss the details here.

3.4.3 Handlers

Once the annotations of an application are defined and the code for them is generated, two major things need to be defined for an application: how should an annotation of an application be visualized on top of the respective object in the graphical editor; and what should happen when the end user interacts with the representation of this overlay (ie. what happens when the end user clicks or double clicks on the respective overlay). This is defined by the handlers of an application.

The *presentation handlers* will take care of how the annotations of the current annotation are shown to the end user. Actually, the P/T-net simulator has only one explicitly presentation handler; in addition, each application has a default presentation handler, which is registered with all ePNK applications by default. This way, all annotations for which there are no dedicated presentation handlers in the application, the default presentation handler will kick in.

Listing 3.17 shows the explicitly defined presentation handler of P/T-net simulator. We will use it to discuss the main concepts of presentation handlers. First of, each presentation handler must implement the interface IPresentationHandler (lines 1–2), which comes with the ePNK. This means it needs to implement method handle() (line 8–9), which takes two parameters: an annotation and and editPart. The annotation is the annotation for which the handler is asked to return a figure (technically an IFigure from Eclipse's Draw2D framework). The editPart is the graphical edit part from Eclipse's GEF framework, which is representing the annotated object of the Petri net

[18]The attribute selected of this class is used only to demonstrate some additional features of the presentation handler; but we do not discuss this in this report.

in the graphical editor. This will be typically be used for making sure that the representation of the annotation is aligned and moved around with the representation of the object in the graphical editor, as discussed below.

let us discuss some details of the implementation of this handle() method shown in Listing 3.17. Basically, this method returns a figure for an annotation in two cases: if the annotation is a TransitionActivationAnnotation (lines 10–25) and if the annotation is a PlaceSelectionAnnotation (lines 26–39); otherwise it returns null, which means this presentation handler does not know how the annotation should be handled, and another presentation handler might kick in.

Let us discuss the case of a TransitionActivationAnnotation (lines 10–25) first. From the edit part of the annotated object, it will checked of the underlying object actually is a TransitionNode (lines 13–17); if this is the case a new RectangleOverlay figure is created, which actually comes from the ePNK; and the constructor of this RectangleOverlay takes the graphical edit part as a parameter, so that the overlay adjusts its size and position to the underlying transition. The RectangleOverlay, actually does all this for us, so that we do not need to bother with the details. By default, a RectangleOverlay will be red and translucent (so that the underlying transition is still visible through the overlay). If the annotation is in mode ENABLED, we switch the colour to BLUE (by default it is still translucent). This overlay is then returned (line 25) as the figure representing this annotation.

The case of a PlaceSelectionAnnotation (lines 26–39) is similar. Here we check whether the object underlying the graphical edit part is a PlaceNode and, if this is the case, create a new EllipseOverlay, which also comes with the ePNK. By default this will be shown translucent and red. But, if the annotation says that the place is selected, its colour will be set to light grey. Then the overlay is returned.

In all the other cases, this presentation handler returns null, meaning it has no representation for this annotation in the given context of the provided edit part.

This is all, a presentation handler needs to do. Of course, you could program your own more sophisticated overlays; but the implementations provided by the ePNK EllipseOverlay, RectangleOverlay, PolylineOverlay (for arcs), and LabelOverlay for textual labels showing values, should cover the most relevant representations for annotations. In Sect. 3.4.4, we will see how an application can be associated this and more presentation handlers.

- action handlers

Listing 3.18 and 3.19...

Listing 3.17: Presentation handler of Simulator

```
public class PTSimulationPresentationHandler implements
                    IPresentationHandler {

  private static final BLUE = ColorConstants.blue;
  private static final LIGHTGREY = ColorConstants.lightGray;

  @Override
  public IFigure handle(ObjectAnnotation annotation,
      AbstractGraphicalEditPart editPart) {
    if (annotation instanceof TransitionActivationAnnotation) {
      TransitionActivationAnnotation activationAnnotation =
          (TransitionActivationAnnotation) annotation;
      if (editPart instanceof GraphicalEditPart) {
        GraphicalEditPart gep = (GraphicalEditPart) editPart;
        java.lang.Object modelObject =
            gep.resolveSemanticElement();
        if (modelObject instanceof TransitionNode) {
          RectangleOverlay overlay =
              new RectangleOverlay(gep);
          if (activationAnnotation.getMode().
              equals(Mode.ENABLED)) {
            overlay.setForegroundColor(BLUE);
            overlay.setBackgroundColor(BLUE);
          }
          return overlay;  }  }
    } else if (annotation instanceof PlaceSelectionAnnotation) {
      PlaceSelectionAnnotation placeAnnotation =
          (PlaceSelectionAnnotation) annotation;
      if (editPart instanceof GraphicalEditPart) {
        GraphicalEditPart gep = (GraphicalEditPart) editPart;
        java.lang.Object modelObject =
            gep.resolveSemanticElement();
        if (modelObject instanceof PlaceNode) {
          EllipseOverlay overlay = new EllipseOverlay(gep);
          if (!placeAnnotation.isSelected()) {
            overlay.setForegroundColor(LIGHTGREY);
            overlay.setBackgroundColor(LIGHTGREY);
          }
          return overlay; } } }
    return null;
} }
```

Listing 3.18: Action handler of Simulator (part1)

```
@Override
public boolean mouseDoubleClicked(
    MouseEvent arg0, ObjectAnnotation annotation) {
  NetAnnotations netAnnotations = app.getNetAnnotations();
  NetAnnotation current = netAnnotations.getCurrent();
  PetriNet net = app.getPetrinet();

  if (current.getObjectAnnotations().contains(annotation)) {
    Object object = annotation.getObject();
    if (annotation instanceof TransitionActivationAnnotation &&
        object instanceof TransitionNode) {
      FlatAccess fn = FlatAccess.getFlatAccess(net);
      Transition t = fn.resolve((TransitionNode) object);

      Map<Place,Integer> marking1 = app.computeMarking();
      if (app.enabled(marking1, t)) {
        Map<Place,Integer> marking2 =
            app.fireTransition(marking1, t);
        NetAnnotation netAnnotation =
            app.computeAnnotation(marking2);
        netAnnotation.setNet(net);

        TransitionActivationAnnotation ta =
            ((TransitionActivationAnnotation) annotation);
        List<ObjectAnnotation> clearPlaceAnnotations =
            new ArrayList<ObjectAnnotation>();
        for (ObjectAnnotation oa:
            current.getObjectAnnotations()) {
          if (oa instanceof TransitionActivationAnnotation &&
              oa != ta ) {
            ((TransitionActivationAnnotation) oa).
              setMode(Mode.ENABLED);
          } else if (oa instanceof PlaceSelectionAnnotation) {
            clearPlaceAnnotations.add(oa);
        } }
        current.getObjectAnnotations().
          removeAll(clearPlaceAnnotations);
        transitionAnnotation.setMode(Mode.FIRED);
```

Listing 3.19: Action handler of Simulator (part2)

```
          for (Arc arc: fn.getOut(transition)) {
            Object o2 = arc.getTarget();
            if (o2 instanceof PlaceNode) {
              Place target = fn.resolve((PlaceNode) o2);
              if (target != null) {
                PlaceSelectionAnnotation placeAnnotation =
                    PtnetsimulatorFactory.eINSTANCE.
                      createPlaceSelectionAnnotation();
                placeAnnotation.setObject(target);
                placeAnnotation.setSelected(true);
                current.getObjectAnnotations().add(placeAnnotation);
                for (PlaceNode ref: fn.getRefPlaces(target)) {
                  placeAnnotation = PtnetsimulatorFactory.eINSTANCE.
                      createPlaceSelectionAnnotation();
                  placeAnnotation.setObject(ref);
                  placeAnnotation.setSelected(true);
                  current.getObjectAnnotations().
                    add(placeAnnotation);
          } } } }
          app.deleteNetAnnotationAfterCurrent();
          app.addNetAnnotationAsCurrent(netAnnotation);
          return true;
      } } }
      return false;
    }
```

3.4.4 Application

- registering handlers
  - plugging in an application
  - nextAnnotation()
  - shutDown()
  - is savable()

The implementation of the transition context application can be found in the project `org.pnml.tools.epnk.tutorials.applications`. Listing 3.20 shows the outline of the class implementing the application, where a part of the computation of the context is still missing – as indicated by ellipses. The missing part can be found in List. 3.21.

We start with the discussion of the overall structure, which is shown in Fig. 3.20. Line 1 shows that the `CalculateTransitionContext` application extends the `Application`, which is an ePNK convenience class making it easy to implement applications. It would be enough if the application implemented `IApplication`; this would, however, require much more programming. Lines 3–5 show the constructor, which does not have any exiting behaviour in its own right; since the constructor of the class `Application` expects a Petri net, this parameter is just passed on.

The actual computation of the transition context is done in the method `initializeContents()`, where for each transition, all the in-coming and out-going arcs as well as the attached places are computed, and an *object annotation* is created for each of them. The actual computation is not yet shown – the code in lines 14–16 shows only the creating of a new *net annotation* to which the object annotations will be added later. Note that, for each net annotation, a corresponding net must be set.

After all the object annotations have been computed and added to the net annotation, the net annotation is added to the list of all the application's net annotations (which is obtained from the application by method `getAnnotation()` as shown in line 9). In the end (lines 23–25), the net annotations current annotation is set to the first one of the computed list. This will be the elements that are initially high-lighted: the context of the first transition.

Listing 3.21 shows the details of how the context is computed for each transition, and how the corresponding object annotations are created and added to the net annotation (note that the listing shows the complete for-loop again – also the part that was shown in List. 3.20 already). As said before, for each transition, a new net annotation is created (by using the respective factory of the annotation package) and the associated Petri net

Listing 3.20: Transition context application: Outline

```
public class CalculateTransitionContext extends Application {

  public CalculateTransitionContext(PetriNet petrinet) {
    super(petrinet);
  }

  public void initializeContents() {

    NetAnnotations netAnnotations = this.getNetAnnotations();
    PetriNet petrinet = this.getPetrinet();

    for (Transition transition: (new FlatAccess(petrinet)).
        getTransitions()) {
      NetAnnotation netAnnotation = NetannotationsFactory.
          eINSTANCE.createNetAnnotation();
      netAnnotation.setNet(petrinet);

      ...

      netAnnotations.getNetAnnotations().add(netAnnotation);
    }

    if (netAnnotations.getNetAnnotations().size() > 0) {
      netAnnotations.setCurrent(
          netAnnotations.getNetAnnotations().get(0));
    }
  }
}
```

Listing 3.21: Transition context application: Computing the context

```
for (Transition transition: (new FlatAccess(petrinet)).
    getTransitions()) {
  NetAnnotation netAnnotation = NetannotationsFactory.
      eINSTANCE.createNetAnnotation();
  netAnnotation.setNet(petrinet);
  ObjectAnnotation objectAnnotation = NetannotationsFactory.
      eINSTANCE.createObjectAnnotation();
  objectAnnotation.setObject(transition);
  netAnnotation.getObjectAnnotations().add(objectAnnotation);

  for (Arc arc:transition.getIn()) {
    objectAnnotation = NetannotationsFactory.
        eINSTANCE.createObjectAnnotation();
    objectAnnotation.setObject(arc);
    netAnnotation.getObjectAnnotations().add(objectAnnotation);

    objectAnnotation = NetannotationsFactory.
        eINSTANCE.createObjectAnnotation();
    objectAnnotation.setObject(arc.getSource());
    netAnnotation.getObjectAnnotations().add(objectAnnotation);
  }

  for (Arc arc:transition.getOut()) {
    objectAnnotation = NetannotationsFactory.
        eINSTANCE.createObjectAnnotation();
    objectAnnotation.setObject(arc);
    netAnnotation.getObjectAnnotations().add(objectAnnotation);

    objectAnnotation = NetannotationsFactory.
        eINSTANCE.createObjectAnnotation();
    objectAnnotation.setObject(arc.getTarget());
    netAnnotation.getObjectAnnotations().add(objectAnnotation);
  }

  netAnnotations.getNetAnnotations().add(netAnnotation);
}
```

is set. Then an object annotation is created (by using the factory again) and added to the net annotation; for the object annotation, we need to set the object that should be annotated; in this case (line 8), it is the transition. Then, by iterating over the in-coming arcs (line 11–21), object annotations for the arcs and their source places are created and added to the net annotation. For each of these object annotations, we need to set the reference to the net object that should be annotated by it. Likewise the object annotations for the out-going arcs and the attached places are created (line 23–33). In the end, all object annotations that have been collected for the context of the transition in a single net annotation are added to the list of net annotations of this application (line 35).

This is all that needs to be done so that the transition contexts are shown as discussed in Sect. 2.6.2, once the application is started. The standard actions for the application then allow the end user to go back and forth in the list of all net annotations.

Of course, there must be a possibility for the user to start the application. In our case, this is done by a pop-up menu, which is installed on a Petri net object in the tree editor of the ePNK. This is done by standard Eclipse mechanisms and not discussed here (have a look at the class `StartApplication` and the "plugin.xml" file in project `org.pnml.tools.epnk.tutorials.applications`, if you are interested in details). It is planned for the future, to realise a mechanism to plug in ePNK applications so that they automatically show up in the ePNK menus (or a separate view).

Of course, there are other kinds of applications, where not all the annotations can be calculated in the initialisation. In that case, some of the methods of the convenience class `Application` can be overridden in order to accommodate for that. Then, it is also possible to install more or other actions than standard forward and backward buttons. In particular, overriding the `nextAnnotation()` method could be used to calculate the next annotations on demand. Note that, if an application allocates resources that need to be freed, when the application is closed, this should be done by overriding the method `shutDown()`.

Right now, all net annotations consist of a set of object annotations. Graphically, such a net annotation is always shown by a red overlay of the respective elements in the graphical editor (provided the respective page is open in a graphical editor). By some programming this behaviour can actually be changed (the simulator for high-level Petri nets of Sect. 2.6.3 is an example). But, we intend to equip applications with a *presentation description*, by which an application can define how specific annotations should be shown to the end user; e. g. by using different colours or different

shapes or textual annotations on top of the graphical representation of the Petri net. Moreover, there will be an *interaction description* for applications, that will allow us to define how the end user can interact with some of the annotations by clicking on them; and which action are triggered by these user interactions.

3.5 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. 3.5.1, we start with a simple version, for which we, basically, need to provide an Ecore 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.

In order to explain the use of attributes in Petri nets (which do not occur in P/T-Systems and HLPNGs), Sect. 3.5.2 briefly discusses the definition of another Petri net type: signal-event nets (SE-Nets). This type is then used later again to explain how to extend the graphical representation of some features of a Petri net.

For more complex Petri net types, we can also define the mapping from the concepts of the Petri net type to their representation in XML. Some Petri net types have a quite sophisticated syntax for their textual labels, which need to be parsed in some way – and sometimes also linked to other *symbols* of the net. Such labels are called *structured labels*, and Petri net types using such labels are called *structured Petri net types*. For these Petri net types, a *parser* and a *linker* for the structural labels must be provided. The parser is needed to convert the text of the label from its concrete syntax to its abstract syntax or "structure"; the linker is needed for linking the use of symbols in some labels to their definition in others. We use the example of high-level Petri nets (HLPNG) for discussing the relevant details in Sect. 3.5.3.

In the end, in Sect. 3.5.4, we will provide a short overview and summary of the main concepts and steps for creating a Petri net type definition

3.5.1 Simple Petri net type definitions: PTNet

The definition of P/T-Systems follows almost exactly the idea outlined already in Sect. 1.2.2, 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. 1.2 on page 5 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 plug-in projects that are relevant for the Petri net type definition of P/T-Systems are `org.pnml.tools.epnk.pntypes` and the mostly automatically generated project `org.pnml.tools.epnk.pntypes.edit`.

3.5.1.1 The model

The type `PTNet` is defined in project `org.pnml.tools.epnk.pntypes`. The main part is the Ecore model in "PTnet.ecore" in the folder "model", where the diagram information is contained in "PTNet.ecorediag". The diagram is shown in Fig. 3.4.

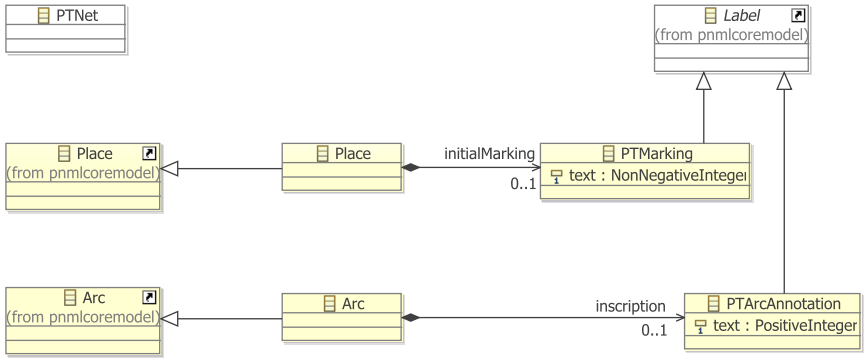

Figure 3.4: The Ecore model for the Petri net type PTNet

There are only some minor, but important differences, to the model from Fig. 1.2 on page 5. We discuss these differences below:

1. There is a class `PTNet` in the Ecore model, which does not occur in the conceptual model. The reason is that packages are not very tangible in programming and in the Eclipse plug-in mechanisms. Therefore, we define a Petri net type as an explicit class within that package; this class inherits from `PetriNetType` from the PNML core model (package `pnmlcoremodel`), which is not shown graphically in the diagram. It is this class (`PTNet` in our example) that is plugged in as a Petri net type definition 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 package `pnmlcoremodel`. And it is these new classes to which the additional labels are attached (initial marking and inscription). The reason for using inheritance here instead of merging packages is, that Ecore does not have the concept of merging packages[19]. 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 have additional labels.

Note that the name of the two classes, `Place` and `Arc` are the same as in the PNML core package, which is not ambiguous since these new classes are defined in another new package. For now, we assume that the names of these classes are the same as in the PNML core model[20].

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 data types from XML Schema which are used in ISO/IEC 15909-2. The cardinality for the `text` attributes is 1 in both cases – the same as in the conceptual model of ISO/IEC 15909-2.

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 the ePNK considers every label that is not an attribute to be an annotation – therefore, there is no need for an explicit class Annotation in the PNML core model of the ePNK[21].

[19] It might be a good idea not to use the merge concept for extending the place and transition of the PNML core model in ISO/IEC 15909-2 when defining a new Petri net type. The merge does not work properly when nets of different types are used within the same document, but which is legal according to ISO/IEC 15909-2

[20] In principle, the names could be different; but this would require some extra programming, which we do not discuss in detail in this manual. Basically, the reflective code of the methods for creating the instance of the respective Petri net element in class `PetriNetTypeImpl` (see Sect. 3.3.4.3) need to be overridden, so that they return an object of the correct type.

[21] Actually, there are classes `NetAnnotation` and ObjectAnnotation in the ePNK. But, these classes represent annotations on top of an existing Petri net, and are *not* annotations in the the sense of the PNML core model of ISO/IEC 15909-2 at all.

5. A last difference is that there is no OCL constraint in this model. In the ePNK, constraints are plugged in in a different way: as EMF constraints, which is discussed in Sect. 3.5.1.4.

Such a model and diagram can be created and edited by the graphical editor of "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).

Note that the Ecore package that contains the definition of a simple Petri net type must meet some conditions:

1. It must contain exactly one class that is derived from `PetriNetType`. The name of this class, however, can be chosen freely.

2. There can be classes which are derived from any of the following classes of the PNML core model: `Place`, `Transition`, `Arc`, `Page`, `RefPlace` or `RefTransition`. The names of these derived classes in the new package should the same[22].

3. These derived classes can have any number of references to some other classes. The classes that these references refer to must be derived from the class `Label` of the PNML core model, and the feature must be a composition (a containment feature); the name and the cardinality of the features can be chosen freely. If the feature has cardinality "many", this means that the respective object can have multiple labels of that kind.

4. The classes that are derived from class `Label` must have exactly one attribute, which has the name `text` and cardinality "1". The type of the attribute `text` can freely be chosen; it can be an Ecore built-in data type, a user-defined data type, an enumeration type, or a data type imported from other packages. The classes that are directly derived from `Label` must not have any reference[23].

5. Note that the package may also contain one class that is derived from the class `PetriNet` from the PNML core model. The name of that class can be freely chosen. The derived class may have the same kind of references as discussed for Petri net objects above. In that case, the

[22] By some programming, however, the names can be changed

[23] We will see later that this is slightly relaxed for structured labels, which are discussed in Sect. 3.5.3.2. Structured labels, however, are not directly derived from class `Label`; they are derived from class `StructuredLabel`.

Petri net itself can have labels attached to it[24], which are called *net labels*.

3.5.1.2 Generating the code

The code generation from that Ecore model follows the EMF standard procedure. In short, we need to generate the *model code* and the *edit code*. But, we 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 contains some configuration information on how the code should be generated. For example, the generator model contains the information to which project and which packages, the Java code for the model should be generated. The generator model also allows us to configure the generation of the EMF tree editor; for example, we 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). The generator model 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, the Ecore model has references to other models, like the PNML core model, and their "genmodels"; in the wizard for creating the "genmodel", make sure that you do not choose these other models as so-called 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 generate[25] the code for the model (*model code*), and the code with the infrastructure for all editors, which is called *edit code*. We cold also generate a simple tree editor for this model; but we do not need it; so we recommend not to generated it in order to avoid confusion and an inflation of file extensions attached to editors that are not used. Generating the code can be done by opening the "genmodel",

[24] We would discourage to define Petri net types were labels can be attached directly to a Petri net. But since ISO/IEC 15909-2 mandates that this is possible for HLPNGs, the ePNK provides the possibility to define such net labels.

[25] If you have imported the plug-in projects for P/T-Nets, the code is already generated. So, you do not need to generated anything. You would need to do that only for a new own Petri net type definition. The following discussion pretends that the P/T-Net is your new Petri net type definition were you just created the Ecore model.

and then selecting (after clicking the right mouse button) "Generate Model Code" and "Generate Edit Code". After that, you will find[26] 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 is generated in a project with the same name extended by a suffix ".edit". We do not need to change anything in the edit code[27]. Note that we must generate the edit code, in order for our Petri net type definition to work properly. If we do not do that that, we will get some exceptions when using the ePNK with the new type.

3.5.1.3 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 there is a new Petri net type. To this end, we need to define another extension, which makes the new Petri net type known to the ePNK. Before, we can do that, we need to make two minor changes in the automatically generated code. We need to make the constructor of the class that represents the new Petri net type public (by default it is protected); in our example, this concerns the constructor of the class `PTNetImpl`, which can be found in the automatically generated package `org.pnml.tols.epnk.pntypes.ptnet.impl`.

Listing 3.22 shows this class with the manually changed and extended parts marked in red. You can see the constructor, which is public now. The manual change is indicated also by the `@generated NOT` tag[28]. The second manual change is the addition of the `toString()` method. This method defines the value of the PNML type attribute for nets of that particular Petri net type: the types unique URI. Here, we use the one from ISO/IEC 15909-2 for P/T-Systems. If you define a new Petri net type, you need to make sure that you use one that is not used by other Petri net types already.

With the constructor public, we can plug in 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 3.23 shows the relevant part from the "plugin.xml" file, which can be found in the project

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

[27] In the `org.pnml.tools.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.

[28] 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`.

Listing 3.22: 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";
  }

}
```

`org.pnml.tools.epnk.pntypes`. The attribute `point` refers to the ePNK

Listing 3.23: 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>
```

type definition extension point, the `id` is a unique identifier for the new type within the ePNK, and the attribute `name` gives the type extension some conclusive name (we use the one from ISO/IEC 15909-2). The `type` element refers to the class that implements the new type; in our example, this is our `PTNetImpl` class – with its fully qualified name. In general, the class that is chosen here must extend the class `PetriNetTypeImpl` from the PNML core model code and which must 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.

If we started the runtime workbench now and used the ePNK editor, it would offer us a Petri net of the new type, when we create a child element of a Petri net document. But, it would be better to wait with that until, we have also added the constraints for connecting arcs, below.

3.5.1.4 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 or from and to pages. In the conceptual model of PTNets, this is excluded by an OCL constraint in the UML model already. In the ePNK, this constraint must be added separately, which is done by the standard mechanisms of EMF Validation in the "plugin.xml".

Listing 3.24 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 (lines 31–34). This OCL expression resembles the one from the conceptual UML model, but is syntactically slightly different – which is

Listing 3.24: 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>
```

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 (lines 23–29); in this example, it is the `Arc` of the `ptnet` package (which we refer to by the URI that is defined in the Ecore model of the Petri net type). 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 changes 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, and we go through it only briefly. 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. Constraints are defined for a category; we use the one defined by the ePNK 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 3-digit 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 choose in our example. Other constraints, like correctness of structured labels, might be defined to be in batch mode; then, the graphical editor will allow for syntactically incorrect labels, but the violation will be reported when explicitly 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 Java constraints in the HLPNG definition (see Sect. 3.5.3.3). Often, Java is more convenient for implementing more complex 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. 1.1 on page 7 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 Sect. 3.5.3.

3.5.2 Petri net type definitions with attributes: SE-nets

In this section, we discuss the Petri net type definition for *signal-event nets* (*SE-nets*), which we had discussed in Sect. 2.4.3 from the end user's point of view. We present this example for several reasons: First and foremost, in the definition of SE-nets, we can show how to use *attributes* in Petri net type definitions. Second, the most prominent feature of SE-net are signal arcs, which run between two transitions; and these arcs are associated with a specific graphical representation – an arc with a flash symbol. Therefore, we come back to this example later in this manual when we define the graphical appearance of Petri nets (Sect. 3.6). Third, the Ecore model for SE-nets contains one feature, which would not be legal in PNML; therefore, we can use this example to show some subtle changes in order to make this illegal feature "invisible" to PNML.

As you could see in Fig. 2.7 on page 23, signal-event nets have different kinds of arcs. Read arcs, inhibitor arcs, and signal arcs. Therefore, arcs need a label that indicates that type. The type definition for SE-nets is defined in the ePNK plug-in projects `org.pnml.tools.epnk.pntypes.signalnets` and `org.pnml.tools.epnk.pntypes.signalnets.edit`[29].

Figure 3.5 shows the Ecore model with the Petri net type definition for signal-event nets, which can be found in the folder "model" of plug-in project `org.pnml.tools.epnk.pntypes.signalnets`. In this model, the class `Arc` is equipped with an `ArcType`, which has a `text` attribute, the type of which is the enumeration `ArcTypes` – also defined in this model. Note that the enumeration has only two possible values `read` for read arcs and `inhibit` for inhibitor arcs. If there is no `ArcType` the arc is considered to be a normal arc, when it is running between a place and transition or vice versa, or as a signal arc, when it is running between two transitions. Moreover, the Ecore model defines that there can be a `Marking` for places. Both label classes inherit from `Attribute`, which means that these labels are not shown as

[29] Unfortunetely, these two rojects were configured in such a way that you cannot import the source code of these projects as discussed earlier. Therefore, the source code is made available separately – you can download the source code of the respective projects from `http://www2.imm.dtu.dk/~ekki/projects/ePNK/install-details.html`; and then import them to the workspace.

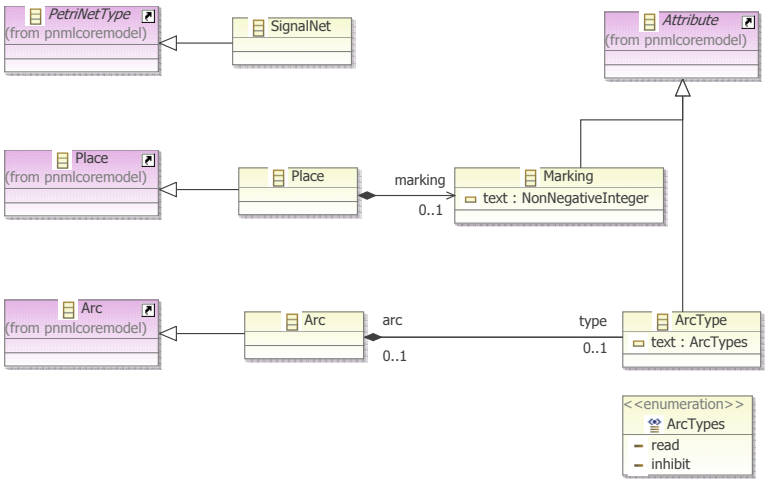

Figure 3.5: The Ecore model for SE-Nets

annotations, but in the properties view, when the respective arc or place are selected in the editor (see Fig. 2.7). We will see Sect. 3.6, how the value of these attributes can be shown in the graphical representation of the place or the arc. All we need to do for a label in the Petri net type definition to be considered an attribute by the ePNK is deriving it from the class `Attribute` of the PNML core model.

If we have a closer look at the Ecore model from Fig. 3.5, we see that the class `ArcType` has a reference `arc` which points back to the arc that "owns" that type – the reference `arc` is, actually, an opposite of reference `type`. This additional reference allows us to navigate back from the arc type to the respective arc, which makes it easier to formulate some constraints for arcs and their arc types[30]. As we had discussed earlier in Sect. 3.5.1.1, classes derived from `Label` and also from `Attribute` are not allowed to have any feature other than the `text` attribute. The reason for this restriction is that PNML does not allow us to serialize this feature. So, we need to make sure that such references to arcs are not serialized – conceptually this is not necessary anyway, since the reference `arc` is the opposite of the reference `type`. Therefore, we switch the serialisation of the feature `arc` off. This can done by selecting the resp. reference in the editor for the Ecore model, and then, in the properties view, selecting the "Advanced"[31] section, and

[30] In the current version, this feature is not used, though.

[31] This "Advanced" section shows all kinds of advanced setting of the respective Ecore element. In case you are new to Ecore and you do not understand the concept of "tran-

then set the property "transient" to "true", meaning that this feature is not serialised to a file (the ePNK serialisation mechanism takes that into account).

From the Ecore model above, we can create the "genmodel" and generated the model code and the edit code as discussed in Sect. 3.5.1.2. And we would need to do the same manual change: implement the `toString()` method, so that it returns the unique URI for that type; and we would need to make the constructor of the class `SignalNetImpl` public. In this example, however, we chose a different way – we create a class `SignalNetFactory` that inherits from `SignalNetImpl` without any additional attributes, methods or constructors. Since the implicit default constructor of `SignalNetFactory` is public, this will do the job. This is actually the preferred method, since regeneration of the model code after a model change does not need any manual changes anymore.

Plugging in the Petri net type to the ePNK extension point works as described in Sect. 3.5.1.3. Listing 3.25 shows the resulting fragment of the "plugin.xml" (with some minor omissions).

Listing 3.25: Plugging in SE-Nets

```
<extension
    id="org.pnml.tools.epnk.pntypes.signalnets"
    name="Signal Nets"
    point="org.pnml.tools.epnk.pntd">
  <type
      class="org.pnml. ... .signalnets.factories.SignalNetFactory"
      description="Signal nets">
  </type>
</extension>
```

Listing 3.26 shows the constraint for SE-nets, which makes sure that an arc type can only be present for arcs that run from a place to a transition; it also guarantees that arcs run from a place to a transition, from a transition to a place, or between two transitions. It is a live constraint, which needs to be checked, whenever the source or target of an arc are set, and whenever the arc type is set.

With these definitions, the ePNK would know what SE-nets are – still the inhibitor arcs and the signal arcs would not yet appear as shown in

sient", do not worry. Just ignore this for now.

Listing 3.26: Adding the constraint for SE-nets

```
<extension point="org.eclipse.emf.validation.constraintProviders">
  <constraintProvider cache="true" mode="Live">
    <package
        namespaceUri="http://org.pnml.tools/epnk/types/signalnets">
    </package>
    <constraints categories="org.pnml.tools.epnk.validation">
      <constraint
          id="org.pnml. ... .validation.correct-arc-connection"
          lang="OCL" mode="Live"
          name="Arc connection constraint for signal nets"
          severity="ERROR" statusCode="401">
        <message>
          The arc {0} with this arc type is not allowed ...
        </message>
        <description>
          Arcs must be between a place and a transition, ...
        </description>
        <target
            class="Arc:http://org.pnml.tools/epnk/types/signalnets">
          <event name="Set">
            <feature name="source"></feature>
            <feature name="target"></feature>
            <feature name="type"></feature>
          </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) and
          self.type->size() = 0 )
        or
        ( self.source.oclIsKindOf(pnmlcoremodel::TransitionNode) and
          self.target.oclIsKindOf(pnmlcoremodel::TransitionNode) and
          self.type->size() = 0 )
        ]]>
      </constraint>
    </constraints>
  </constraintProvider>
</extension>
```

Fig. 2.7. To this end, we still need to extend the graphical appearance of SE-nets, which will be discussed in Sect. 3.6.

3.5.3 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. 2.5.2 and for a detailed discussion of all models and concepts, we refer to [6]).

3.5.3.1 Overview of HLPNGs

As discussed in Sect. 2.5.2, HLPNGs have different kinds of complex labels: *declarations* of variables, sorts, and operators; *types* defining the sort of the tokens of a place, *markings* which are multiset terms defining the initial marking of a place, *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 anymore – some labels, like markings, arc annotations, or conditions may use *symbols* that are defined in other labels – in particular, in the declarations.

Figure 3.6 shows the Ecore model defining the concepts of HLPNGs, which can be found in the folder "model" in project[32] `org.pnml.tools.epnk.pntypes.hlpngs.pntd`. This model follows the same principles as the model for PTNets, which was discussed in Sect. 3.5.1.1. The main differences are that the defined Petri net type `HLPNG` extends a more advanced class `StructuredPetriNetType`, and 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 is discussed in Sect. 3.5.3.2.

The actual contents of all these labels is defined in their containment `structure`; note that we use `Term` as the contents for the labels `HLMarking`, `Condition`, and `HLAnnotation`, since all of them are terms – just with different additional constraints imposed on them (see Sect. 3.5.3.3). Note that by

[32]This is the plug-in 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 concepts 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. You can import all of these projects to your workspace by the Eclipse "Import As" feature in the "Plug-ins" view.

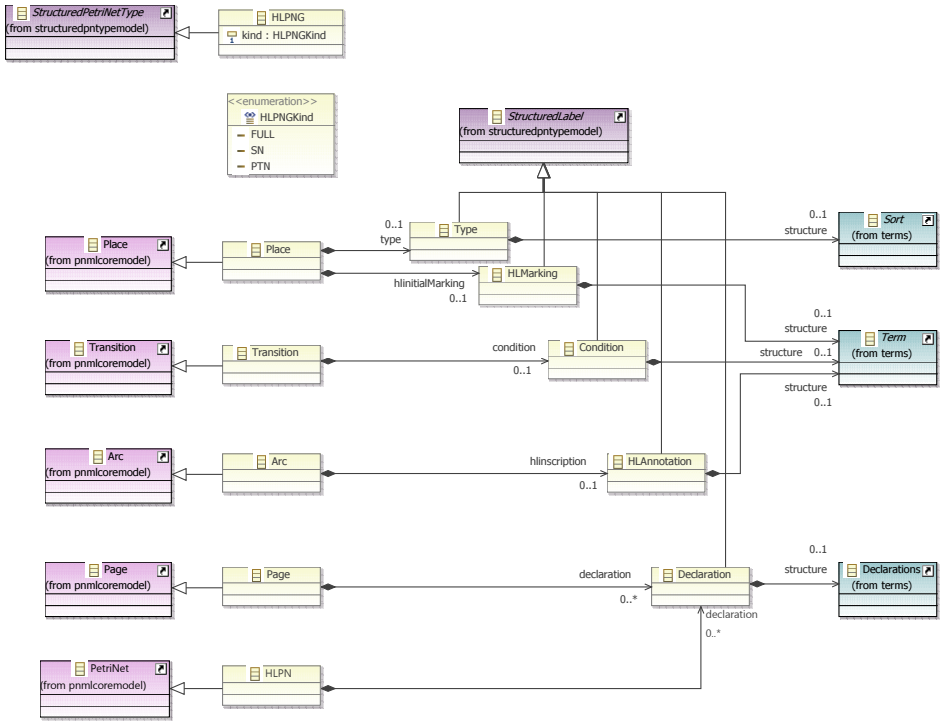

Figure 3.6: The Ecore model for HLPNGs

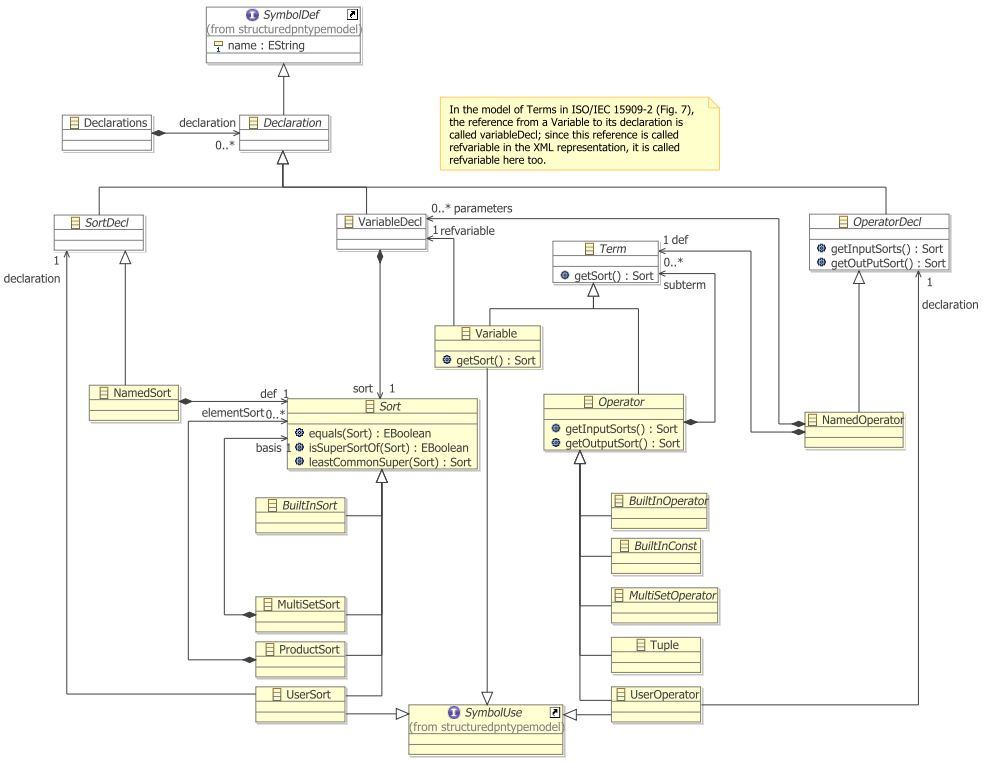

Figure 3.7: The Ecore model for the main concepts of HLPNGs

contrast to normal labels and attributes, *structured labels* can have – actually must have – a composition, which normally[33] has the name `structure`. But there should not be any other features than that.

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. 3.7 – this as well as the diagrams of all the other models can be found in the plug-in `org.pnml.tools.epnk.pntypes.hlpngs.datatypes`. For a detailed discussion of these models and their concepts, we refer to [6]. 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

[33]The name could be changed, but this would require some programming, which will be discussed later.

their use in a uniform and generic way – on the side, making the concepts of *symbol definition* and *symbol use* explicit, so that the model is more concise. These concepts are part of the PNML core model concerning structured Petri net types, which will be discussed in the next section.

One other issue worth noting in the Ecore model of Fig. 3.6 is the class `HLPN`, which extends class `PetriNet` from the PNML core model. This represents the Petri net itself. Normally, Ecore models defining a new Petri net type would not need to extend the class `PetriNet` itself; it would be enough to extend the class `PetriNetType`. HLPNGs, however, have so-called *net labels*, which are labels that are directly attached to the net – and not to a page. For net types with net labels, the class `PetriNet` must be extended and equipped with compositions to the respective labels – in our example, these are declarations. But, we would discourage defining such net labels for Petri nets types.

3.5.3.2 Structured Petri net types and structured labels

As mentioned above, the ePNK provides some general interfaces and infrastructure for defining structured Petri net types, which distill the general concepts of more complex Petri net 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. 3.8. 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 stores the contents of this label as a text String. The actual structural contents is defined by classes that extend it (we have seen some examples in Fig. 3.6 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 `parse()` 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. If the label could be parsed, it must

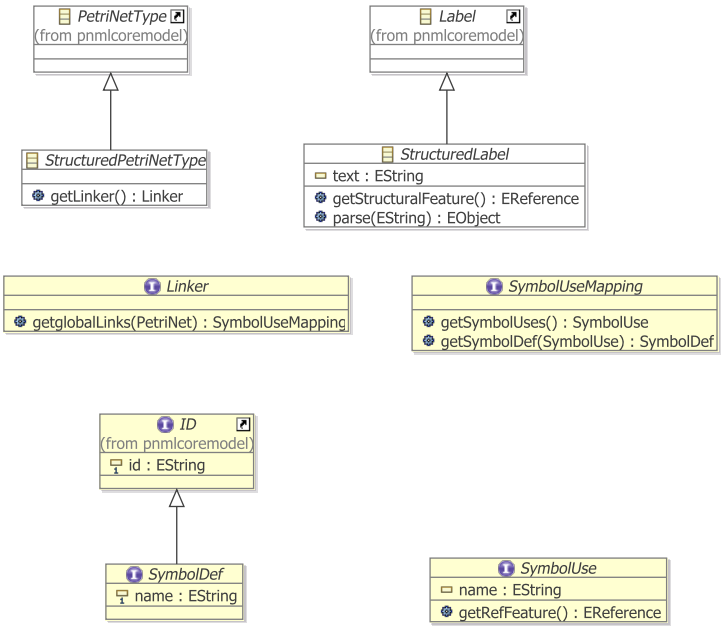

Figure 3.8: The model for structured Petri net types

return some object (to be precise an `EObject` which is the EMF version of objects) with all the substructure of that label – the abstract syntax of the label. In particular, that object must have a type that is compatible with the label's 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` has 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 access to the linker; to this end, the class `StructuredPetriNetType` has a method `getLinker()`, which must be implemented by classes that extend it. `Linker` is an interface: a single method `getglobalLinks()`, which takes a Petri net and returns a `SymbolUseMapping`, which is also an interface. Conceptu-

ally, the class `SymbolUseMapping` maps every `SymbolUse` to its definition `SymbolDef`. All the symbol uses for which there exists a mapping, can be obtained (as a list) via the method `getSymbolUses()`; 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 structured labels. We will have a look at the implementation of some examples next: We consider the label `Condition` in the Petri net type definition for HLPNGs again (see Fig. 3.6) – 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 are 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, class `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 the 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 is used to look up the definition 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 order to plug in the Petri net type definition to the ePNK, we need to make the constructor public in the class `HLPNGImpl`, and we need to implement the `toString()` method so that it returns the unique URI of HLPNGs method (as discussed in Sect. 3.5.1.3).

3.5.3.3 Constraints

For HLPNGs, we needed to implement quite many constraints. As an example for a *Java constraint*, we discuss one of these constraints 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 an OCL constraint that forbids connecting places with places and transitions with transitions. But, this is exactly the same 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 implementations of the Java constraints 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 3.27 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[34]. If it is not, 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 similar to plugging in OCL constraints. The relevant fragment of the "plugin.xml" is shown in Listing 3.28. The main differences are that the attribute `lang` is "Java" now and the attribute `class` refers to the class `ConditionIsBoolType`, which was discussed above. As target class, the transition class of HLPNGs is 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 the constraint might be violated during editing; a violation will be detected and reported only when the user explicitly invokes the 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.

[34]The implementation of `getSort()` for terms is actually quite complex; it amounts to implementing a type system for the annotation language of HLPNGs. But we do not discuss the details here.

Listing 3.27: 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 3.28: 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>
```

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

3.5.3.4 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 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[35].

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 `meta data` represents the table, to which the entries should be added when the method is called.

Let us have a look at some examples. Listing 3.29 shows an excerpt of the `registerExtendedPNMLMetaData()` method of the class `HLPNGImpl`, which implements the Petri net type for HLPNGs. 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 3.29 define how the structure feature of 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 to XML by this entry; in this case, it is the composition from the class `Type` to the class `Sort` (see Fig. 3.6 on page 120). The source and target classes are mentioned explicitly as second and third parameter again. We refer to the feature and the two classes via the singleton classes that describes the elements of the packages (HLPNGdefinition and Terms), which are automatically generated by EMF. These *package classes*, provide access to all the classes and features within a package (see [2] for more

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

Listing 3.29: 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());

    ...
}
```

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 an XML attribute, that might be necessary for creating an Ecore object from the XML element (we will discuss an example later). In most cases, this XML attribute is 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. This parameter can be left empty, when the Ecore instance can be constructed reflectively from the information on the object class only.

The ePNK uses this table and its entries in two directions: In the one direction, the table is used to serialise a Petri net to its XML syntax; in the other direction, the table is used to create the model elements from the XML syntax. In the latter case, the *factories* play an important role. Listing 3.30 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 the third (representing the object class) and the 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 the values of some of the XML element's attributes for creating an object of the appropriate type). 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

Listing 3.30: 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);

}
```

Listing 3.31: Mappings of an attribute

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

an object for some XML attribute. The implementation of these factories is straightforward 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 factory can be set to `null`. In which case the standard mechanism for creating an object of the target class will be used; therefore, we need factories only in very special cases. In most of the cases, the factory can be set to `null`[36].

Listing 3.31 shows an example[37] 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. Some of the PNML constructs cannot be mapped to XML by the mechanisms provided by EMF Extended Metadata. Therefore, the ePNK needed to provide its own mechanism for mapping Ecore concepts to XML. In the rest of this section, we discuss some of these special situations.

To this end, we consider the serialisation of the simple term `x‘f(x,x)`, where `x` is a variable and `f` is a user defined operator. The PNML representation is shown in Listing 3.32, where "5" is the unique id of variable `x` and "1" is the id of the user defined operator `f`, In addition to being a bit verbose, there is one thing that is special about this mapping: There is an

[36] Note that except for two features, which were used 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 and easier to understand).

[37] Actually, this is the only example of this kind in HLPNGs.

Listing 3.32: 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>
```

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 this 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 such an XML element an *association element*. The mapping for these association elements is shown in Listing 3.33. 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 level above[38].

The class `TermAssoc` does not need to be programmed. This class, as

[38]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 3.33: 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());
```

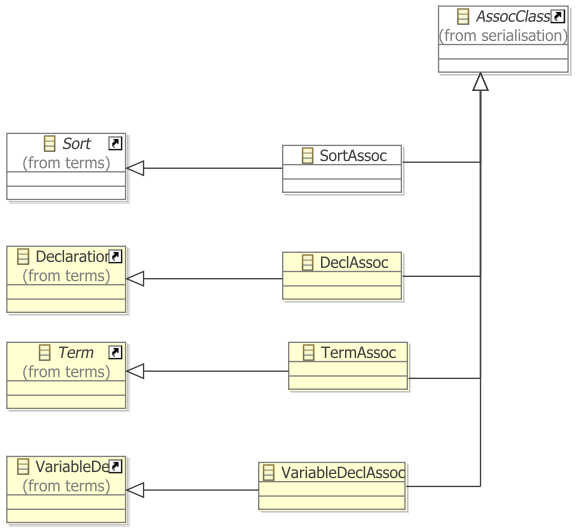

Figure 3.9: The package `hlpngserialisation`

well as the other classes for representing association elements, could completely be generated from a model. This model is shown in Fig. 3.9. 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 classes will not occur in the model anymore, once it is completely loaded – they are only used while 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 3.32). In the case of parameters of an operation declaration, this is different: Listing 3.34 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 3.35. 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 parameter `true` says, that this is a bundled association. The second table 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*.

Listing 3.34: 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>
```

Listing 3.35: 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());
```

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-elements for which context independent element mappings exist (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 3.36 defines the composition `declaration` as the standard feature of `Declarations`. Note

Listing 3.36: Defining a standard feature

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

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

3.5.4 Petri net type definitions: Summary and overview

In Sect. 3.5.1–3.5.3, we have seen most of the mechanisms for defining new Petri net types. Basically, a Petri net type definition consists of a new Ecore package where the Petri net type of the PNML core model is extended and the classes for the extended Petri net elements are modelled. Form this model the major parts of the code (model and edit code) can be generated. In the generated code, some manual changes need to be made. The Ecore package needs to follow some modelling principles that are discussed in Sect. 3.5.1.1 and the manual changes are discussed in Sect. 3.5.1.2. All the extensions must be added as *labels* of the respective kind of node of the Petri net.

If a label should not be shown as annotation of the respective element in the graphical editor of the ePNK, this can be achieved by deriving it from the ePNK class `Attribute`. Attributes can be edited in the properties view of the ePNK only. Sect. 3.5.2 discussed an example.

More complex Petri net types might require to also implement a parser and to store the actual information of the label not only as text but also as an abstract syntax tree in the PNML file. Such labels are called structured labels and have been discussed in Sect. 3.5.3.2. In case of complex Petri net types, it might also be necessary to customize the XML representation, of the labels and the concepts of its abstract syntax. To this end, the ePNK allows Petri net types to define a XML mapping, which is discussed in Sect. 3.5.3.4.

For all kinds of nets, it is possible to add additional constraints on top of the Ecore model of the respective type. These constraints are plugged in via the standard mechanisms for EMF Validation. Two examples are discussed in Sect. 3.5.1.4 and Sect. 3.5.3.3. The constrains can either be programmed in Java or can be OCL.

Note that it is possible to extend the class `PetriNet` of the PNML core model (when net labels are needed in the Petri net type). It is also possible to add labels to pages and reference nodes by extending the respective classes in the Ecore model for the new Petri net type.

When an annotation is defined for a page, the question is whether the respective annotation should be shown as a label annotated to the node page on the super page or whether the label should be shown as a page label on the page itself. The name of a page is shown as an annotation of the page on the super page; by default, an annotation of a page is shown as page labels on the page itself, if there can be multiple annotations of that kind for the page; and it is shown as a label of the page node on the super page, if there can be only one annotation of that kind. But, this can be changed by overriding the method `showLabelOnPage()` of the class `Page` of the respective Petri net type – which requires manual coding again.

3.6 Defining the graphical appearance

For some kinds or Petri nets, some places, transitions or arcs should be shown in a dedicated graphical representation. And the graphical appearance might depend on the context of the respective element – and the graphical appearance might change dependent on the changes of the context of this element. An example are signal arcs, inhibitor arcs, and read arcs in SE-nets, an example of which is shown in Fig. 3.10 again.

In this section, we discuss how such dedicated graphics can be plugged into the ePNK. To this end, we continue the discussion of the projects that implement SE-nets, which was started in Sect. 3.5.2. As you can see from Fig. 3.10, SE-nets have a dedicated graphics for arcs (as signal arc, read

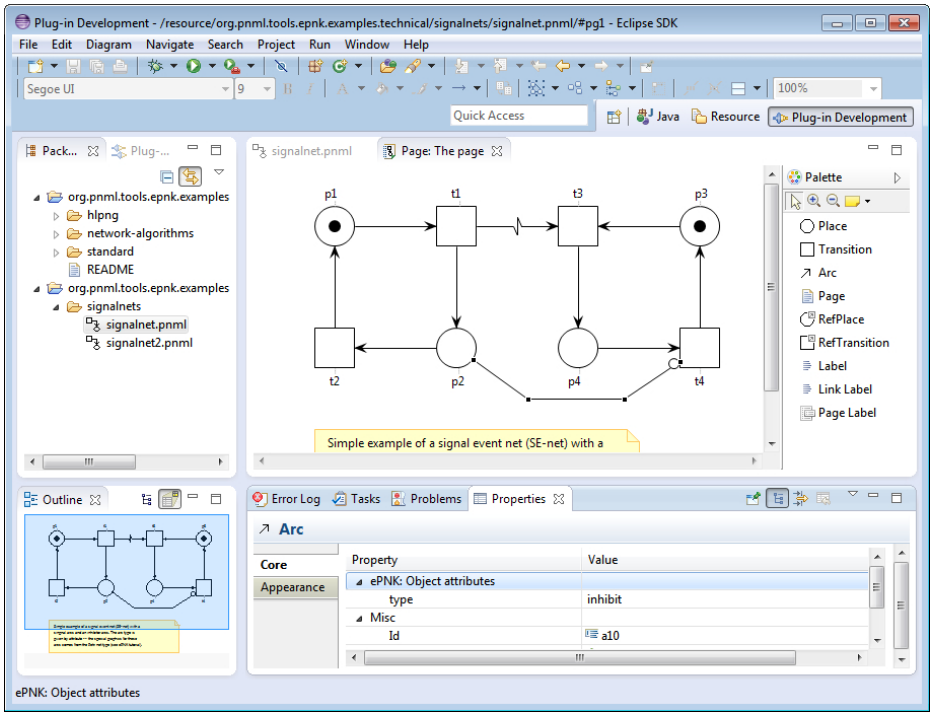

Figure 3.10: A SE-net with its dedicated graphics

arc, or inhibitor arc). But there is also a dedicated graphics for places: the marking is shown by black dots – up to some upper bound – in the respective places.

We start discussing the implementation of the dedicated graphical representation for arcs. To this end, we need to implement a *figure class*, which is the GEF/GMF terminology for the graphically visible elements (view) of a model element in an editor. Listing 3.37 shows the main part of the class `SignalnetArcFigure`, which implements the graphical appearance of the arcs of SE-nets (the class `SignalnetArcFigure` can be found in the package `org.pnml.tools.epnk.pntypes.signalnets.graphics.figures` of plug-in project `org.pnml.tools.epnk.pntypes.signalnets`). This class extends the class `ArcFigure` of the ePNK. In line 3, an enumeration of possible arc types is define, which is private to this class. Note that we do not re-use the enumeration from the model here, but define another enumeration, in order to make the implementation a bit simpler. The current type of the arc is stored as an attribute of this class (line 5). The constructor (lines 7–11) takes the arc (the model element behind this figure) as a parameter;

Listing 3.37: The class `SignalnetArcFigure`: main part

```
public class SignalnetArcFigure extends ArcFigure {

  private enum Type { NORMAL, READ, INHIBIT, SIGNAL }

  private Type type;

  public SignalnetArcFigure(Arc arc) {
    super(arc);
    type = getType();
    setGraphics();
  }

  @Override
  public void update() {
    Type oldType = type;
    type = getType();
    if (oldType != type) {
      setGraphics();
  } }

  private void setGraphics() {
    RotatableDecoration targetDecorator = null;
    RotatableDecoration sourceDecorator = null;

    if (type == Type.READ) {
      targetDecorator = new ReisigsArrowHeadDecoration();
      sourceDecorator = new ReisigsArrowHeadDecoration();
    } else if (type == Type.INHIBIT) {
      targetDecorator =  new CircleDecoration();
    } else if (type == Type.SIGNAL) {
      sourceDecorator = new FlashDecoration();
      targetDecorator = new ReisigsArrowHeadDecoration();
    } else {
      targetDecorator = new ReisigsArrowHeadDecoration();
    }

    this.setTargetDecoration(targetDecorator);
    this.setSourceDecoration(sourceDecorator);
  }

  ...
```

it calls the constructor of the super class `ArcFigure` of the ePNK, which also takes the arc as a parameter, then calculates the current type (by the private method `getType()`, which is shown in List. 3.38) and then properly sets the graphical features by calling the method `setGraphics()`, which is specific to this class.

The method `setGraphic()` changes the graphical features of the arc according to the current type of the arc (lines 21–39). In this example, we change the *decorations* of the arc only; we use the decorations at both ends (source and target). As decorations, we use the usual arrow shaped ones (`ReisigArrowHeadDecoration`[39]), circles (`CircleDecoration`), and flashes (`FlashDecoration`), which are provided by the ePNK. In the method, the variables for the decorations on both ends are initialized to `null` (lines 22-23). Then, dependent on the type of the arc the respective decorations are set. Note that the `FlashDecoration` is attached to the source, but it will actually show up in the middle of the arc (or actually in the middle of the first segment of the arc) by the specific way it is is implemented. The reason for this choice is that there can be at most one decoration at each end of a *connection*. Since signal arcs have two decorations, the flash and the arrow head, one needs to be at the source end of the connection. In the end, the only thing that is necessary to do is actually setting the decorations – note that calling the respective methods with `null`, means that there is no decoration for that connection.

Note that changes in the underlying model might make changes in the graphical appearance necessary. Such a change could be an explicit change of the type of the arc by the end user or just reconnecting an arc to a different kind of element. Whenever such a change happens, the ePNK notifies the figure of the affected model element by calling the method `update()`, which is specific to all extensible figure classes of the ePNK. It should be overridden by the extending classes. Lines 14–19 of List. 3.38 show the implementation of this method in our example. The type of the arc is computed again; if it changed, the `setGraphics()` method is called again.

This is all there is to do for implementing another appearance of an arc. Of course, this figure still needs to be plugged in, which is discussed later. In the update method, you could do all kinds of other changes such as changing the colour of the arc (`setForeGroundColor()` or the line style (`setLineStyle()`) – and many things more.

[39]This name was chosen in honour of Wolfgang Reisig, who insisted on arrow heads in Petri nets being drawn in a very specific way. The implementation of `ReisigArrowHeadDecoration` tries to meat Wolfgang Reisig's standards.

If you need other decorations than the ones that come with the ePNK, you can implement them yourself. But, we do not discuss this here since this is a GMF or, actually, an Eclipse draw2d concept. A look at the implementation of the ePNK decorations might give you a clue.

Listing 3.37 shows the last part of the class `SignalnetArcFigure`: the implementation of method `getType()`. This is mostly straightforward: computing the type based on the information of the arc underlying this figure. The only surprise might be the initial type check of `this.arc` for `Arc`. The reason is that `this.arc` refers to a final attribute of `ArcFigure` of the ePNK, which refers to the `Arc` of the PNML core model, whereas in the class `SignalnetArcFigure`, we need to refer to the `Arc` of the SE-net package – which has the same name, but in a different package.

Listing 3.38: The class `SignalnetArcFigure`: compute type

```
  ...

  private Type getType() {
    if (this.arc instanceof Arc) {
      ArcType arctype = ((Arc) arc).getType();
      if (arctype != null) {
        switch (arctype.getText().getValue()) {
        case ArcTypes.READ_VALUE:
          return Type.READ;
        case ArcTypes.INHIBIT_VALUE:
          return Type.INHIBIT;
        }
      } else {
        Node source = arc.getSource();
        Node target = arc.getTarget();
        if (source instanceof TransitionNode &&
          target instanceof TransitionNode) {
          return Type.SIGNAL;
        }
      }
    }
    return Type.NORMAL;
  }

}
```

In the above example, we have changed the appearance of the arc on

a very high level of programming, by changing the attributes of the figure. And if the desired graphical appearance can be achieved this way, this is the recommended way of doing this. In some cases, however, changing the attributes of the figure it not enough – we rather would need to "draw" some additional things. This can be done by using a different strategy for extending the figure: overriding the `fillShape()` or `outlineShape()` methods. We explain this strategy by another example: showing the initial marking by a respective number of black tokens in the place. Listing 3.39 shows the class `SignalnetPlaceFigure`, which implements this graphical appearance. The `update()` method just informs the figure that it should repaint itself[40] when something has changed. The actual appearance is now defined by overriding the method `fillShape()`. In this method, first, all the normal drawing of the place is done by calling the same method of the `super` class. After that, the marking of the place is computed[41]. If the marking is between 1 and 4, the respective number of tokens are drawn in the client area of the place. To this end, the drawing methods on the `graphics` object are used. Depending on the number of tokens, the appropriate positions are chosen (note that for space reasons, we omit the code for drawing four tokens). If there are more than four tokens, they are not represented as black circles anymore. They are "drawn" as a string representing the number of tokens.

Note that you could do more things and could also use some other low-level methods of figures to do that. But, we do not discuss that here. You might get some more inspiration my looking at another tutorial which implements some more exotic appearances of arcs, places and transitions in project `org.pnml.tools.epnk.extensions.tutorial.types`; the resp. figures can be found in the in package `org.pnml.tools.epnk.extensions.tutorial.types.arctypes.graphicalextensions.figures`.

At last we need to make the new figures defined for SE-nets known to the ePNK – we need to plug them in. This is done by implementing and plugging in a factory for these figures. The factory for the graphical extension for SE-nets is shown in List. 3.40. It extends the abstract ePNK class `GraphicalExtension`. The first method (line 4–8) defines, for which types of Petri nets this extension provides some graphics – as a list of classes of the respective Petri net types. In our example, this is the class representing SE-nets – obtained from the automatically generated package class. The

[40] We could have done that in a slightly smarter way so that repaint is called only if the marking has changed – as for the arcs.

[41] Since this requires some navigation in the model, this is delegated to a separate method `getMarking()`, which is not discussed here.

Listing 3.39: The class `SignalnetPlaceFigure`

```
public class SignalnetPlaceFigure extends PlaceFigure {

  public SignalnetPlaceFigure(Place place) {
    super(place);
  }

  public void update() {
    this.repaint();
  }

  protected void fillShape(Graphics graphics) {
    super.fillShape(graphics);
    Rectangle rectangle = this.getClientArea();

    int m = 0;
    if (place instanceof Place)
      m = getMarking((Place) place);
    int cx = rectangle.x + rectangle.width/2;
    int cy = rectangle.y + rectangle.height/2;
    if (m == 0) {
      return;
    } else if (m == 1) {
      graphics.setBackgroundColor(getForegroundColor());
      graphics.fillOval(cx-6, cy-6, 12, 12);
    } else if (m == 2) {
      graphics.setBackgroundColor(getForegroundColor());
      graphics.fillOval(cx-11, cy-11, 12, 12);
      graphics.fillOval(cx, cy, 12, 12);
    } else if (m == 3) {
      graphics.setBackgroundColor(getForegroundColor());
      graphics.fillOval(cx-6, cy-13, 12, 12);
      graphics.fillOval(cx-13, cy, 12, 12);
      graphics.fillOval(cx+1, cy, 12, 12);
    } else if (m == 4) {
      ...
    } else {
      graphics.drawString(""+m, cx-5, cy-7);
  } }

  private int getMarking(Place place) { ... }
}
```

Listing 3.40: The factory class `SignalnetGraphics`

```
public class SignalnetGraphics extends GraphicalExtension {

  @Override
  public List<EClass> getExtendedNetTypes() {
    ArrayList<EClass> list = new ArrayList<EClass>();
    list.add(SignalnetsPackage.eINSTANCE.getSignalNet());
    return list;
  }

  @Override
  public List<EClass> getExtendedNetObjects(EClass netType) {
    ArrayList<EClass> list = new ArrayList<EClass>();
    if (netType.equals(SignalnetsPackage.eINSTANCE.getSignalNet())) {
      list.add(SignalnetsPackage.eINSTANCE.getArc());
      list.add(SignalnetsPackage.eINSTANCE.getPlace());
    }
    return list;
  }

  @Override
  public ArcFigure createArcFigure(Arc arc) {
    if (arc instanceof org.pnml.tools.epnk.pntypes.signalnets.Arc) {
      return new SignalnetArcFigure(
          (org.pnml.tools.epnk.pntypes.signalnets.Arc) arc);
    }
    return null;
  }

  @Override
  public IUpdateableFigure createPlaceFigure(Place place) {
    if (place instanceof
        org.pnml.tools.epnk.pntypes.signalnets.Place) {
      return new SignalnetPlaceFigure(
          (org.pnml.tools.epnk.pntypes.signalnets.Place) place);
    }
    return null;
  }

}
```

second method (line 11–18) defines for which kinds of elements there is a specific graphics – again represented as a list of class objects. In our example, these are the classes representing the arc and the place of SE-nets. At last, there are two methods, that create the respective figure object for an element by using the respective constructors of the figure classes. Note that `GraphicalExtension` has also methods for creating a figure for transitions and other kinds of nodes – but we do not need to override them here since our extension does not provide special graphics for them.

Actually the class `GraphicalExtension` has some more methods, which define priorities for the graphical extensions – in case more than one graphical extension is plugged in and applies to the same element. And it can also be defined, whether a graphical extension should apply to all subtypes of a Petri net type or not. The meaning of the methods is documented as Java doc comments for the methods in the interface for the factory `IGraphicalExtension`.

At last, the graphical extension needs to be plugged in to the ePNK. The relevant part of the "plugin.xml" of the project is shown in List. 3.41. This is straightforward. The attribute "point" of the extension refers to the ePNK extension point `org.pnml.tools.epnk.diagram.graphics`, and the class attribute of the graphicsextension refers to the factory of List. 3.40.

Listing 3.41: Plugging in the graphical extension

```
1 <extension
      id="org.pnml.tools.epnk.pntypes.signalnets.graphics"
      name="Signal event net graphical extensions"
      point="org.pnml.tools.epnk.diagram.graphics">
    <graphicsextension
6     class="org.pnml. ... .signalnets.graphics.SignalnetGraphics"
      description="Special graphics for ... signal event nets">
    </graphicsextension>
  </extension>
```

With this extension installed, the SE-nets should now look like the one in Fig. 3.10.

3.7 Adding tool specific information

As discussed in Sect. 1.2.1, the PNML allows tool specific information to be added to all elements of Petri nets – indicated by the special XML element

`<toolspecific>`. The ePNK reads and writes any tool specific information, and, in principle, the contents of these tool specific extensions could be accessed and modified via the class `AnyType`, which is defined in the plug-in `org.eclipse.emf.ecore`. But this is tedious and, basically, means navigating in the element's XML structure.

Therefore, the ePNK provides an extension point for plugging in tool specific extensions, so that they can be 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 is a tool specific extension mandated by ISO/IEC 15909-2:2011. We have seen an example already in Fig. 1.3 and Listing 1.1 on page 7.

This tool specific extension is defined in the project `org.pnml.tools.epnk.toolspecific.tokenpositions`. Most of this code in this project as well as the "plugin.xml" was automatically generated by EMF from the Ecore model "Tokenpositions.ecore" in the folder "model". This model is shown in Fig. 3.11. The new classes are `PNMLToolInfo` and `Tokengraphics`.

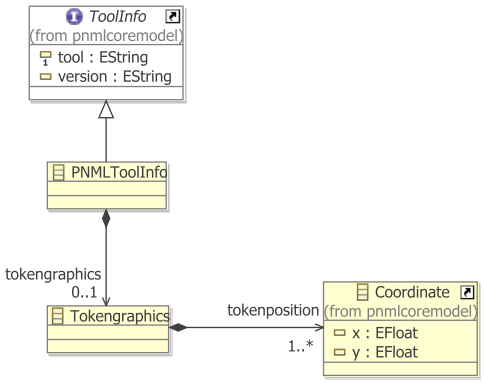

Figure 3.11: The model for tool specific extension tokenpositions

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` is re-used from the PNML core model.

From this model, the code can be generated in the same way as described in Sect. 3.5.1.2. 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 do 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 3.42. The

Listing 3.42: 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;
  }

}
```

factory implements the ePNK interface `ToolspecificExtensionFactory`, which consists of four methods. The two methods `createToolInfo()` create an instance of this tool specific extension; the method with the two String

parameters, `tool` and `version` is used, when the tool name and version are 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 as constants.

Listing 3.43 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, there is an attribute `class` that defines the factory for the tool specific extension; this class must implement the interface `Toolspecific ExtensionFactory`. Moreover, there is a brief description of this extension.

Listing 3.43: 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>
```

Note that the ePNK does not provide any way yet of explicitly defining the XML syntax of these extensions. The standard XMI serialisation will be used – which is compliant with ISO/IEC 15909-2:2011 for tool specific extensions. Eventually, the ePNK might provide a mapping mechanism similar to the one for Petri net types.

3.8 Overview of the ePNK and its projects

In this section, we give a brief overview of the different parts of the ePNK, the project structure and where to look for different kinds of functionality in the ePNK API an its projects. As mentioned earlier, developers should not change anything in these projects. Anyway, this overview should help 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, which could be used by developers in their extensions. Note that we do not discuss the details of the API here. In particular, we do not discuss the API concerning the code

that is generated from the Ecore models (model and edit code) since it is mostly straightforward. Section 3.3.4.1 gives a brief overview of the main principles behind the model code that is generated from Ecore models; for more information, we refer to the EMF book [2].

Like all extensions of Eclipse, the ePNK is organized in many Eclipse projects, which together make the ePNK. Most of these projects are so-called plug-in projects; and these are the ones most relevant for developers, since these are the projects to look up the API and to which extensions need to refer (in form of dependencies). In addition, there are some projects, which contain documentation only (like this manual), there are some projects that do not contain any code, but from which other projects are generated; and there are so-called features, which define collections of plug-in projects in order to deploy them. And there is a project for generating the ePNK update site from the features.

In this manual, we focus on the plug-in projects of the ePNK, the most important of which are listed below. We start with an overview of the ePNK core projects, which make up the framework of the ePNK[42]:

`org.pnml.tools.epnk`:
: This is the core project of the ePNK. In this project, you will find the PNML core model, some additional models, and the *model code*, that was generated from them. All the models, can be found in the folder "model". The PNML core model is contained in `PNMLCoreModel.ecore`; in order to avoid clutter in the graphical diagram, the core model is actually split up into three separate diagrams: `PNMLCoreModel.ecorediag` contains the most important concepts; `PNMLCoreModelGraphics.ecorediag` contains the graphical features of PNML; and `PNMLCoreModelProxies.ecorediag` contains some extensions to the PNML core model that are necessary to maintain labels in the graphical editor of the ePNK by so-called *label proxies* and *page label proxies*. These proxy elements, however, are not of any concern for normal developers.

  There are four other models in this project: `PNMLDataTypes.ecore` defines the *data types* for non-negative and positive numbers, which are used instead of the respective XML Schema Data Types of ISO/IEC 15909-2. `PNMLStructuredPNTypeModel.ecore` defines the concepts of structured Petri net types (see Sect. 3.5.3.2). `Serialisation.ecore` provides some general structure that is used for the XML serialisation

[42] Technically, all these plug-in projects are part of the ePNK features `org.pnml.tools.epnk.core` and `org.pnml.tools.epnk.extensions.basic`.

of so-called association elements (see Sect. 3.5.3.4). `PNMLPageDiagramInfo.ecore` is the model for storing the GMF diagram information for pages as ePNK tool specific information in PNML models, which are not of concern for normal developers.

This project provides also some *convenience classes* in package `org.pmm.tools.epnk.helpers`, which might be helpful in practice. The class `FlatAccess` allows handling a Petri net that is distributed over several pages as if it was flat (see Sect. 3.3.3 for an example). Another convenience class is `NetFunctions`, which provides many static methods for finding out to which net an element belongs, what the type of this net is, and for obtaining lists of elements of some kind of a given Petri net.

In this plug-in, also the two extension points of the ePNK are defined: one for defining new Petri net types (PNTD), another for defining new tool specific extensions.

In addition to the model code, also the so-called edit code and editor code are generated from these models, which together define the tree editor for PNML (see below).

Note that, in this project, also the constraint context and the constraint category `org.pnml.tools.epnk.validation`, to which all other constraints for new Petri net types should be added, are defined here (see Sect. 3.5.1.4 and 3.5.3.3).

`org.pnml.tools.epnk.edit:`

This project contains the edit code that was generated from the models in project `org.pnml.tools.epnk`. Though most of this code was automatically generated from the models, there are several manual changes, that enable generically dealing with plugged in Petri net type definitions.

Moreover, the generated standard EMF images in the folder `icons` were replaced by nicer ones.

`org.pnml.tools.epnk.editor:`

This project contains the editor code for the EMF tree editor for PNML that was generated from the models in project `org.pnml.tools.epnk`. In this project, there are only a few, but crucial extensions, that made it possible to integrate the EMF tree editor with the graphical editor for pages (see the plug-in project `org.pnml.tools.epnk.diagram` below).

`org.pnml.tools.epnk.pntypes`:
: This project contains the model and the generated model code for P/T-nets (`PTNet`), as well as the extension that plugs in this type to the ePNK. The model code is completely generated from the model `PTNet.ecore` in the folder "model", except for two changes in class `PTNetImpl` as discussed in Sect. 3.5.1.

`org.pnml.tools.epnk.pntypes.edit`:
: This is the project with the edit code that was generated from the Ecore model `PTnet.ecore` of project `org.pnml.tools.epnk.pntypes`. There are no manual changes in the generated code – only the icons in the folder `icons` were replaced by nicer ones.

`org.pnml.tools.epnk.toolspecific.tokenpositions`:
: In this project, the tool specific extension for token positions (as defined in ISO/IEC 15909-2) is defined (see Sect. 3.7). The model code and the edit project `org.pnml.tools.epnk.toolspecific.tokenpositions.edit` was generated (which does not contain any manual changes – not even nicer icons) from the model `Tokenposition.ecore`.

  Note that the ePNK does not take the information of these token positions into account in the graphical representation of places. The only reason they are defined in the ePNK is that ISO/IEC 15909-2:2011 mandates them – and we use this extension as an example to show how to define tool specific extensions in Sect. 3.7.

`org.pnml.tools.epnk.actions`:
: This project defines the standard actions of the ePNK, which are the pop-up menus for adding missing ids and for linking the labels of structured Petri net types (see Sect. 3.5.3.2).

  Moreover, the classes `AbstractEPNKAction` and `AbstractEPNKJob` are defined in this project, which are convenience classes to make it easier to define functions for the ePNK that run in the background (see Sect. 3.3.3).

`org.pnml.tools.epnk.diagram`:
: This project contains the code for the GMF-generated graphical editor for pages of Petri nets. This code was generated from the GMF models in project `org.pnml.tools.epnk.gmf`. But, there are major manual changes for making this graphical editor generic and for integrating it with the tree editor for PNML (see project `org.pnml.tools.epnk.editor`).

The package `org.pnml.tools.epnk.gmf.extensions.graphics` and its sub-packages `decorations` and `figures` are relevant for developers, who want to contribute specific graphical appearances for some Petri nets. Here you find the factory and the figures of the ePNK, which need to be extended for customizing the graphical appearance; and you can use the predefined ePNK decorations for that purpose (see Sect. 3.6 for more details).

`org.pnml.tools.epnk.gmf.integration:`
This project defines the pop-up menus for starting the graphical editor on a page that is selected in a tree editor or in a graphical editor.

`org.pnml.tools.epnk.annotations:`
This project defines the infrastructure for annotating Petri nets by applications. Up to now, the annotations provide the most basic concepts only – they will be extended in the future.

A simple example of annotating the context of transitions was discussed in Sect. 3.4.

`org.pnml.tools.epnk.applications:`
This project provides the interfaces and classes for implementing and starting applications on Petri nets, which was briefly discussed in Sect. 3.4.

`org.pnml.tools.epnk.applications.view:`
This project implements the applications view, that shows all currently running implementations. Developers would typically not directly use this project.

Next, we discuss some of the net types, and functions and applications, which we had discussed in this manual as a tutorial[43]:

`org.pnml.tools.epnk.functions.tutorials:`
This project contains the functions that were discussed in Sect. 3.3.1 and Sect. 3.3.2, which can serve as a guideline for defining own extension projects.

`org.pnml.tools.epnk.functions.modelchecking:`
This project contains the model checker extension for the ePNK that is discussed in Sect. 3.3.3. Note that the project `MCiE` is used in this

[43]Technically, these plug-in projects come from the ePNK feature `org.pnml.tools.epnk.extensions.tutorial`.

model checker extension only (MCiE is not relevant for anything else in the ePNK – except if you want to implement your own model checker based on MCiE).

`org.pnml.tools.epnk.tutorials.applications`:
: This project implements an example of an application using annotations. It contains the code for the transition context application which was discussed in Sect. 3.4.

`org.pnml.tools.epnk.pntypes.signalnets`:
: This project is the plug-in project implementing the Petri net type definition for SE-nets (see Sect. 3.5.2 and the graphical extensions for SE-nets (see Sect. 3.6).

`org.pnml.tools.epnk.pntypes.signalnets.edit`:
: This plug-in project contains the edit code, which is generated from the model of SE-nets – without any manual changes

The plug-in projects with the prefix `org.pnml.tools.epnk.pntypes.hlpng` resp. `org.pnml.tools.epnk.pntypes.hlpngs` together are used for defining high-level Petri nets (HLPNGs). The following list gives an overview in a bottom up way – ending with the actual Petri net type definition for HLPNGs:

`org.pnml.tools.epnk.pntypes.hlpngs.datatypes`:
: In this project, all the models that define the concepts of sorts, operators, variables and terms for HLPNGs are contained. In particular, there is a model `HLPNGDataTypes.ecore` for the general structure of terms, declarations and the built-in sorts and operators that occur in all versions of HLPNGs. And there are many more models and diagrams for specific versions of HLPNGs (Clauses 5.3.2–5.3.12 of ISO/IEC 15909-2:2011).

`org.pnml.tools.epnk.pntypes.hlpngs.datatypes.concretesyntax`:
: This project is an Xtext project that defines the grammar for the concrete syntax of the different labels of HLPNGs, from which a parser is generated. Here, we do not discuss the details of generating the parser. The manually written class `HLPNGParser` accesses the automatically generated parsing operations and provides methods for parsing every kind of label of HLPNGs. The other important manually written class is `HLPNGLinker`, which provides the global Linker for labels (as discussed in Sect. 3.5.3.2).

Note that the plug-in project ending with `concretesyntax.ui` was automatically created when the Xtext project was created by a wizard; it provides a stand-alone textual editor for the labels of HLPNGs. Since this stand-alone editor is not needed for the ePNK, this project is not part of the standard deployment of the ePNK.

`org.pnml.tools.epnk.pntypes.hlpng.pntd:`
This project actually combines all the parts discussed above into a Petri net type definition for HLPNGs. The main model is `HLPNG Definition.ecore`, which was shown in Fig. 3.7. The other model `HLPNGSerialisation.ecore` defines the auxiliary classes that temporarily store XML elements that represent associations and are used and referred to in the XML mappings (see Sect. 3.5.3.4) .

The global linker `HLPNGLinker` from the project `org.pnml.tools.epnk.pntypes.hlpngs.datatypes.concretesyntax` above is made available in the Petri net type by manually implementing the method `getLinker()` in class `HLPNGImpl`. The `parse()` method for the different structured labels are also manually implemented – they refer to the different `parserXXX()` methods of class `HLPNGParser` from the project `org.pnml.tools.epnk.pntypes.hlpngs.datatypes.concretesyntax` above.

`org.pnml.tools.epnk.helpers.unparse:`
This project implements a serialisation function that transforms the abstract syntax of HLPNG labels into the concrete syntax used for HLPNGs in the ePNK. This is mostly relevant for the end users – who want to generate the concrete syntax for the labels of HLPNGs (see Sect. 2.5.2 on page 37).

But, in some cases this label serialiser might also be relevant for developers that want to show terms of HLPNGs to the end user (the simulator for high-level nets is an example).

`org.pnml.tools.epnk.applications.hlpng.simulator:`
This is the plug-in project implementing the basic simulator for HLPNGs. We do not discuss the implementation in this manual, we discussed the simulator only from the end users' point of view in Sect. 2.6.3. You will find many more details in the master's thesis that implemented it [22].

3.9 Deploying extensions

In this section, we will briefly discuss how own extensions of the ePNK could be deployed, so that others can use it. Typically, an extension comprises several plug-in projects. In order to combine them, Eclipse provides a special kind of project, which is called a *feature* – and this is the unit in which Eclipse extensions should be deployed.

A feature in turn, can be used in an Eclipse *update site project* which can be used to create your own *update site*, so that your plug-ins (resp. features) could be installed from this site, similar to the way you installed the ePNK.

Since features and update sites are standard Eclipse concepts, we do not explain the details here. For now, looking up the keywords "feature" and "update site" in the Eclipse help (or googling for them) should be enough.

If you have a feature for the ePNK that might be interesting for a wider audience, you can also contact us, so that we can make it available via the ePNK update site.

Chapter 4

Complete Tutorial: Net type and Application

In this chapter, we discuss an example, which goes through the complete development process of a new Petri net type and an application for the ePNK. This serves as a tutorial which comes across all major aspects of the ePNK. In order to cover some more interesting aspects of the ePNK, we have chosen a slightly artificial Petri net type, which we call *technical Petri net type*. This technical Petri net type are classical Place/Transition Systems (P/T-systems)[1] with with three additional features: *read arcs*, *inhibitor arcs* and *reset arcs*. The ePNK application is a simple simulator for this net type.

As usual a *read arc* just check whether their is a token on the attached place, but does not change the marking of this place; an *inhibitor arc*, by contrast, checks that there is no token on the attached place, but also does not change the marking when the transition fires. A *reset arc* does not have any influence on the enabledness of the transition, but when it fires, all tokens from places attached to the transition with a reset arc, are removed.

Actually, in our *technical Petri net type*, a *reset arc* does not directly run from a place to a transition, but from a *page* to a *transition*. Then, all places contained in that page will be reset (i. e. all tokens will be removed) when the transition fires. In combination with using reference places, this reset of a page allows us to reset larger parts of a Petri net with a single reset arc which avoids cluttering the net with too many reset arcs.

We discuss this net type and the application from the end-user's point of view in Sect. 4.1. In Sect. 4.2, we discuss the conceptual idea of how to realize this application with the ePNK; at last, in Sect. 4.3, we discuss the major

[1] In a P/T-system a marking of a place may have multiple tokens.

technical steps to actually implement the application. This also covers the respective modelling and code generation steps, necessary configurations, and possible pitfalls and problems with the Eclipse tools and Eclipse IDE– and even the installation of the ePNK and EMF. All the code of that example is available online and we recommend that you install the example in your *development workspace* while working through the technical steps of this tutorial; which is discussed in Sect. 4.3.

4.1 The tool

Figure 4.1 shows a screenshot of our example tool that we are going to develop in this tutorial as an extension of the ePNK. Figure 4.1 shows the graphical editor of the ePNK with a Petri net of the new *technical Petri net type* that we use for this tutorial. The net is shown in the graphical editor of the ePNK (actually, you can see the net's two pages). We use Fig. 4.1 for briefly explaining the features of this *technical net type*. After that, we also briefly discuss the features of the simulator.

4.1.1 The technical net type

Figure 4.1 shows all the features of the *technical net type*. First of all, there are the standard concepts of Petri nets, such as *places*, *transitions* and *arcs* and the *initial marking* of places (indicated by a *label* attached to the *place*, which defines the number of *tokens* on that *places* initially). Moreover, there are the additional concepts of *pages* and *reference nodes*, which are coming from the PNML [6] or its ePNK implementation. See Sect. 1.2 for details. The example net shown in Fig. 4.1 consists of two *pages*, pg_1 on the left and pg_2 on the right. Page pg_2 is actually a sub-page of pg_1 indicated by the large rounded rectangle shown on pg_1; the contents of page pg_2 is shown in the graphical editor open on he right-hand side. Note that all the elements, which graphically appear to be inside the rounded rectangle representing page pg_2 on page pg_1 are actually objects of page pg_1; they are just arranged in such a way that they appear to be inside page pg_2. The only elements contained in pg_2 are the *reference places* shown on the right-hand side. These reference places, however, refer to the places p_2, p_3, p_4, p_5 and p_7 of the page pg_1. This is not directly visible in the graphical representation reference places; but you can see in the properties view that the reference place with id rp_5 (and name p_7) actually refers to the place p_7 on page pg_1. So we use the graphical alignment of the places p_2, p_3, p_4, p_5 and p_7 "inside" the rounded rectangle to put emphasize on this relation,

since it has meaning for the effect of the attached reset arc, which we discuss later.

In the *technical net type*, we can connect *places* and *transitions* with *normal* arcs. As usual *normal* arcs can run from a place to a transition or the other way round. In addition, there are two other kinds of arcs, which run from a place to a transition: *read* arcs and *inhibitor* arcs. As the name suggest, a *read* arc will not change the number of tokens on the attached place; but the attached transition can fire only, when there is at least on token on the place attached to the other end of the read arc. The inhibitor also does not change the number of tokens on the attached place when the transition fires. But, by contrast to the read arc, the transition will only be allowed to fire, if there is no token on the attached place (a token on the attached place "inhibits" the firing of the transition). A *read* arc is graphically represented as a line without arrow heads on either end, but it technically runs from a place to a transition. In our example, there is only one *read* arc, running from place p_7 to transition t_6. An *inhibitor* arc is graphically represented with a "lollipop" decoration at the transition end – and the direction of the arc is from the place to the transition. In our example, there is only one *inhibitor* arc, running from place p_7 to transition t_5.

The concepts discussed so far are all well-know concepts in Petri nets. We will see later, that in our simulator for *technical net type*, the end-user is able to deactivate *read* arcs and *inhibitor* for some simulation step – ignoring the *deactivated* arcs. There is one additional concept in our *technical net type*: these are *reset* arcs, which – just for the fun of it – run from a page to a transition. In our example, there is one *reset* arc running from page pg_2 to transition t_7. A reset arc does not have any effect on the enabledness of the attached transition; but, when the transition fires, the tokens from all places contained in the attached page will be removed. To be more precise, the tokens will be removed from the places contained on the page and from the places to which the *reference places* on that page refer to (we say that the *reference place resolve* to that *place*). In our example, these are the places p_2, p_3, p_4, p_5 and p_7 again. Graphically, a *reset* arc is represented by a dashed line with a double arrowhead. The dashed line indicating that this arc does not prevent the transition from firing; the double arrow head indicating that all tokens will be removed from the places on that page.

At last, there is a minor twist[2], which is only of graphical nature. The

[2]To be honest, we have chosen this feature only in order to demonstrate how to customize the graphical appearance of net objects in this tutorial.

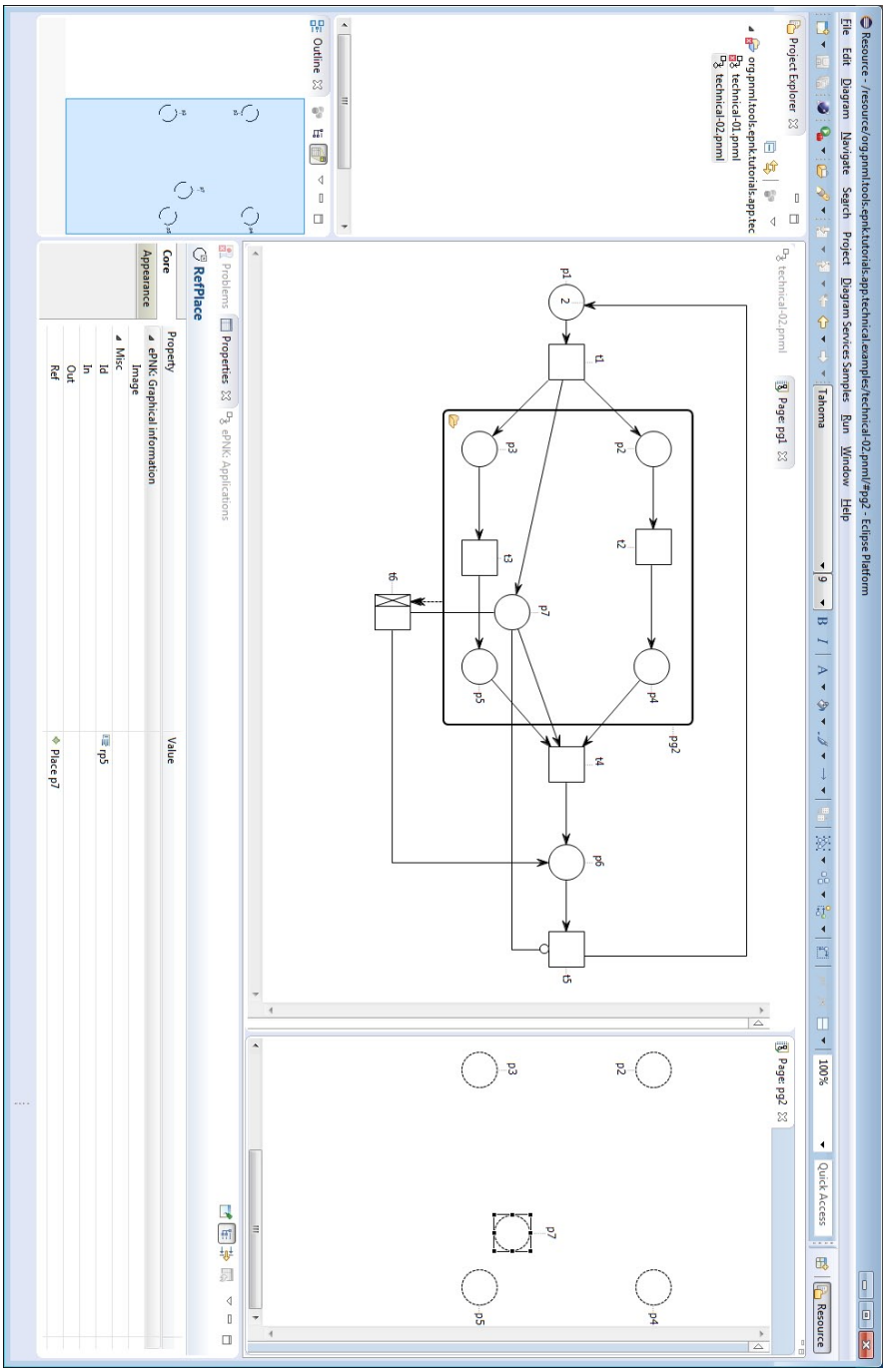

Figure 4.1: The example tool with an example of a technical net

graphical representation of transition t_7 shows a cross in the left third of the rectangle representing the transition. This indicates that this transition does not have any *normal* arc running to it. Since this situation is sometimes not desired, it is indicated with a special graphics; likewise, if there is no *normal* arc starting at the transition, this is graphically represented by a cross in the right third of the transition.

4.1.2 The application

In addition to realizing a new net type, which then can be created and edited in the graphical editor of the ePNK, the ePNK allows adding *application* on Petri nets. The applications could be some analysis, simulation or verification; and the applications can visualize which their results with a graphical feedback to the end-user, on top of the graphical representation in the graphical ePNK editor.

In this tutorial, we discuss how to develop a *simulator* for our *technical net type*, which we had introduced in Sect. 4.1.1. In the following, we discuss this simulator and its features from the end-user's point of view.

When a net or actually a page of a net of some type is open in the graphical editor of the ePNK (and when this editor has the focus), all applications that are defined for this net can be started by selecting the application in a small drop down menu in the *ePNK applications view*, which is indicated by a small red circle in Fig. 4.2. In this figure, the simulator called *Technical Simulator (Tutorial)* was started already; once started, some overlays in the graphical overlay indicate the initial marking of the net, and also the enabled transition are highlighted. The current marking of the net in the simulator is shown by a blue number to the top-right of the respective place; but only places which have at least one token hav such an annotation. This annotation for a place is also called a *marking* of the place – to be precise, it is the *current marking* of the place – as opposed to the *initial* one which is represented in the net itself. In Fig. 4.2, place p_1 has two tokens; all the other places do not have a marking. The *enabled* transitions in the current marking are highlighted with a red overlay – in the example, only transition, t_1, is enabled.

Note that, in the situation of Fig. 4.2, also transition t_6 at the bottom is highlighted with some light grey overlay and also the read arc to place $p7$ has a light grey overlay. This indicates that, when ignoring read and inhibitor arcs, transition t_6 would be enabled. We say that this transition is *weakly enabled*. By clicking on the read arc, the end-user could choose to ignore that arc and then fire the transition. We discuss that later.

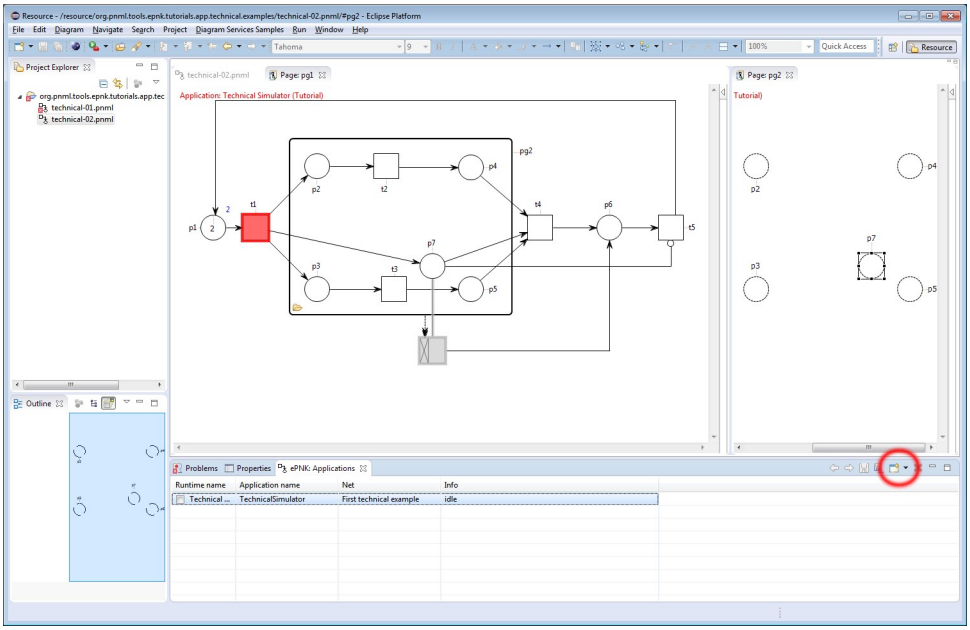

Figure 4.2: The example net with the example simulator started

When double-clicking on an *enabled transition*, the respective transition will *fire*, the *marking* of the places will change and the *enabled* transitions in the new marking will be highlighted. Figure 4.3 shows the situation after the end-user has fired (double-clicked on) the sequence of transitions t_1, t_1, t_2, t_3, and t_4. In that situation, places p_2, p_3, p_7 and p_6 have one token each (have *marking* 1), and transitions t_2, t_3, and t_6 are *enabled*. Transition t_5 is only *weakly enabled* since the inhibitor arc from place p_7 to transition t_5 prevents it from firing (place p_7 has a token).

In Fig. 4.3, you can also see another detail of the simulator: the current marking is not only shown as an annotation of the respective place. It is also shown as annotation of each reference place that *resolves to* the place – as you can see for the reference places shown on the right-hand side.

Figure 4.4 shows the situation after *firing* transition t_6. Since this transition has a reset arc from page pg_1 all tokens are removed from the respective places on that page. The only place that has tokens now is place p_5: it has two tokens now since transition t_6 added another token. Transition t_5 is enabled now, since there is no token on place p_7 anymore. Moreover, transition t_6 is weakly enabled.

Let us come back to the notion of a *weakly enabled* transition, which is a

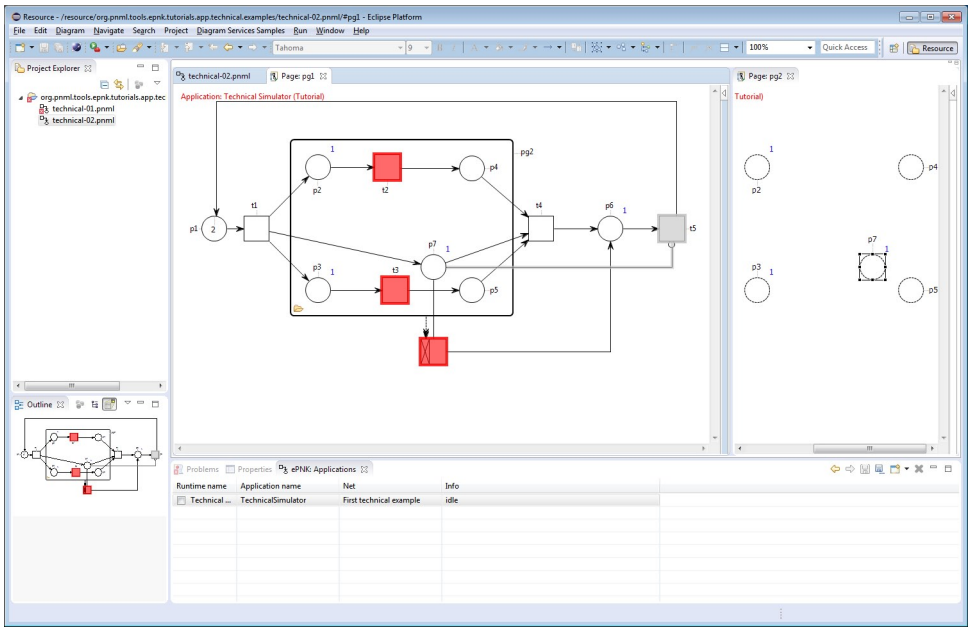

Figure 4.3: The simulator after firing transition t_1, t_1, t_2, t_3 and t_4

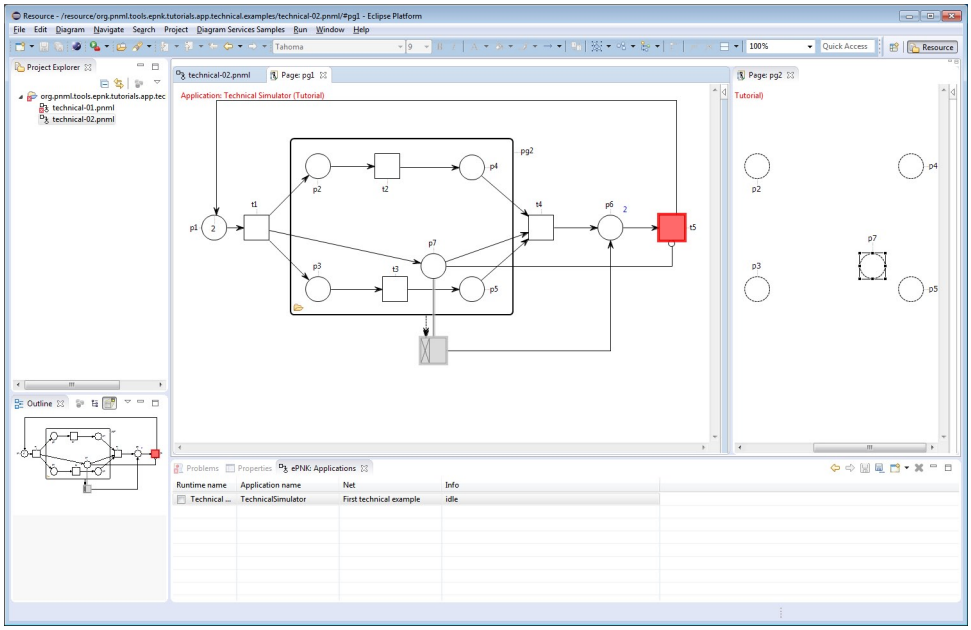

Figure 4.4: The simulator after additionally firing t_6

transition that would be enabled when ignoring conditions imposed by *read* arcs and *inhibitor* arcs. In our example in Fig. 4.4, the bottom transition t_6 is weakly enabled only since the place at the other end of the *read* arc, place p_7 does not have a token. This is indicated by the light-grey overlay of the transition and of the read arc.

The speciality of our technical net simulator is that the end-user can deactivate *read* arcs and *inhibitor* arcs, by clicking on them. Deactivated arcs will be shown with a red overlay. Once all *read* arcs and *inhibitor* preventing the enabledness of a transition are *deactivated*, the transition will be enabled, indicated by a red overlay again. Figure 4.5 shows the effect of the end-user clicking on the read arc in the situation of Fig. 4.4. Then, the end-user can double-click on it and fire it.

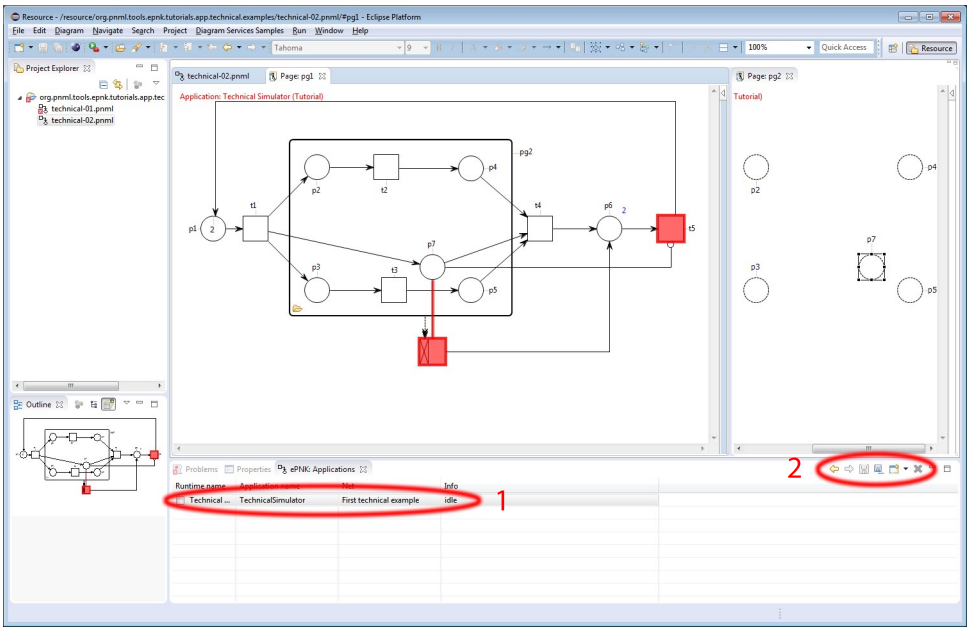

Figure 4.5: The simulator after deactivating the read arc

At last, let us have a look at some standard features of the ePNK and the application view. The application view shows a list of all the ePNK applications currently running in the ePNK on some net. In the situation shown in Fig. 4.5, only one application is running. The end-user can select one of the running applications or un-select all applications. Then, the visual feedback from the selected application is shown in the graphical editor of the respective net. The end-user can also delete (shut down) applications

by selecting the check boxes of the respective applications (see line marked by 1 in Fig. 4.5) and clicking on the delete button (which will turn red once at least one application is selected) in the ePNK applications view (see the part marked with 2 in Fig. 4.5).

Note that there are also some other buttons (see the part that is marked with 2 in Fig. 4.5). The *back* and *forward* buttons allow the end-user to go back and forth in the simulation, the disk buttons allows the end-user to save the current state the and firing sequence of the simulator to a file. Such a *saved state of an application* can be loaded again, when starting a new application with "Load application" in the start application drop down menu.

4.2 Conceptual steps

In this section, we discuss the conceptual steps for realizing the tool which we had discussed in Sect. 4.1. The first step is the definition of our *technical Petri net type*, which basically consists of a class diagram capturing the concepts of our *technical Petri net type* and some constraints. The second step is the definition of the graphical appearance of the features our *technical Petri net type* – in our case the arcs and the transitions. The last step is the definition of the *simulator application* for our *technical Petri nets*.

Note that, in this section, we do not discuss any technical details, how to create the models, how to generate the code, or how to plug in the extensions to the ePNK in order not to loose track of the overall picture. The technical details are discussed in Sect. 4.3.

4.2.1 Petri net type

In this section, we present a class diagram (actually an Ecore diagram), which reflects the extensions of our *technical Petri net type* as discussed in Sect. 4.1.1. Basically, the extension on top of the basic net elements, *places*, *transitions*, and *arcs*, are that arcs can be of kind *normal*, *read*, *inhibitor* and *reset*. In addition, places can have an *initial marking*, indicating how many tokens are on each place initially.

4.2.1.1 Petri net type definition

Figure 4.6 shows the class diagram with all the features of our *technical Petri net type*, which is called a *Petri net type definition* or *PNTD* for short. This diagram refers to some classes which are defined by the ePNK in the

PNML core model: these are the classes shown in magenta on the left and on the top of the diagram. The other classes shown in light cream are the definition of our *technical Petri net type*, which extend the concepts of the *PNML core model*.

There are two major extensions in our Petri net type definition in Fig. 4.6, the *arc* and *place*. Both extend the respective concept of the PNML core model. Arcs have a concept of *ArcTypeAttribute* with an attribute *text* of the enumeration type *ArcType*, which is also defined in this PNTD. *Places* have a *MarkingLabel* with an attribute *text* of the type *NonNegativeInteger*, which is a data type defined in the PNML core model.

The class *ArcTypeAttribute* extends the class *Attribute* from the PNML core model, and the *MarkingLabel* extends the class *Label* from the PNML core model. This defines how the ePNK should handle these additional features. A label will be graphically represented as an annotation to the respective element. In our example, the initial marking is shown as such a textual annotation (see Fig. 4.1) "2" inside the place, this label however could be freely moved by the end-user. The type attribute for places is not shown as an annotation of the arc; it is represented by the graphical representation of the respective arc. It can be edited by the end-user in the properties view, once the respective arc is selected.

Note that the classes *ArcTypeAttribute* and *MarkingLabel* are associated with the class *Arc* or *Place* with a composition. The name of this composition, will be the name of the respective feature and its cardinality says how many of these features each element can have. In our case, both cardinalities are [0..1], which means that the feature is optional and there can be at most one.

You might have realized the the enumeration *ArcType* does have two literals only: *READ* and *INHBITOR*. At a first glance, this might look awkward since in the discussion of our technical Petri net type we mentioned four different types: *normal*, *read*, *inhibitor* and *reset*. The reason is that the arc type feature is optional, and that we interpret a missing or not set arc type feature as *normal*. Likewise *reset* is the default (or actually only) interpretation for arcs running from a page to a transition. Therefore, it is enough that the enumeration *ArcType* represents *READ* and *INHBITOR*, which are the only ones the end-user needs to set explicitly.

Note that also the *marking* feature is optional. If the marking feature is not set, the default interpretation is 0.

In the diagram from Fig. 4.6, there are two additional classes defined: *Transition* and *TechnicalNetType*, both of which do not define any extensions on top of the PNML core model classes they inherit from. The class

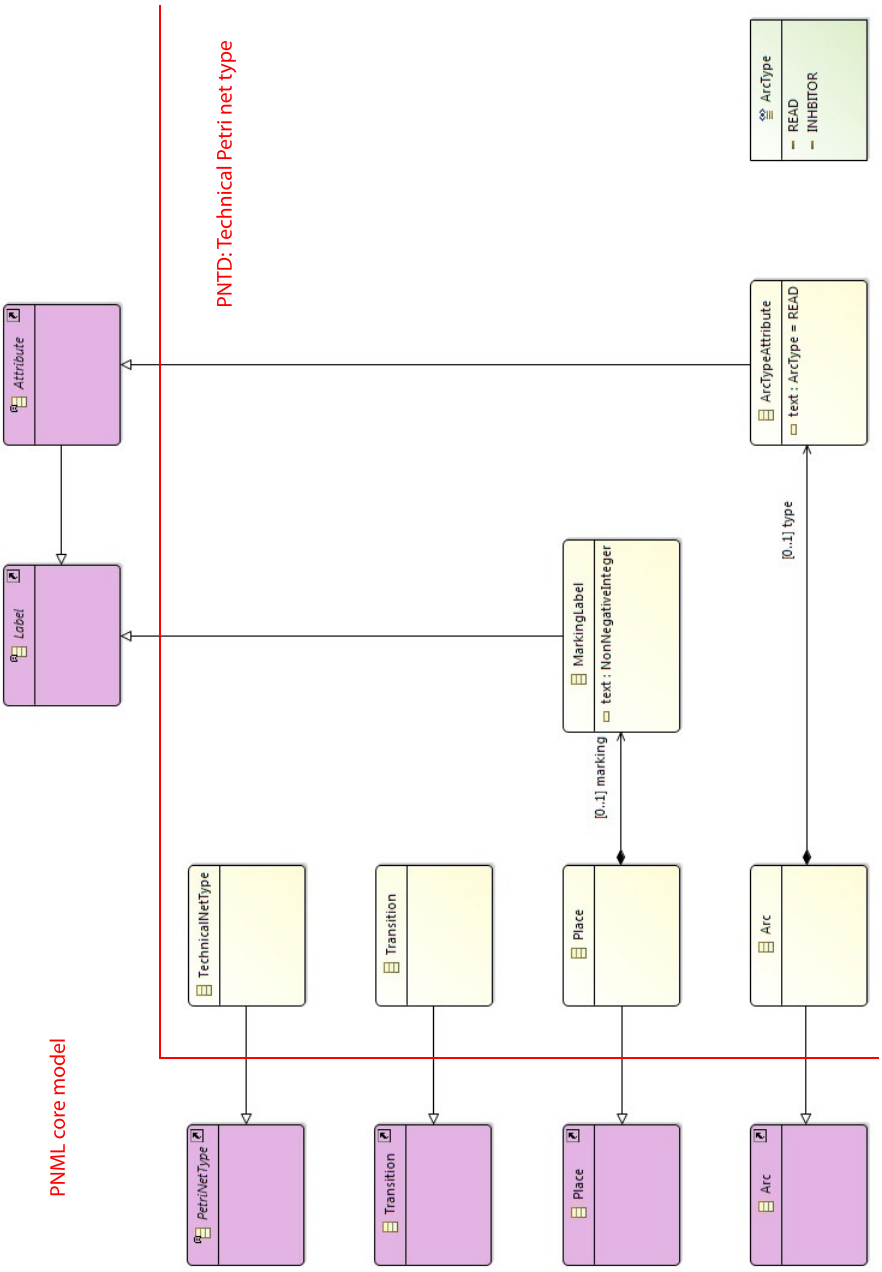

Figure 4.6: The Petri net type definition for the technical Petri net type

Transition would actually not be necessary, since in our technical net type transitions do not have any extensions. But introducing an explicit class for transitions in our PNTD allows us to explicitly to refer to the transitions of this type. The class *TechnicalNetType*, however, is necessary: it represents the net type we define here, and we will need it later to plug in the extension to the ePNK.

4.2.1.2 Constraints

The diagram of Fig. 4.6 defines the additional concepts of our technical Petri net type. But, is is not precise enough since it would allow arcs running from a transition to a place to be an inhibitor arc, which we do not want. Actually, without additional restrictions, arcs could run between all kinds of nodes, for example between two places or between two transitions; arcs would be even be allowed to run between two pages.

Therefore, we need to define an additional constraint that forbid arcs that our technical net type should not have. We define an OCL constraint for that purpose.

Listing 4.1: OCL constraint for arcs

```
context technical::Arc inv:
( self.source.oclIsKindOf(pnmlcoremodel::PlaceNode) and
  self.target.oclIsKindOf(pnmlcoremodel::TransitionNode) )
or
( self.source.oclIsKindOf(pnmlcoremodel::TransitionNode) and
  self.target.oclIsKindOf(pnmlcoremodel::PlaceNode) and
  self.type->size() = 0 )
or
( self.source.oclIsKindOf(pnmlcoremodel::Page) and
  self.target.oclIsKindOf(pnmlcoremodel::TransitionNode) and
  self.type->size() = 0 )
```

Listing 4.1 shows this OCL constraint for arcs: it states that an *Arc* of or our technical net type can have three different forms:

1. It can run from a place to a transition. Since there also can be arcs between reference nodes of that kind, the requirement is slightly generalized to *PlaceNode* and *TransitionNode*. In that case there is no restriction of the arc's type.

2. It can run from a transition to a place. For the same reasons as above, the requirement is slightly more general: the arc can run from a *TransitionNode* to a *PlaceNode*. For arcs from a transition to a place, however, the arc's type should not be set (i. e. it must be a *normal* arc).

3. It can run from a page to a transition, slightly more generalized to run from a *Page* to a *TransitionNode*. Also in that case, the arc's type should not be set. In this case, the arc is interpreted as a *reset* arc.

In order to demonstrate that constraints can also be defined in Java, we actually introduce our example has one additional constraint: there should not be duplicate read or inhibitor arcs between the same two nodes. The implementation of *Java constraints*, however, will first be discussed in the technical details section (see Sect. 4.3.3.2).

4.2.2 Graphics of the Petri net type definition

In Sect. 4.1.1, we have discussed already that some features of our technical Petri net type should be graphically visualized in a particular way. The type of an arc should be indicated by a dedicated graphics for *read*, *inhibitor* and *reset arcs* (see Fig. 4.1). It is actually quite typical that Petri net objects which have an attribute extension, also have a dedicated graphics for that type of Petri net object; the reason is that the value of an attribute is visible only in the properties view of the tool when the object is selected, but not in the graphical representation of the net itself, unless there is a dedicated graphical representation for it.

In our technical Petri net type, we have chosen to have a dedicated graphics for transitions, too: transitions with no ingoing normal arc or no outgoing should be indicated with a cross in the left or right third of the transition, respectively, as shown for transition t_6 in Fig. 4.1.

For each type of net object with a dedicated graphics, we need to implement a *Figure* class, which takes responsibility for the graphics of that element. Basically, there are two ways of implementing the such figures. The first one is to change the figure by changing its configuration and graphical attributes. In our example, we use that way for the graphical representation of arcs. The second one is to override how the figure is actually drawn. In our example, we use that way for the graphical representation of transitions.

Listing 4.2 shows the code snippet which configures how an arc is drawn, dependent on its type. Basically, there are four cases: if the arc is a *read* arc, both the source and the target decorator are set to `null`, so that the arc does not have any arrow head; if the arc is an *inhibitor* arc the source decorator is

set to `null` and the target decorator is set to a new *CircleDecoration* which is a class provided with the ePNK; for a *reset* arc, the line style is set to dashed, and the target decorator is set to a new *DoubleArrowHeadDecoration*; at last, for normal arcs the line style is solid, there is no source decorator, and the target decorator is a normal *ArrowHeadDecoration*, which is provided by the ePNK. If need should be, new decoration classes could be implemented for the graphical extension. But, this is not necessary in our tutorial example.

Note that Listing 4.2 gives just a glimpse of the respective class. We discuss more details on how to plug in this class and how and when to properly update the graphical representation of a net object in the technical details.

Listing 4.2: Arc graphics: defining appearance of acs

```
private void setGraphics() {
  if (arcType == ArcType.READ){
    this.setTargetDecoration(null);
    this.setSourceDecoration(null);
    this.setLineStyle(SWT.LINE_SOLID);
  } else if (arcType == ArcType.INHIBITOR){
    this.setTargetDecoration(new CircleDecoration());
    this.setSourceDecoration(null);
    this.setLineStyle(SWT.LINE_SOLID);
  } else if (arcType == ArcType.RESET) {
    this.setTargetDecoration(new DoubleArrowHeadDecoration());
    this.setSourceDecoration(null);
    this.setLineStyle(SWT.LINE_DASH);
  } else {
    // everything else is interpreted as NORMAL arc
    this.setTargetDecoration(new ArrowHeadDecoration());
    this.setSourceDecoration(null);
    this.setLineStyle(SWT.LINE_SOLID);
  }
}
```

Listing 4.3 shows the code snippet which takes care of drawing the graphical representation of a transition – overriding the *fillShape* method. On top of drawing the figure as usual by calling `super.fillShape(graphics)`, the two conditional statements are code for "drawing" the separator and a cross in the left respectively right third of the transition using the programming mechanisms of Eclipse Draw2D, if necessary. Also here, there are some more technical details on how to plugin the respective class to the ePNK and how

to update the graphics when needed. We discuss this in the technical details.

Listing 4.3: Transition graphics: drawing the figure

```
protected void fillShape(Graphics graphics) {
  super.fillShape(graphics);
  graphics.pushState();

  graphics.setLineWidth(1);
  Rectangle rectangle = this.getClientArea();
  int w = rectangle.width / 3;
  if (!this.hasNormalInArcs) {
    graphics.drawLine(rectangle.x + w, rectangle.y,
      rectangle.x + w, rectangle.y + rectangle.height-1);
    graphics.drawLine(rectangle.x + w, rectangle.y,
      rectangle.x, rectangle.y + rectangle.height-1);
    graphics.drawLine(rectangle.x, rectangle.y,
      rectangle.x + w, rectangle.y + rectangle.height-1);
  }

  if (!this.hasNormalOutArcs) {
    graphics.drawLine(rectangle.x + 2*w, rectangle.y,
      rectangle.x + 2*w, rectangle.y + rectangle.height-1);
    graphics.drawLine(rectangle.x + rectangle.width-1, rectangle.y,
      rectangle.x + 2*w, rectangle.y + rectangle.height-1);
    graphics.drawLine(rectangle.x + 2*w, rectangle.y,
      rectangle.x + rectangle.width-1,
      rectangle.y + rectangle.height-1);
  }

  graphics.popState();
}
```

4.2.3 The simulator application

At last, we need to define the simulator application, which allows the end-user to simulate the technical nets, and which visualizes the current state of the Petri net and the currently enabled transitions graphically on top of the net shown in the graphical editor.

4.2.3.1 Runtime annotations

In order to separate concepts from the actual graphical representation, the ePNK allows us to define *annotations* for nets, which reflect runtime information for our application. In our case, this would be the current state of the simulator, information on the enabled transitions and activated and deactivated arcs.

Like for the definition of a Petri net type, the conceptual annotations of an application can be defined by a class diagram, which extends some basic annotation concepts provided by the ePNK. Figure 4.7 shows this model, where the two classes shown on the top and in magenta represent the based concepts from the ePNK: *object annotations* is any form of annotation of any Petri net object; *textual annotations* is information that typically is represented by a text instead of a graphical overlay.

The classes represented in light cream are the *annotations* needed for our simulator. *EnabledTransition* is an annotation for weakly enabled transition – the value of attribute *enabled* indicates whether the transition is *truly* enabled. *Marking* represents the current marking of a place (i.e. the number of tokens on that place). The actual value is represented by its attribute *value*. Since the value should be shown as textual annotation, this class extends not only the *ObjectAnnotation* but also *TextualAnnotation* of the ePNK.

At last *InvolvedArc* is an annotation for arcs, which indicate arcs of a weakly enabled transition that prevent the enabledness of this transition. This way, the end-user can toggle whether the arc is *active* or not. The status of this user's selection is represented by attribute *active*.

In order to keep track of the arcs involved in an enabled transition, there are two compositions from *EnabledTransition* to *InvolvedArc*: one for the transition's incoming arcs and one for the outgoing arcs (the outgoing arcs are actually not used in our example). This way, it is possible to navigate between enabled transition annotations and the involved arc annotations.

Note also that there is a bidirectional relationship between the *EnabledTransition* itself. The reason is that a single transition can have many reference transitions referring to it. In our simulator, also the reference transitions that resolve to an enable transition should be marked as enabled. The reference *resolve* represents the transition, the reference *ref* represents all the reference transitions. We will see later in more detail how our simulator uses this information.

An *PNK application*, basically, consists of a list of *net annotations*, each of which which in turn consists of some *object annotations*. Moreover, there

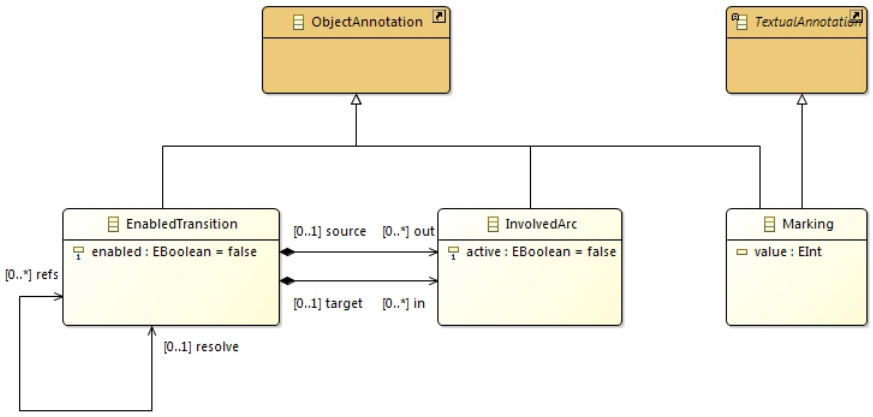

Figure 4.7: Simulator: annotations model

always is a *current* net annotation, which for example represents the current state of the simulator. In our example, a net annotation represents the current markings of the places plus the enabled transitions and the involved arcs and their state.

In order to visualize the current state of an ePNK application in the graphical editor, an ePNK application uses one or more *presentation handlers*. A *presentation handler* tells, for each annotation, how it should be visualized in the net. In order for end-user to be able to interact with the simulator via the annotations, an ePNK application uses one ore more *action handlers*. An *action handler* defines how the simulator should react to a user interaction (mouse press, mouse release, and mouse double click). This could for example, be firing a transition by updating the net annotations, or toggling the active status of an involved arc. We discuss more details in the technical parts of this tutorial.

Here we briefly discuss the main ingredients and functionality that needs to be implemented for our simulator (and actually most kinds of simulators):

1. We need some data structure or dedicated class, which represents a *marking of the net*, which basically is a mapping from places to integers.

2. We need a method, which computes the *initial marking of the net* from a net model.

3. We need methods, which, for a given marking and some transition,

computes whether the transition is *enabled*, and which are the *involved arcs*.

4. We need a method which for a given marking and a given enabled transition computes the marking which is the result after *firing the transition in the given marking*.

5. We need a method, which from a marking *computes a net annotation*. representing the marking of the places as well as the enabled transitions, and the involved arcs.

We discuss in the technical steps how the above methods can be combined into a complete simulator using also the presentation handlers and action handlers.

4.3 Technical steps

In Section 4.2, we have discussed the major conceptual steps for realizing a Petri net type definition (PNTD) for our technical net type and simulator for them based on the ePNK.

In the following sections, we discuss all the technical details, some subtle issues concerning tooling, and possible problems that might occur on the way, and how to deal with them. We start with some installation information in Sect. 4.3.1 and how to install the project implementing the extensions discussed in this tutorial in your Eclipse workspace. Then, we go through the steps which we discussed in Sect. 4.2 in some more detail, discussing the models and the source code of the Eclipse projects.

Note that the tricky issues and problems are sometimes not in the resulting projects, models or code, but in how to actually create, configure, and manipulate them. Therefore, we also discuss how to actually create and edit some of the artifacts in the Eclipse IDE (using the Eclipse package which is pre-configured for working with models: *Eclipse Modeling Tools*).

4.3.1 Installation

We begin with discussing how to install Eclipse, the ePNK and the source code for the technical example, and how to start and use it.

4.3.1.1 Eclipse

Before you can work with the ePNK, you need to install Java and Eclipse on your computer. When using the ePNK as a developer and not only as

an end-user, it is recommended that you install the *Eclipse Modeling Tools* package of Eclipse. The ePNK was testes with Eclipse Mars and you can find and download the Mars Eclipse Modeling Tools package from the Eclipse Modeling Tools web site for Mars: `http://www.eclipse.org/downloads/packages/eclipse-modeling-tools/mars2`. You might also use the latest version of Eclipse Modeling Tools package of the latest version of Eclipse (currently Neon), but the ePNK has not yet been tested under Eclipse Neon yet.

In this tutorial, we also use OCL constraints. Therefore, it is recommended that you install the "OCL Examples and Editors SDK" feature. You can do that by starting your new version of Eclipse after you have installed it and by selecting "Install New Software..." in the "Help" menu. In the opened "Install" dialog, you should choose the default update site in the "Work with" field[3] and type "OCL" in the field below. After a while, the feature "OCL Examples and Editors SDK" should show up as an available feature in category "Modeling". Select this feature and follow through the installation process and restart Eclipse.

4.3.1.2 ePNK

After you have installed and started Eclipse, you need to install the ePNK version 1.1 from the ePNK update site at `http://www2.compute.dtu.dk/~ekki/projects/ePNK/1.1/update/`. You can do that as discussed before by selecting "Install New Software..." in the "Help" menu; then, select `http://www2.compute.dtu.dk/~ekki/projects/ePNK/1.1/update/` as update site and install all features of the ePNK. Make sure that you select the newest features of the ePNK – `org.pnml.tools.epnk.core` should have version number 1.1.2 or higher.

Make sure that, in the "Install" dialog, the checkbox "Contact all update sites during install to find required software" is checked. This makes sure that all the extensions that the ePNK needs to run will be installed too. Then, follow through the installation process. Note that you might be asked to confirm the installation of unsigned features.

4.3.1.3 Import of example projects

The Eclipse projects discussed in this tutorial are also available online. You can download them as exported Eclipse projects from the ePNK up-

[3]For Eclipse Mars, that update site would be `http://www.eclipse.org/downloads/releases/mars`.

date site: `http://www2.compute.dtu.dk/~ekki/projects/ePNK/1.1/tutorials/ePNK-app-tutorial.zip`. In order to install these projects into your workspace, download the above file, and save it somewhere on your computer. Then, right-click in the project explorer of your Eclipse workspace, and select "Import..."; in the opened "Import" dialog select "Existing Projects into Workspace" from category "General"; in the subsequent dialog, select "Select archive file" and, by using the "Browse..." button, navigate to the file you had downloaded before. Once you have chosen this file, you should see four Eclipse projects; select all of them and "Finish" the import.

After that, you should see the four Eclipse projects in your Eclipse workspace, and after building the workspace, there should not be any errors shown in your workspace. In order to test whether these projects where correctly installed and built, you should start another instance of your Eclipse (the so-called *runtime workbench* as opposed to the Eclipse which you have started already, which is called *development workbench*).

The first time you start up a new *runtime workbench*, you need to create a new *run configuration* in your development workbench. To this end, choose the "Run" symbol in the toolbar and select "Run configuration", which starts a "Create, manage and run configurations" dialog; in this dialog, choose "Eclipse Application" and click on "New launch configuration", enter a name for this configuration (e. g. ePNK Tutorial), and then click on "Apply" or "Run". After some time, a new instance of Eclipse should start up (the *runtime workbench*) – in this instance, the Eclipse tutorial projects that you just have imported to your workspace are installed and running.

In order to test whether the installed projects are working fine, you should import a project with a Petri net example – actually it is the one shown in Fig. 4.1. You can obtain this project from `http://www2.compute.dtu.dk/~ekki/projects/ePNK/1.1/tutorials/ePNK-app-tutorial-example.zip`. Import it to the workspace of your Eclipse runtime workbench as discussed before. Once you have imported it, open the *PNML document* `technical-02.pnml` by double-clicking on it[4]. Then open the elements until you reach a page and double click on the page. On this page, a graphical editor should start up – after a while. Once this editor opened, open the "ePNK: Applications" view (using the "Show view" menu in the "Windows" menu.) In the "ePNK: Applications" view, use the start application drop down menu (see red circle in Fig. 4.2) and select "Technical

[4]If double-clicking does not work, right-click on it and select "Open with..." and then selecting "PNML Editor" from "Others..."

Simulator (Tutorial)", which should start up as shown in Fig. 4.2 with transition t_1 marked as enabled. You can now play around with firing transitions by double-clicking on them as discussed in Sect. 4.1.2. Note that you can also interact with the arcs which are marked grey or red in order ot toggle their activation status.

Once this works, you can shut down the simulator application, by selecting the checkbox in front of it and then pressing the "delete" button.

If this works, the example projects are properly installed and you can continue with the next technical step of the tutorial in Sect. 4.3.2. For now, you can shut down (close) your runtime workbench (the one with the example of the technical Petri net, which you had simulated).

Note that when you want to start up a runtime workbench the next time, you do not need to create another *run configuration*. Just click on the "Run as.." (launch) button next time you want to start a runtime workbench.

In the following sections, we discuss the four projects which implement our *technical Petri net type*, its graphical representation, and the simulator in more detail. Moreover, we discuss how you would create these projects and some of the artifacts from scratch in the Eclipse IDE.

Note that all the example code discussed in this tutorial is taken from these projects. Sometimes, we deletes import statements, comments or compacted the code a bit for the discussion. You will find the source code for all listings discussed in this tutorial in these projects for more details.

4.3.2 PNTD

We start with discussing the project `org.pnml.tools.epnk.tutorials.app.pntd`, which defines the *Petri net type definition* for our *technical net type*, which we had conceptually discussed in Sect. 4.2.1.1.

4.3.2.1 Ecore model

The class diagram defining the concepts of our *technical net type*, which we had discussed in Sect. 4.2.1.1 already, can be found in the file `technical.ecore` in folder `model` of *EMF project* `org.pnml.tools.epnk.tutorials.app.pntd`. It is actually an Ecore model, which is EMF's "lightweight version" of class diagrams. You can inspect it with the *Ecore Model Editor*, which is a simple tree editor. Typically, you can do this by double-click on this file in the Package explorer of the Eclipse development workspace.

Figure 4.8 shows the Eclipse development workspace with the Ecore model for the technical net type opened in the tree editor. You can also see

that this model is actually referring to elements of the *PNML Core Model* which it extends; parts of this model can be seen at the bottom of the editor. You can also see the attributes of the defined package in the properties view: its name "technical", its unique URI, which we have chosen for this package "http://epnk.tools.org/tutorials/app/technical", and its name space prefix "tech".

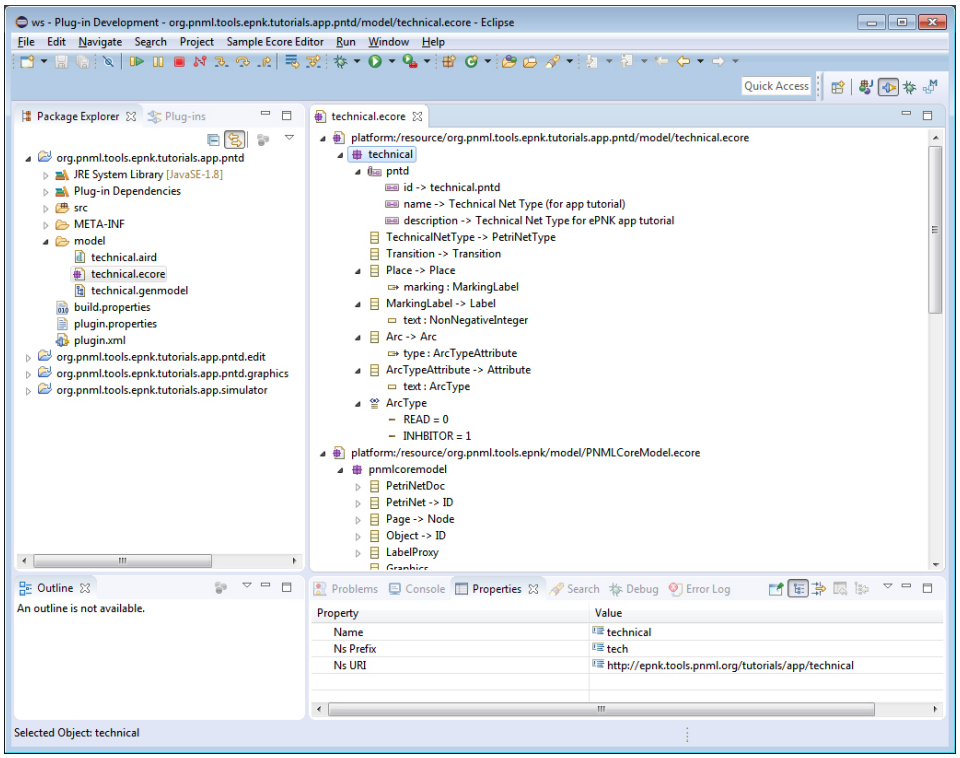

Figure 4.8: Ecore model of PNTD for technical net type in tree editor

If you want, you can also open this model in a graphical editor. To do this, you would need to switch to the "Modeling" perspective of your Eclipse workspace, and open the file `technical.aird` by double clicking on it; then navigate to "Representation per category" → "Design" → "Entities" → "technical class diagram" and double click on → "technical class diagram". Figure 4.9 shows the model when opened in the graphical editor.

We do not discuss the concepts of this model here again, since we have discussed them already in Sect. 4.2.1.1 in Fig. 4.6.

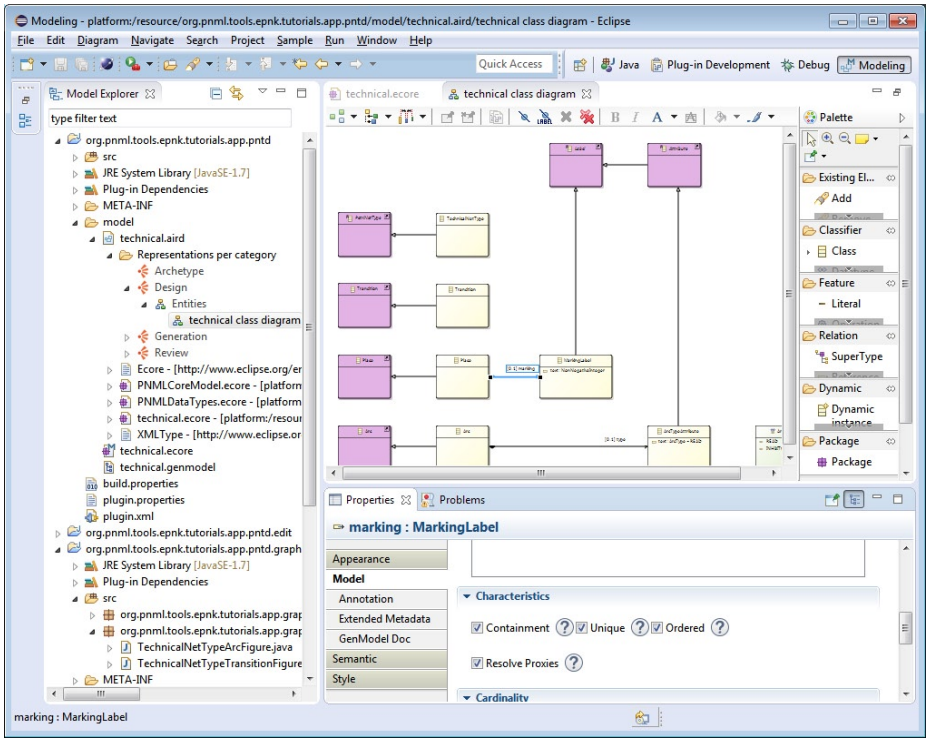

Figure 4.9: Ecore model for technical net type as diagram

4.3.2.2 Creating the EMF project and model

Instead, we briefly discuss how to create such an EMF project and a model as well as some technical details on how to edit some model features in this subsection. Before you do this, it might be a good idea to switch back to the "Plug-in Development" perspective in your Eclipse workspace.

A project which is based on a model from which code can be generated automatically is called an *EMF project*. You can create a new empty EMF project via the "New" action in the "File" menu of Eclipse. Select "Other...", and then, in the opened "New" dialog, select "Empty EMF project" from category "Eclipse Modeling Framework".

In the newly created EMF project, you will find a folder `model`, which is where you should create a new Ecore model. A new Ecore model can be created with the "New" action in the "File" menu. You will find the option "Ecore model" in the "Eclipse Modeling Framework" category (try to use a better name than "My.ecore"); in the "New" dialog, you will be asked for the top-level model object and the XML Encoding. The default should be fine; make sure that the model object is set to `EPackage`.

After you have created a new model, it will be shown in the Ecore tree editor with a package with an empty name. Give a reasonable name to the package, which uses non-capital letters, and choose a name space prefix and a unique URI representing your new package. The example from Fig. 4.8 might give you an idea on what to choose. Note however, that the domain `http:\\epnk.tools.pnml.org` is reserved for the ePNK itself and for using sub-domains of `tools.pnml.org` you would need to ask permission from `http:\\www.pnml.org`.

In the newly created package, you should add one class which represents the newly defined Petri net type. You can do that by right-clicking on the package and then selecting "New Child" → "EClass" and give it a reasonable name – it is called `TechnicalNetType` in our example. This class must extend the class `PetriNetType` from the `pnmlcoremodel`. To this end, you would need to set the class's "ESuper Types" in the properties view. Before you can chose a class from the *PNMLCoreModel*, you need to make this package know to the editor. To do this, you need to load the resource containing the `pnmlcoremodel` in the tree editor. You can do this by right-clicking somewhere in the tree editor and selecting "Load Resource..."; then, in the "Load Resource" dialog, select "Browse Target Platform Packages..." and select `http://org.pnml.tools/epnk/pnmlcoremodel`. Once you have done that, you can select `PetriNetType` as the "ESuper Types" in the properties view for your new class for your Petri net type. Make sure to save

this file right away.

Now you can continue adding the concepts of your new Petri net type. You should add a class for each Petri net concepts that you want to extend and make sure that it inherits from the corresponding class of the `pnmlcoremodel`. The name of these classes can, in principle, be any legal class name. But, you save a lot of programming, if the class names are the same as in the `pnmlcoremodel` – don't worry, since you have created them in a new package, there will be no confusion with names, since your new classes live in a different name space. All new features of your net type are represented by compositions to other classes, which need to inherit from either `Label` or `Attribute` of the `pnmlcoremodel`. And these classes must have an attribute with name `text` and some data type. The data type can either be an existing data type, which is built in to Ecore or defined in the ePNK, or a datatype defined in the new package. In our example, the datatype `ArcType` is an enumeration defined in the package itself, the datatype `NonNegativeInteger` is defined by the ePNK.

Note that in our example, the cardinality from the Petri net objects to its features is [0..1], which means that there can be zero or one of each feature for every object of the respective type. But, the cardinality could also be $[0..*]$ allowing arbitrarily many instances of this feature for a single element. In the tree editor for Ecore models, the value -1 as upper bound represents "arbitrarily many".

Note also that the reference from the Petri net object to its feature must be compositions. In the Ecore model, this means that the property `Containment` of the respective reference must be set to `true`.

At some point, it might be easier to edit and extend the Ecore model by using a graphical editor. To this end, you can create a diagram for an existing Ecore model: right-click on the file of the Ecore model and select "Initialize Ecore Diagram ..."; then, in the "Create Representation File" dialog, select a file name and a folder for the diagram file (it should typically be in the same folder as the model). In the "Create Representation Wizard", which opens after a while, select "Entities" in category "Design" and continue; in the next dialog select your package and give the diagram some name. In the diagram editor that opens, double-click on "here" to create an initial representation of you model. If you want to see related elements from other packages in this diagram you can right-click on the diagram and select "Add Related Elements". You will probably need to arrange the elements in a nicer way. In the end, don't forget to save the diagram. Note that for opening the diagram again, you will need to switch

to the "Modeling" perspective of your Eclipse workspace as discussed at the end of Sect. 4.3.2.1 for Fig. 4.9 already.

4.3.2.3 Code generation

In order to plug in the PNTD defined by the Ecore model to the ePNK, we need to first generate code from this model and make some adjustments to the code. The code will be generated from a so-called *gen model* that is associated with the Ecore mode and defines some additional information for the code generator on how the code should be generated and where the generated code should go.

In our example, the *gen model* is the file `technical.genmodel`. Once opened in the *EMF Generator* model editor, you can generate the code by right-clicking on the top-level element and selecting "Generate Model Code" and "Generated Edit Code". You can also generate the other code, but you do not need the "Test Code" and the "Editor Code" when using the model as a PNTD for the ePNK.

When you have created a new EMF project and a new Ecore model, however, there is no gen model yet. You first need to create the gen model for your new model file. You can do this by right-clicking on the file with the Ecore model and, then, selecting "New" → "Other..." and chosing "EMF Generator Model" from category "Eclipse Modeling Framework" and following through the dialog. When asked for the folder, make sure the gen model is in the same folder as the model; when asked for the model URI, you will need to click on "Load"[5] and continue. At some point, a dialog for selecting the packages for which the gen model should be generated will show up, which looks like the one shown in Fig. 4.10. Note that it crucial, that in this dialog you only select your package as "Root package"; all the other ones, you should select in "Referenced generator models". Before you continue, your selection in this dialog should look like shown in Fig. 4.11. Once you made these selections, you can "Finish" the generation of the gen model.

Then the gen model for your model file should be created, and it should be open in the "Gen Model" editor. Before you generate code from this new gen model, you need to make two manual changes in this gen model. First, select the top-most element and change one property in the properties view: change the property `Operation Reflection` in the category `Model`

[5] If clicking on load results in an error, there is probably some error in you Ecore model. Try to validate the Ecore model first and fix the error.

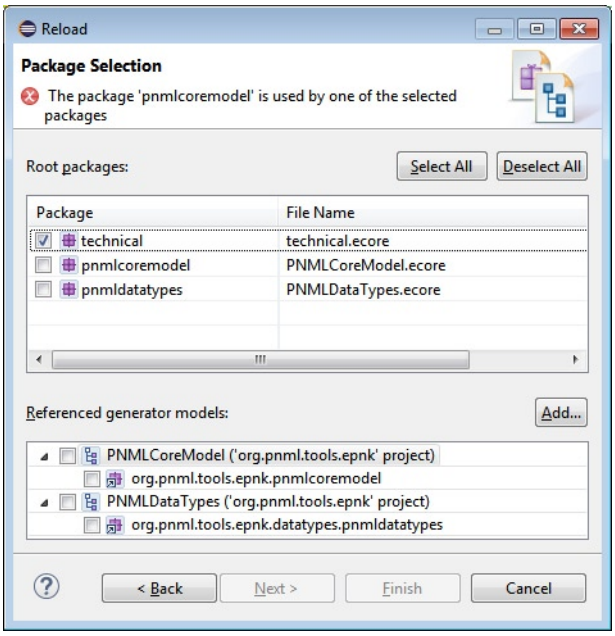

Figure 4.10: Select Package Dialog

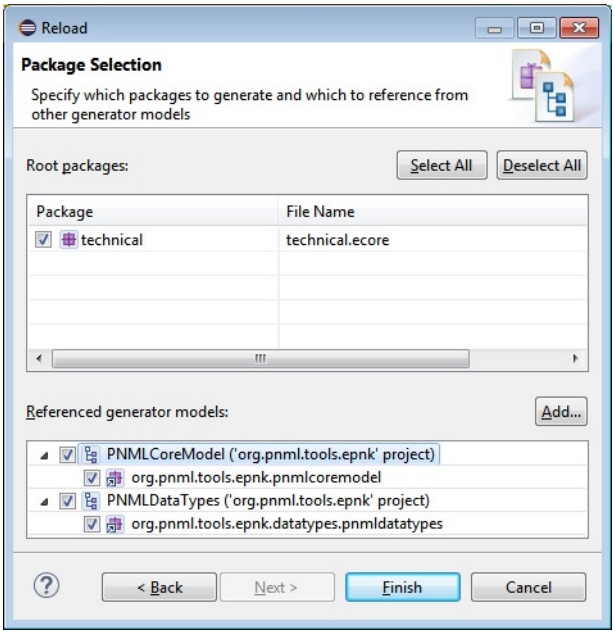

Figure 4.11: Select Package Dialog

to `false`. Second, open the top-level element and select the package element. For this package, set the property `Base Package` to some reasonable Java package name; in our example, it is set to `org.pnml.tools.epnk.tutorials.app`; this setting directs the generator to generate the code in a subpackage of this Java package.

Now, you can generate the model and the edit code from this gen model as discussed above. The generated code should show no errors; if it does, you probably forgot to change the property `Operation Reflection` to `false` in the gen model after generating it.

Note that, if you should make changes in the model later on, you would need to "Reload" your gen model again, so that it can become aware of the model changes and update accordingly (reusing as much as possible from the existing settings). In order to do that, first make sure that you have saved and closed your model file. Then, right-click on the gen model and select "Reload" and follow through the dialog which very much works like the dialog for initially creating the gen model.

4.3.2.4 Manual changes in the code

Before plugging in the generated net class to the ePNK, we need to make two manual changes in the Java class generated for the class `TechnicalNetType`, which represents the new Petri net type defined in this tutorial. In our example, this class can be found in the `src` folder in package `org.pnml.tools.epnk.tutorials.app.technical.impl` and is called `TechnicalNetTypeImpl`.

The two changes that need to be made in this class are the following: first, the constructor of this class must be made public, which is necessary so that the ePNK will be able to create new instances of this class. Second, the `toString()` method of this class must be implemented; it should return a URI, which uniquely identifies Petri nets of this type in PNML. It must be a string representing a URI. In principle, it can be any URI, you just need to make sure that it is unique.

Listing 4.4 shows the class `TechnicalNetTypeImpl` with the changes made in lines 15–19 and 26–37, highlighted in red. Note that for readability reasons, we have omitted some comments, but otherwise the class is completely shown. In addition to making the constructor public and implementing the `toString()` method, the changes are marked with the tag `generated NOT`, which indicates that there are manual changes to the generated code, and this way prevents the manual changes being overwritten the next time the code is generated again from the gen model.

Listing 4.4: Manual changes in class `TechnicalNetTypeImpl`

```
package org.pnml.tools.epnk.tutorials.app.technical.impl;

import org.eclipse.emf.ecore.EClass;
import org.pnml.tools.epnk.pnmlcoremodel.impl.PetriNetTypeImpl;
import org.pnml.tools.epnk.tutorials.app.technical.TechnicalNetType;
import org.pnml.tools.epnk.tutorials.app.technical.TechnicalPackage;

/**
 *
 * @generated
 */
public class TechnicalNetTypeImpl
             extends PetriNetTypeImpl implements TechnicalNetType {

  /**
   * @generated NOT (made constructor public)
   * @author ekki@dtu.dk
   */
  public TechnicalNetTypeImpl() {
    super();
  }

  @Override
  protected EClass eStaticClass() {
    return TechnicalPackage.Literals.TECHNICAL_NET_TYPE;
  }

  /**
   * The URI of the net type.
   *
   * @generated NOT (needs to return unique URI of this net type)
   * @author ekki@dtu.dk
   */
  @Override
  public String toString() {
    return "http://epnk.tools.pnml.org/tutorials/app/technical";
  }

}
```

The above changes are the only code that needs to be written manually for making the new Petri net type work with the ePNK. In our example, we have three additional Java classes, which are written completely manually. One implements a Java constraint, which is discussed in Sect. 4.3.3. The other two are convenience classes, which make it easier to implement our simulator and constraints later. Since it is good practice and increases maintainability to have such convenience classes with static methods, we briefly discuss them here, even though we do not need them right away.

Listing 4.5: Enumeration with all `ArcType`s

```
1  package org.pnml.tools.epnk.tutorials.app.technical.helpers;

   /**
    * [...]
    *
6   * @author ekki@dtu.dk
    * @generated NOT
    *
    */
   public enum ArcType {
11
     NORMAL, READ, RESET, INHIBITOR

   }
```

In Listing 4.5, you can see a Java enumeration for arc types. Remember, that our model of the PNTD has a similar type; but in the model, there have been two possible values only: `READ` and `INHIBITOR`. The end-user will only be able to set the type attribute of arcs to these two values – and the value can be left undefined. In Listing 4.5, the enumeration defines the values of all possible *interpretations* of arc types. The static method `getArcType()` shown in Listing 4.6 shows, how to compute the "interpretation" from the actual value set for the arc and its source and target nodes it is connected to. It is crucial to implement such an "interpretation" only once in the manually written code; otherwise this code would be repeated and scattered all over the project – possibly even using different interpretations in different parts of the software – making maintenance a nightmare.

Note that the code from Listing 4.5 and 4.6 is written completely manually, and does not run the risk of being overwritten by the code generator.

Listing 4.6: Class `TechnicalNetTypeFunctions` with static helper methods

```
1  package org.pnml.tools.epnk.tutorials.app.technical.helpers;
   // [...]

   /**
    * @author ekki@dtu.dk
6   * @generated NOT
    */
   public class TechnicalNetTypeFunctions {

     public static ArcType getArcType(Arc arc) {
11     if (arc instanceof
           org.pnml.tools.epnk.tutorials.app.technical.Arc) {
         org.pnml.tools.epnk.tutorials.app.technical.Arc tArc =
             (org.pnml.tools.epnk.tutorials.app.technical.Arc) arc;
         Node source = arc.getSource();
16       Node target = arc.getTarget();
         ArcTypeAttribute type = tArc.getType();

         if (source instanceof Page &&
           target instanceof TransitionNode) {
21         return ArcType.RESET;
         }
         if (type != null) {
           if (source instanceof PlaceNode ||
             source instanceof TransitionNode) {
26           switch (type.getText()) {
             case READ: return ArcType.READ;
             case INHBITOR: return ArcType.INHIBITOR;
             default: return ArcType.NORMAL;
             }
31         }
         } else {
           if (source instanceof PlaceNode ||
             source instanceof TransitionNode) {
             return ArcType.NORMAL;
36     } } }
       return null;
     }

     // [...] other helper methods omitted
41 }
```

Since the code is part of an EMF project, where most code is automatically generated, the code is tagged with `generated NOT` anyway. This makes it easy to search for all code which is not generated. In the project for the PNTD, there are actually five manual changes: two of them in the class for the net type and three manually written classes (the third class is discussed later in Sect. 4.3.3).

Since we might want to use the class `TechnicalNetTypeFunctions` in our other projects later, we need to export the Java package containing it from this project. You can do that by using the "Plug-in manifest" editor by double-clicking on the `plugin.xml` and selecting the "Runtime" tab. In that tab, you should add the respective Java package to "Exported Packages" by pressing the "Add..." button.

4.3.2.5 Plugging in the PNTD

The last step of defining a PNTD for the ePNK is actually plugging the generated and manually changed class for the net type, `TechnicalNetTypeImpl` in our example, in to the ePNK. This could be done by using the Eclipse "Plug-in manifest" editor. But, it is actually easier to do that directly by changing the XML code of the `plugin.xml`. To this end, open the `plugin.xml` file with the 'Plug-in manifest" editor by double-clicking on it and go to the tab called "plugin.xml".

Listing 4.7 shows the snipped from the `plugin.xml`, which plugs the PNTD in to the ePNK. It is a usual Eclipse *extension* referring to the *extension point* `org.pnml.tools.epnk.pntd` (line 4), which is defined by the ePNK. The attributes `id` and `name` is just a unique id and name for this new Petri net type. The actual new type is defined in the `type` element in lines 5–7; its attribute `class` refers to our class `TechnicalNetTypeImpl`, which we had modified manually earlier and which defines the new Petri net type. We need to refer to this class by its fully qualified Java class name (including the packages); the description is just some text describing the new Petri net type.

Note that at this point, with only the project `org.pnml.tools.epnk.tutorials.app.pntd` and the automatically generated project `org.pnml.tools.epnk.tutorials.app.pntd.edit`, we can use our Technical Petri net type with the ePNK already. The only problem would be that *read*, *inhibitor* and *reset* arcs would not be shown with a dedicated graphics. All arcs would be graphically shown as *normal* arcs. Moreover, the end-user would still be able to draw arcs between arbitrary nodes, even between pages. We discuss how to fix the latter problem in the next section, and

Listing 4.7: `plugin.xml` snipped plugging the PNTD in to the ePNK

```
<extension
  id="org.pnml.tools.epnk.tutorials.app.pntd"
  name="Technical Net Type (for app tutorial)"
  point="org.pnml.tools.epnk.pntd">
  <type
    class="org.pnml.tools.epnk.tutorials.app.technical.impl.
TechnicalNetTypeImpl"
    description="Technical Net Type for ePNK app tutorial" />
</extension>
```

how to fix the first problem in Sect. 4.3.4.

Whenever you create a new Petri net type, it would be a good idea to check whether your PNTD works with the ePNK at this point. Only after that, you should proceed.

4.3.3 Constraints

In this section, we discuss how to define and plugin constraints for a net type, so that the ePNK (and Eclipse in general) will take them into account.

As discussed in Sect. 4.2.1.1, we have two constraints for our *technical Petri net type*. The first is that an arc should run from a place to a transition or the other way round, or it should run from a page to a transition; moreover, only an arc running from a place to a transition can have a type (for the other arcs, the type attribute should not be set). In Listing 4.1, we have already seen an OCL formulation of that constraint. The other constraint was that there should be no duplicate read or inhibitor arcs between a place and a transition. This constraint is realized as a Java constraint.

All constraints are defined in the our PNTD project `org.pnml.tools.epnk.tutorials.app.pntd`, which defined the PNTD. The reason is that constraints conceptually are a part of a model. Actually, it would be possible to include the constraints to the Ecore model. But, we follow a slightly here.

4.3.3.1 OCL constraint

We start with discussing how to add the OCL constraint for properly connecting arcs to the project. This can be done by pluging in an OCL constraint by defining it in the `plugin.xml` of the project `org.pnml.tools.epnk.tutorials.app.pntd`. Listing 4.8 shows the part of `plugin.xml` that

defines a constraint provider, with the OCL constrain for arcs. All of that snippet is necessary, but we highlight some more important settings and features in the definition (marked in red).

The most important part is the actual OCL constraint, which we had shown in Listing 4.1 on page 168 already. The OCL constraint Listing 4.8 is shown as XML CDATA in order to escape all the special symbols of OCL in lines 30-39. Note, however that the first line from Listing 4.1, which gave the context is missing here. This context is actually defined by the target element in lines 21–28: the `class` attribute defines the Ecore class it refers to by the name `Arc` of the class followed by the package URI of the model package in which it is defined. The target also defines *events* which cause the validator to check the constraint again. In our example, these are set events of the features `source`, `target` and `type` of the arcs; this means that the validator kicks is whenever one of these features is of an arc changes. This goes together with the fact that we define the constraint as a *live constraint* (see line 10), which means that after any change an end-user makes (with respect to the defined events), the constraint is checked. If the constraint should fail, the complete action of the end-user will be undone. Therefore, the end-user cannot create a model that is invalid with respect to a *live constraint*, provided that the event definition covers all features and events that might invalidate a constraint of an arc. In our case, these features are `source`, `target` and `type`.

In addition, there is a *severity* of the constraint (line 12), which is an *error* in our example, and *language* of the constraint (line 9) is defined as OCL. Moreover, there is a message, which will be output to the end-user, whenever the validation fails – and the message of the problem marker attached to the model element. The tag {0} in this message, will be replaced with the object on which the constraint failed (the *target*).

Actually within the same constraint provider, more than one constraint can be defined, which is indicated by the ... in line 42. We will actually see an additional constraint there when discussing the Java constraint in the following section.

Note that the OCL constraint must be syntactically correct, and getting the syntax of OCL right might be a bit tricky if you do not have much experience. Experimenting with the OCL syntax by changing the OCL constraint in the `plugin.xml`, starting the runtime workbench and testing whether the OCL constraint works, and staring all over again, if it does not work, is way too time consuming. We need a more efficient way to get the OCL syntax right – and even a way to check how an OCL expression evaluates in a given

Listing 4.8: `plugin.xml` defining the OCL constraint for arcs

```
<extension point="org.eclipse.emf.validation.constraintProviders">
   <constraintProvider cache="true">
      <package namespaceUri=
            "http://epnk.tools.pnml.org/tutorials/app/technical"/>
      <constraints categories="org.pnml.tools.epnk.validation">
         <constraint
               id="org.pnml.tools.epnk.tutorials.app.pntd.
validation.correct-arc-connection"
               lang="OCL" mode="Live"
               name="Arc connection constraint for Technical Net"
               severity="ERROR" statusCode="401">
            <message>
                The arc {0} with this arc type is not allowed.
            </message>
            <description>
               Arcs must be between a place and a transition, a
               transition and a place, or between two transitions.
            </description>
            <target class="Arc">
               <event name="Set">
                  <feature name="source"/>
                  <feature name="target"/>
                  <feature name="type"/>
               </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) and
           self.type->size() = 0 )
         or
         ( self.source.oclIsKindOf(pnmlcoremodel::Page) and
           self.target.oclIsKindOf(pnmlcoremodel::TransitionNode) and
           self.type->size() = 0 )  ]]>
          </constraint>
          ...
      </constraints>
   </constraintProvider>
</extension>
```

situation. To this end, we had recommended to install the "OCL Examples and Editors SDK" feature to your Eclipse in Sect. 4.3.1.1. If this feature is installed in Eclipse, you can open the "Console" view, and in that view select "Interactive OCL" as shown in Figure 4.12. In this console, select M1 in the drop down menu (marked by a red circle). If you then select an element in the Ecore editor, you can type some OCL constraint in the field at the bottom of the "Interactive OCL" view and check whether it is syntactically correct (you even get some syntax support, which indicates possible options while typing). Once you type enter, the field on top will show whether the syntax of the OCL constraint for the element selected in the Ecore model is syntactically correct.

Actually, the "Interactive OCL" console can not only be used for checking the syntactical correctness of OCL constraints. If you start the runtime workbench, and open an instance of a model, you can select an element of the instance, and evaluate an OCL expression on this instance. For example, you could open a PNML document and a page in the graphical editor, select an arc and evaluate the constraint. This is shown in Fig. 4.13, where the OCL expression `self.type.text` is evaluated on an arc (the one running from place p_7 to transition t_5; the evaluation shows that it is an inhibitor arc. Note that you need to switch the "Interactive OCL" console to "Modeling level" M2 for this purpose.

Checking syntactical correctness, exploring the possible options for expressions, and even for checking whether an OCL expression is evaluating as expected, the "Interactive OCL" console is a very efficient tool.

4.3.3.2 Java constraint

Listing 4.9 shows a Java implementation of a constraint, which guarantees that there are no duplicate arcs of type *read* or *inhibitor* between the same nodes. The `validation()` method of this class is called with some validation context from which the object on which the constraint should be checked can be obtained (the *target* that will be defined when the constraint is plugged in).

The implementation of the `validation()` method first obtains the target object form the validation context and checks whether it is an `Arc` of the our technical arc type. Then, it computes the "interpreted" type, of the arc via the utility class `TechnicalNetTypeFunctions`, which we had discussed in Sect. 4.3.2.4, as well as the nodes to which the source and target of this arc resolve (in case these are reference nodes); this is done with the `NetFunctions` utility class coming from the ePNK.

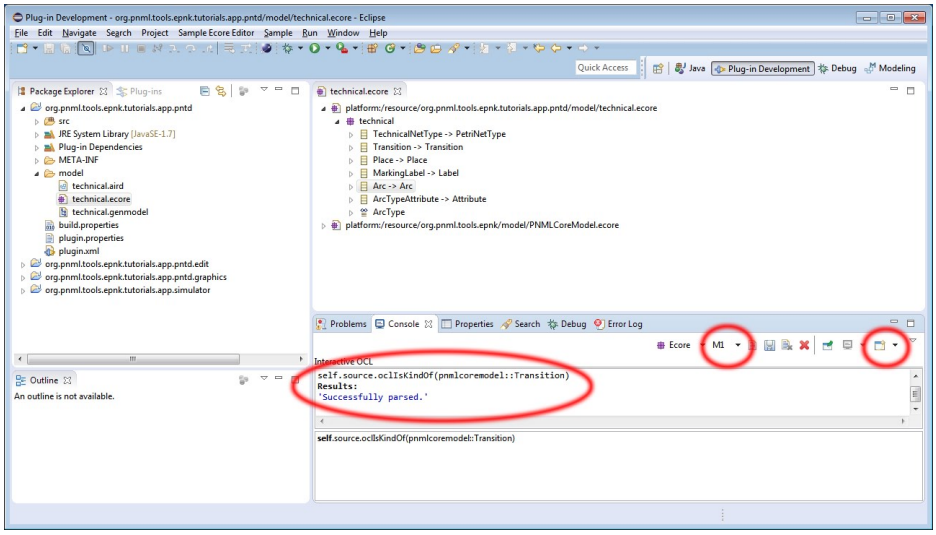

Figure 4.12: OCL console in Eclipse development workbench

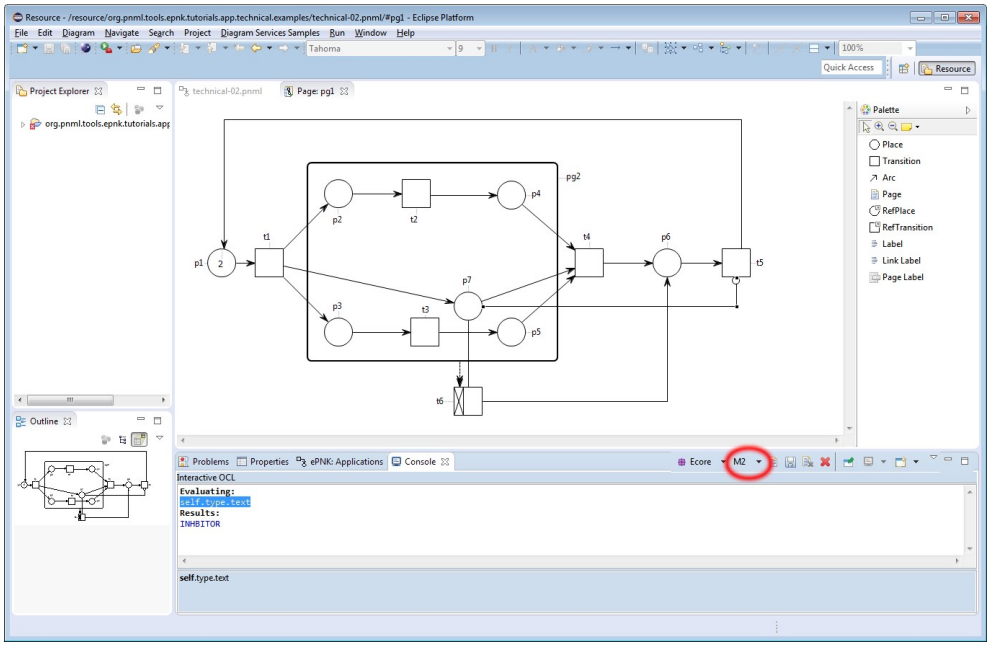

Figure 4.13: OCL console in Eclipse runtime workbench

After that a so-called `FlatAccess` object is obtained, which allows us to obtain all arcs that conceptually belong to a node, even when the node is actually split up via reference nodes on different pages. In case the arc is an *inhibitor* or *read* arc, and it has a source node and if the flatAccess object could be obtained, it iterates over all the other arcs that have the source node as their source too. The set of all these arcs can be easily obtained from the `flatAccess` object. Than it checks for all these `other` arcs (if they are different from the arc itself), whether the `other` arc is a duplicate, i.e. whether the other arc has the same target and the same type. In that case, the `validation()` method returns a failure status using the context – the singleton array contains the arc on which the constraint had failed.

Basically, a Java constraint is a class implementing a `validation()` method, which should return a failure status when the constraint is violated on the target object and a success status otherwise. The rest is left to your Java skills – which, it in many cases, makes Java constraints easier to formulate than OCL even tough the implementation looks a bit more verbose.

Note that the Java constraint is also marked `@generated NOT` since it is is manually written code in an plugin where most other code is automatically generated. This is not strictly necessary since also this class is placed in a Java package without any generated code. It is good practice to have clearly separate packages with generated code and manually written code.

After implementing a Java constraint it needs to be added to a constraint provider, which is similar to plugging in OCL constraints. It is in this part where it is defined on which target object the constraint should be checked and whether it acts as a *live constraint* or as a *batch constraint*. We chose to make the duplicate arcs constraint a *batch constraint* only. A *batch constraint* is checked only when the end-user explicitly starts a validation in the tree editor of a PNML document.

Listing 4.9 shows how the Java constrain is added to our constraint provider from earlier (see Listing 4.8, where some of the important features are highlighted in red. But, except for the *language*, which is Java now, and the class attribute, which refers to the fully qualified name of the Java class, this is very similar to the OCL constraint. Note that since the constraint is a *batch* constraint only, we do not need to define an *event* for the target object. The target itself refers to the same Ecore class as before, the arc of our new model.

Listing 4.9: Java class implementing the no duplicate arcs constraint

```
package org.pnml.tools.epnk.tutorials.app.technical.constraints;
// [...]

/**
 * [...]
 *
 * @author ekki@dtu.dk
 * @generated NOT
 */
public class NoDuplicateArcs extends AbstractModelConstraint {

  public IStatus validate(IValidationContext ctx) {
    EObject object = ctx.getTarget();
    if (object instanceof Arc) {
      Arc arc = (Arc) object;
      ArcType arcType = TechnicalNetTypeFunctions.getArcType(arc);
      Node source = NetFunctions.resolve(arc.getSource());
      Node target = NetFunctions.resolve(arc.getTarget());

      FlatAccess flatAccess =
        FlatAccess.getFlatAccess(NetFunctions.getPetriNet(arc));
      if ((arcType == ArcType.INHIBITOR ||
           arcType == ArcType.READ)        ) &&
          source != null &&
          flatAccess != null) {
        for (org.pnml.tools.epnk.pnmlcoremodel.Arc other:
            flatAccess.getOut(source)) {
          if (other != arc) {
            if (other instanceof Arc) {
              Arc arc2 = (Arc) other;
              Node target2 =
                NetFunctions.resolve(arc2.getTarget());
              if (target == target2) {
                if (TechnicalNetTypeFunctions.getArcType(arc) ==
                    TechnicalNetTypeFunctions.getArcType(arc2)) {
                  return ctx.createFailureStatus(new Object[]{arc});
    } } } } } } }
    return ctx.createSuccessStatus();
  }

}
```

Listing 4.10: `plugin.xml` adding the Java constraint

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

         <constraint
               lang="Java"
               class="org.pnml.tools.epnk.tutorials.app.technical.
constraints.NoDuplicateArcs"
               severity="ERROR"
               mode="Batch"
               name="No duplicate arcs"
               id="org.pnml.tools.epnk.tutorials.app.technical.
validation.no-duplicate-arcs"
               statusCode="402">
            <target class="Arc"/>
            <message>
               The arc {0} is a duplicate arc.
            </message>
            <description>
               Arcs of the same type (read or inhibitor) are not
               allowed between the same nodes.
            </description>
         </constraint>
      </constraints>
   </constraintProvider>
</extension>
```

4.3.4 Graphical extensions

In this section, we discuss how to implement the graphical extensions for our technical net type in more detail. As discussed in Sect. 4.1.1 and Sect. 4.2.2, our technical net type needs a customized graphical representation for arcs and for transitions.

In this tutorial, we demonstrate two different ways of implementing customized graphics for some net object. The first one is on a high level of abstraction: it, basically, changes attributes of the figure and composes the figure from other figures. The second one is on a lower level of abstraction: it overrides the method which draws the figure on the canvas. And, of course, both methods could be combined. We had discussed this already in Sect. 4.2.2.

In this section, we also discuss the mechanisms, which make sure that the graphical representation is properly updated, when attributes that affect the appearance change. Moreover, we discuss how to plug in the customized figures into the ePNK.

4.3.4.1 Project set up

Since graphics is conceptually separate from the model defining the Petri net type and since all code for graphics is programmed manually, the graphical extensions are typically implemented in a separate project. This project is a normal Eclipse *Plug-in Project*. In our example, it is the project `org.pnml.tools.epnk.tutorials.app.pntd.graphics`. But, the name "pntd" as part of the name indicates that this project belongs to the definition of the Petri net type conceptually.

In case you need to create a new such project, you can simply create a new *Plug-in project* in your workspace by choosing "New" in the "File" menu and then selecting "Plug-in project" from category "Plug-in Development"; we recommend to switch to the "Plug-in Development" perspective of Eclipse, which offers you the relevant tools and views for developing plug-ins in the toolbar and menus.

Once you have created a new plug-in project, you should add some dependencies to this project, which you typically will need for graphical extensions for a PNTD. You can do that by opening the "Plug-in manifest" editor by double-clicking on the `MNIFEST.MF` file in the `META-MF` folder and selecting the "Dependencies" tab. In addition to the project defining your Petri net type, you should add the project `org.eclipse.tools.epnk.diagram` to the "Required Plug-ins". In a plug-in project set up this way, you could

then implement the classes as discussed below yourself.

4.3.4.2 Arc: composing a figure

We start with discussing the graphical extension for arcs, where we use compose and configure the figure on a higher level of abstraction. Listing 4.11 shows the class implementing the graphical appearance of an arc of the technical net type. Part of this class, the `setGraphics()` method, has been shown in Listing 4.2 an discussed in Sect. 4.2.2 already; therefore, we have omitted this part indicated by ellipses and do not discuss this method here again. Listing 4.11 shows the remaining parts, in particular the constructor and the `update()` method, which we discuss below. We have discussed the `setGraphics()` method (lines 28–30) already in Sect. 4.2.2. Based on the current type (attribute `arcType`) of the arc. This attribute is actually the type that we had manually implemented (see Sect. 4.3.2.4) in the PNTD project for guaranteeing a uniform interpretation of the arc type values. The initial value of this attribute is computed in the constructor (line 5) by using the utility class `TechnicalNetTypeFunctions`, which also was manually implemented in the PNTD project. After setting this attribute the `setGraphics()` method is called from the constructor, in order to configure the graphics accordingly.

Note that the figure class inherits from class `ArcFigure`, which is defined by the ePNK, along with similar classes for figures for transitions and places. All these classes have an *update()* method, which will be called whenever something that might have an effect on the graphical appearance has changed. By default, a change of the source, the target, and any of the arcs labels and attributes defined in the respective net type are considered as potentially changing the graphical appearance, triggering the ePNK to call the *update()* method of the respective `ArcFigure`. It is up to the implementing figure to react to this update by overriding the *update()* method. In our example, this method temporarily stores the latest type of the arc, and then computes it again. If there was a change the `setGraphics()` is called again to properly configure the graphics.

Note that our implementation in lines 21-25 does slightly more. It computes the target of the arc and issues a notification of some change (not specified in detail) to the target. The reason for this is the following: The graphical appearance of a transition depends on whether it has normal incoming arcs or not. The transition figure will automtically be notified by the ePNK when arcs are attached to it or removed from it; but when the type of an attached arc changes, the transition is not notified automatically

Listing 4.11: Class for arc graphics

```
1  package org.pnml.tools.epnk.tutorials.app.graphics.figures;

   // [...]

   public class TechnicalNetTypeArcFigure extends ArcFigure {
6
     private ArcType arcType;

     public TechnicalNetTypeArcFigure(Arc arc) {
       super(arc);
11     arcType = TechnicalNetTypeFunctions.getArcType(arc);
       setGraphics();
     }

     @Override
16   public void update() {
       ArcType oldArcType = arcType;
       arcType = TechnicalNetTypeFunctions.getArcType(arc);
       if (arcType != oldArcType) {
         setGraphics();
21       Node target = arc.getTarget();
         if (target instanceof InternalEObject) {
           target.eNotify(new ENotificationImpl(
             (InternalEObject) target,
             Notification.NO_FEATURE_ID, null, null, null));
26   } } }

     private void setGraphics() {
       // [ ... ] discusses before
     }
31
   }
```

by the ePNK, because the type attribute belongs to the arc and not to the transition. Therefore, the arc figure needs to notify the target transition about a potential change, so that the transition figure can update its appearance if necessary. Our implementation does that with the code in lines 21-25. Generally, the implementation of the custom figures for a Petri net type can be mostly independent of each other. But, if the appearance of one element depends on features belonging to other elements, it would be the responsibility of these other figures to issue a notification as shown in 21-25. In our case, it is the `target` of the arc taht needs to be notified. Note that we do not need not notify the source of an arc, because only for arcs pointing to a transition the end-user is allowed to change the type (see discussion of constraints in Sect. 4.3.3.1). This shows that the notification might take careful considerations, taking the appearances of other figures and even constraints into account.

Issuing notifications actually needs some more consideration in order not to issue cyclic notifications. This is the most important reason to issue a notification only, if the type of the arc really changed. Another reason is efficiency – you do not need to redraw a figure if its appearance does not change.

4.3.4.3 Transition: drawing a figure

Listing 4.12 shows the complete implementation the appearance of transitions. It extends the `TransitionFigure`, which is provided by the ePNK. Similar to arcs, the constructor computes whether, initially the transition has normal input arcs and normal output arcs. The `TechnicalNetType Function` class provides two methods for that, which we did not discuss though. The reason for implementing these methods is again to make sure that there is a uniform interpretation of transition having normal in and out arcs. In the `update()` method, the old values of both attributes are temporarily stored, then these attributes are recomputed. If either of these values has changed the graphics is updated. Note however, that this is not done by changing the figure as such. Instead the `repaint()` method is called, which is a method every figure has; it will indirectly call the `fillShape()` method, which implements how a transition is drawn. We have seen this `fillShape()` method in Listing 4.3 on page 171 in Sect. 4.2.2 already. Therefore, we do not discuss it here again. Note that the `update()` method does not issue notifications on other object since other objects' appearance does not depend on features of a transition.

Note, however, that the graphical appearance of a transition might

Listing 4.12: Class for transition graphics

```
package org.pnml.tools.epnk.tutorials.app.graphics.figures;
// [...] import org.eclipse.draw2d.Graphics;

public class TechnicalNetTypeTransitionFigure
               extends TransitionFigure {

  private boolean hasNormalInArcs;
  private boolean hasNormalOutArcs;

  public TechnicalNetTypeTransitionFigure(Transition transition) {
    super(transition);
    hasNormalInArcs = TechnicalNetTypeFunctions.hasNormalInArcs(
        (org.pnml.tools.epnk.tutorials.app.technical.Transition)
        transition);
    hasNormalOutArcs = TechnicalNetTypeFunctions.hasNormalOutArcs(
        (org.pnml.tools.epnk.tutorials.app.technical.Transition)
        transition);
  }

  @Override
  public void update() {
    boolean oldHasNormalInArcs = this.hasNormalInArcs;
    boolean oldhasNormalOutArcs = this.hasNormalOutArcs;
    hasNormalInArcs = TechnicalNetTypeFunctions.hasNormalInArcs(
        (org.pnml.tools.epnk.tutorials.app.technical.Transition)
        transition);
    hasNormalOutArcs = TechnicalNetTypeFunctions.hasNormalOutArcs(
        (org.pnml.tools.epnk.tutorials.app.technical.Transition)
        transition);
    if (oldHasNormalInArcs != this.hasNormalInArcs ||
         oldhasNormalOutArcs != this.hasNormalOutArcs) {
      this.repaint();
    }
  }

  @Override
  protected void fillShape(Graphics graphics) {
    // [...] discussed earlier already
  }

}
```

change when an arc is added to a reference transition which refers to this transition. But, the ePNK takes care of issuing an update, also when features of reference transitions change. So, we do not need to do anything about that at all.

4.3.4.4 Graphical extension

The different figures that define a graphical extension of a Petri net type need to be combined and made available to the ePNK. This is done by implementing a class, which extends the class `GraphicalExtension` from the ePNK. This class serves two purposes: first, it has methods with some meta information on which Petri net types this class provides graphical extensions for and saying for which elements of the Petri net it provides graphical extensions; second, it serves as a factory that, for a given net element, creates an instance of a figure implementing the graphical representation of that element.

Listing 4.13 shows the class `TechnicalNetGraphics`, which combines the features of our graphical extensions and makes them available to the ePNK. The method `getExtendedNetTypes()` returns a list of classes representing the Petri net types for which this is an extension. Note that, this list does not refer to Java classes but to classes from the Ecore models (`EClass`) defining the Petri net type. Programmatically, these classes are available via objects that represent this package which have type `EPackage`. An instance of this class can be obtained from a class in the generated code; in our example, it is the available via the generated interface `TechnicalPackage`. The call `TechnicalPackage.eINSTANCE.getTechnicalNetType()` returns the Ecore class `TechnicalNetType`. Note that we can obtain the Ecore classes representing the place, the transition, and arc of the technical package in a similar way. Basically, The implementation of method `getExtended NetTypes()` says that it is responsible for Petri nets of `TechnicalNetType`. Note that it is actually possible that the same graphical extension provides graphical extensions for several Petri net types. This can be either by adding more net types to the list returned by `getExtendedNetTypes()`; or this can be by saying that the graphical extension applies to subtypes of the Petri net type, which we briefly discuss later.

Similarly, for a given net type the method `getExtendedNetObjects()` returns a list of Ecore classes extending places, transition and arcs, for which it defines graphical extensions. At last, there are methods which for a given net element provide a new instance of a figure for that element. In our case, it returns the figures that we have defined in Sect. 4.3.4.2 for arcs and

Listing 4.13: The graphical extension class for the Technical Net type

```
package org.pnml.tools.epnk.tutorials.app.graphics.factory;
// [...]
public class TechnicalNetGraphics extends GraphicalExtension {

  @Override
  public List<EClass> getExtendedNetTypes() {
    ArrayList<EClass> results = new ArrayList<EClass>();
    results.add(TechnicalPackage.eINSTANCE.getTechnicalNetType());
    return results;
  }

  @Override
  public List<EClass> getExtendedNetObjects(EClass netType) {
    ArrayList<EClass> results = new ArrayList<EClass>();
    if (netType.equals(TechnicalPackage.eINSTANCE.
        getTechnicalNetType())) {
      results.add(TechnicalPackage.eINSTANCE.getArc());
      results.add(TechnicalPackage.eINSTANCE.getTransition());
    }
    return results;
  }

  @Override
  public IArcFigure createArcFigure(Arc arc) {
    if (arc instanceof
        org.pnml.tools.epnk.tutorials.app.technical.Arc) {
      return new TechnicalNetTypeArcFigure(arc);
    }
    return null;
  }

  @Override
  public IUpdateableFigure createTransitionFigure(
      Transition transition) {
    if (transition instanceof
        org.pnml.tools.epnk.tutorials.app.technical.Transition) {
      return new TechnicalNetTypeTransitionFigure(transition);
    }
    return null;
  }
}
```

Sect. 4.3.4.3 for transitions, provided that the arc or transition are of the respective Java type.

Note that the class `GraphicalExtension` of the ePNK, has several other methods, which allow us to define a priority for a graphical extension, which might be needed when several extensions for the same net type and element would be available. Moreover there are methods for defining whether the extension would apply for all subtypes of a given net type and for extended elements. By default, a graphical extension has priority 0 and does neither apply to subtypes of net types nor to extended elements. Changing these setting might have quite far-reaching consequences, which we do ot discuss here in detail. Therefore, we recommend not to change the defaults.

4.3.4.5 Pluging in the graphical extension

Ultimately, the `GraphicalExtension` needs to be plugged in to the ePNK, so that the ePNK will know about it. The easiest way to do that is again to copy a XML snippet to the `plugin.xml` of the project where the graphical extension class is defined. A minor complication might be that the `plugin.xml` does not exist in a newly created plug-in project. So we need to create it first. The easiest way to do that is opening the "Plug-in manifest" as discussed above and to select the "Extensions" tab; pressing the "Add..." button, but cancelling the opened dialog right away. After that, we have create a new and empty `plugin.xml` file in our project.

The snippet that plugs in our `TechnicalNetGraphics` to the ePNK is shown in Listing 4.14. The parts which can be freely chosen are marked in red: the `id`, the `name`, the `class` and the `description`. The `class` attribute, must of course refer to a Java class extending `GraphicalExtension` by a fully qualified name of a Java class; and this class must be on the class path of this project; a warning in the `plugin.xml` will indicate if this is not the case. The `point` attribute of the extension must refer to the ePNK extension point `org.pnml.tools.epnk.diagram.graphics` for graphical extensions.

Once you have finished a plugin project with a graphical extension, it is a good idea to check whether it works in the ePNK. To this end, start the runtime workbench of Eclipse and, in this runtime workbench create a net of your new type and open a graphical editor on it. If your graphical extensions do not properly appear, it is a good idea to start the runtime workbench in a debugger. By setting a break point in the constructor of the graphical extension, you can see whether your extension is ever loaded by the ePNK. If not, you probably forgot to plug in the extension, or the class

Listing 4.14: Snippet from `plugin.xml` for pluggin in the graphical extension

```
<extension
      id="org.pnml.tools.epnk.tutorials.app.graphics"
      name="Technical Net Graphics"
      point="org.pnml.tools.epnk.diagram.graphics">
   <graphicsextension
         class="org.pnml.tools.epnk.tutorials.app.graphics.
factory.TechnicalNetGraphics"
         description="Dedicated graphics for Technical Net Type">
   </graphicsextension>
</extension>
```

the extension is referring to does not exist at all or is not on the class path. If the class is loaded, you might set break point in the other methods and see what happens when the ePNK calls these methods.

4.3.5 Simulator application

At last, we discuss the technical details of the implementation of the simulator for our technical Petri net type. In our example projects, this is realized in a separate EMF project: `org.pnml.tools.epnk.tutorials.app.simulator`. It is realized as an EMF project, since we extend the *ePNK annotation model* by some specific types of annotations. This is done by an Ecore model which extends the ePNK annotation model.

In the following, we briefly discuss our extended annotation model, which we call `technicalannotations` (refering to our *technical* net type), where the focus is on how to create it and how to generate code from it. Then, we discuss some core parts of the implementation, a class representing a marking of nets of our technical Petri net type, and core functions for realzing the functionality.

In the end, we discuss the *presentation handler* defining the graphical representation of the annotations, the *action handler* which handles user actions, and how to combine all the parts into an application, and how to plug in the simulator application to the ePNK.

4.3.5.1 Annotation model

We had discussed the annotations that we need for our simulator already in Sect. 4.2.3.1. The Ecore model of these annotations was shown in Fig. 4.7

on page 173. Basically, there is an annotation for *weakly enabled transitions*, which has an attribute saying whether the transition is truly *enabled* or only weakly enabled. Moreover, there is an annotation for the arcs that prevent the true enabledness of a transition; we call this annotation `InvolvedArc`. This annotation allows the end-user to activate or deactivate these arcs, where the current status is indicated by attribute *active*. The `InvolvedArc` is associated with the corresponding `EnabledTransition` annotation so that we can navigate back and forth between these related annotations. The `EnabledTransition` is also used for annotating reference transitions that refer to enabled transitions. Conceptually, these `EnabledTransition` annotations belong to each other, which is represented by the reference `ref` and `resolve` which are opposites of each other. At last, there is an annotation indicating the `Marking` of a place or an associated reference place. The attribute `value` represents the current number of tokens on that place.

When creating the project from scratch, you would first create an empty *EMF project* and then a new *Ecore model* as discussed in Sect. 4.3.2.2. Again, you would give the package a reasonable name, `techsimannotations` in our example, and chose some *namespace* prefix and a *URI* for this package.

Then, you would create the classes of the model as shown in Fig. 4.7 on page 173. Note that these classes inherit from the ePNK ; the `Marking` inherits also from `TextualAnnotation`. Both class `ObjectAnnotation` and `TextualAnnotation` come from the ePNK base package `http://tools.pnml.org/epnk/netannotations/1.0`. Before you can add these classes as super types to your classes, you must load the `http://tools.pnml.org/epnk/netannotations/1.0` as a resource to your editor by using the "Load Resource" action as discussed in Sect. 4.3.2.2; make sure that you use the "Browse Target Platform Packages..." feature to selec the package `http://tools.pnml.org/epnk/netannotations/1.0`.

Then, you can choose the super types for your new annotation classes. Note that in Ecore models, it is possible to chose more than one super type; Ecore supports multiple inheritance.

In this Ecore model there occurs one feature which we have not discussed before. There are references which are *opposites* of each other, in a sense they form two ends of the same association. In order to create such opposite references in the Ecore model editor, you would create two independent references in opposite directions first; then, you would make them *opposites* of each other: to this end, you would chose one reference, and in the properties set the property "EOpposite" choosing the reference into the other direction.

After that, you can create an EMF Generator model from your new Ecore model and generated code from it as discussed in Sect. 4.3.2.3. In the "Package Selection" dialog when creating a new generator model, remember that the only package selected in section "Root package" should be your new model; for all models loaded from the ePNK, the gen models should be selected in the "Referenced generator models". Don't forget to set the `Base Package` property in your generator model and to set the `Operation Reflection` to `false` as discussed in Sect. 4.3.2.3.

Once the gen model is created, you can generate the model code. Note that, for the annotation model, it is enough that you generate the model code; you do not need to generate the edit, the editor or the test code.

Once you have generated the code for your annotations, you can start realizing the actual simulator application. Before you start with that, you might want to add some additional dependencies to your project (by opening the "Plug-in manifest" editor clicking on the `plugin.xml`). The code generator will have added some dependencies to your project already.

In addition to the project which defines your PNTD, the project `org.pnml.tools.epnk.tutorials.app.pntd` in our example, and the ones which the code generator has added already, you will probably need to add the following projects to the dependencies of your project: `org.pnml.tools.epnk.applications` and `org.eclipse.gmf.runtime.diagram.ui`.

4.3.5.2 NetMarking

In a simulator, the *marking* of a net plays a key role since it represents the current state of a Petri net in a simulation. Conceptually, a marking is a mapping from places to integers. So the marking could be represented as a Java `Map<Place,Integer>`. But, accessing a Java `Map<Place,Integer>` and keeping it consistent, might be a bit tedious. Therefore, we implemented a class `NetMarking` in our simulator, which eases the access and update of such mappings. Internally, the marking is represented as a Java `Map<Place,Integer>`; but the class `NetMarking` provides some methods for easier manipulating the marking.

Programming this class is straight-forward; therefore, we do not discuss the implementation here. You can look up the implementation of this class in the project `org.pnml.tools.epnk.tutorials.app.simulator` of the code provided for this tutorial; you will find it in the Java package `org.pnml.tools.epnk.tutorials.app.simulator.marking`. Here, we show the available methods of this class only, since we will use them later in the implementation of the simulator. Listing 4.15 shows all the methods of class

`NetMarking`.

Listing 4.15: Methods of the class `NetMarking`

```
public class NetMarking {

  private Map<Place,Integer> marking;

   // Creates a new empty marking.
  public NetMarking()

  //Creates a new marking, which is a copy of a given marking.
  public NetMarking(NetMarking marking)

  public void setMarking(Place place,int value)

  public int getMarking(Place place)

  public void incrementMarkingBy(Place place, int increment)

  public void decrementMarkingBy(Place place, int decrement)

  // Adds marking2 to this marking
  public void add(NetMarking marking2)

  // Subtracts marking2 from this marking
  public void subtract(NetMarking marking2)

  // Checks whether this marking is greater or equal than marking2
  public boolean isGreaterOrEqual(NetMarking marking2)

  // Returns the set of all places that have at least one token
  public Collection<Place> getSupport()

}
```

4.3.5.3 Core functions

In this section, we discuss the implementation of the method which computes from a given marking and a given enabled transition a new marking, which results from firing this transition. This methods implements the core functionality of our simulator.

In order to make it easy to implement this method, our simulator implements some more basic functions. The first two methods compute for a given marking how many tokens a transition will consume from each place and how many tokens it will produce on each place. Actually, the number of consumed and produced tokens for each place can be considered to be markings again. So the result type of these methods is `NetMarking`. The implementation of these two methods are shown in Listing 4.16. The implementation is straight-forward: Initially an new empty marking is created. Then, the methods iterate over all in-coming or out-going arcs, and, for each normal arc, incrementing the marking for the source or the target place, respectively. In the end, the resulting marking is returned. The only interesting point is that we, again, use the `FlatAccess` for obtaining all in-coming and out-going arcs of the transition and for resolving reference places to the actual place they refer to. As we see later, our simulator has a method, which obtains an instance of `FlatAccess` for the net on which the application is running, and we use this method `getFlatAccess()` in these methods `consumes` and `produces`.

There are two other functions, which we need in our simulator: `isWeakly Enabled()`, which computes for a given marking whether a given transition is weakly enabled, meaning that considering normal arcs only, the transition would be enabled; the other `preventingArcs()` computes for a given marking and a given weakly enabled transition, which *read* arcs and which *inhibitor* arcs would prevent it from firing anyway. The implementation of both of these methods is shown in Listing 4.17. Note that due to our `consumes()` method and the `isGreaterOrEqual()` method for markings the implementation of the `isWeaklyEnabled()` method is very simple. We just need to compute that consumed tokens of the transition and check whether the marking is greater than that. For computing the preventing arcs, we need to iterate over all the incoming *arcs*: a *read arc* will be added to the result, if its source place does not have a token in the given marking; a *inhibitor arc* will be added, if the source place has a token in the given marking.

At last, it is easy to implement the `fireTransition()` method based on the previous methods. Listing 4.18 shows the implementation of this method. It starts with copying the marking from which the transition should be fired. Then, it consumes the token from the incoming normal arcs, resets the places on pages that have a reset arc to the transition; and, at last, it produces the tokens on the out-going normal arcs. The trickiest part is the reset of all places on sub pages, even though the implementation is straight-forward.

Listing 4.16: Implementation of the `consumes()` and `produces()` methods

```
private NetMarking consumes(Transition transition) {
  FlatAccess flatAccess = this.getFlatAccess();

  NetMarking consumes = new NetMarking();
  for (org.pnml.tools.epnk.pnmlcoremodel.Arc arc:
      flatAccess.getIn(transition)) {
    if (arc instanceof Arc &&
        TechnicalNetTypeFunctions.getArcType(arc) ==
          ArcType.NORMAL ) {
      Node source = arc.getSource();
      if (source instanceof PlaceNode) {
        source = flatAccess.resolve((PlaceNode) source);
        if (source instanceof Place) {
          consumes.incrementMarkingBy((Place) source, 1);
  } } } }
  return consumes;
}

private NetMarking produces(Transition transition) {
  FlatAccess flatAccess = this.getFlatAccess();

  NetMarking produces = new NetMarking();
  for (org.pnml.tools.epnk.pnmlcoremodel.Arc arc:
      flatAccess.getOut(transition)) {
    if (arc instanceof Arc &&
        TechnicalNetTypeFunctions.getArcType(arc) ==
          ArcType.NORMAL ) {
      Node target = arc.getTarget();
      if (target instanceof PlaceNode) {
        target = flatAccess.resolve((PlaceNode) target);
        if (target instanceof Place) {
          produces.incrementMarkingBy((Place) target, 1);
  } } } }
  return produces;
}
```

Listing 4.17: Code for `isWeaklyEnabled()` and `preventingArcs()`

```
private boolean isWeaklyEnabled(NetMarking marking,
    Transition transition) {
  NetMarking consumes = consumes(transition);
  return marking.isGreaterOrEqual(consumes);
}

private Collection<Arc> preventingArcs(NetMarking marking,
    Transition transition) {
  FlatAccess flatAccess= this.getFlatAccess();

  Collection<Arc> preventors = new ArrayList<Arc>();
  for (org.pnml.tools.epnk.pnmlcoremodel.Arc arc:
      flatAccess.getIn(transition)) {
    ArcType arcType = TechnicalNetTypeFunctions.getArcType(arc);
    if (arc instanceof Arc &&
        ( arcType == ArcType.INHIBITOR ||
          arcType == ArcType.READ)) {
      Node source = arc.getSource();
      if (source instanceof PlaceNode) {
          source = flatAccess.resolve((PlaceNode) source);
        if (source instanceof Place) {
          if (arcType == ArcType.INHIBITOR &&
              marking.getMarking((Place) source) > 0) {
            preventors.add((Arc) arc);
          } else if (arcType == ArcType.READ &&
              marking.getMarking((Place) source) == 0) {
            preventors.add((Arc) arc);
  } } } } }
  return preventors;
}
```

Listing 4.18: Impementation of the `fireTransition()` method

```
private NetMarking fireTransition(NetMarking marking1,
    Transition transition ) {
  FlatAccess flatAccess= this.getFlatAccess();

  NetMarking marking2 = new NetMarking(marking1);

  // consume tokens from preset
  NetMarking consumes = consumes(transition);
  marking2.subtract(consumes);

  // reset places on page connected to reset arc
  for (Object a: flatAccess.getIn(transition)) {
    if (a instanceof Arc &&
        TechnicalNetTypeFunctions.getArcType((Arc) a) ==
        ArcType.RESET) {
      Arc arc = (Arc) a;
      Node source = arc.getSource();
      if (source instanceof Page) {
        Page page = (Page) source;
        for (Object object: page.getObject()) {
          if (object instanceof PlaceNode) {
            Object resolved =
              flatAccess.resolve((PlaceNode) object);
            if (resolved instanceof Place) {
              marking2.setMarking((Place) resolved, 0);
  } } } } } }

  // produce tokens on postset
  NetMarking produces = produces(transition);
  marking2.add(produces);

  return marking2;
}
```

4.3.5.4 Annotation functions

In Sect. 4.3.5.3, we have discussed the methods which implement pure functionality on Petri nets. In order to visualize the markings, we need some additional functions or methods which ultimately show the markings in a net. To this end, we need to convert a marking into a net annotation. And we need a way to convert a net annotation into a marking. This separation would actually not be strictly necessary, but it makes the design clearer and the easier implementation easier understand.

Basically, we need the following methods: One for computing the initial marking from the net itself, one for computing a marking from the current annotation of the net, and one for creating a new net annotation from a given marking (showing the marking as well as the weakly enabled transitions and the involved arcs).

Listing 4.19 shows the method for computing the initial marking of a net. The method initializes a new empty marking. Then, it iterates over all the places of the net (using the `FlatAccess` object from the application again) and sets the value of the marking for each place (if it is not zero) accordingly.

Listing 4.19: Implementation of the `computeInitialMarking()` method

```
private NetMarking computeInitialMarking() {
  NetMarking marking = new NetMarking();;
  for (org.pnml.tools.epnk.pnmlcoremodel.Place place:
      this.getFlatAccess().getPlaces()) {
    if (place instanceof Place) {
      int number = TechnicalNetTypeFunctions.getMarking(place);
      if (number > 0) {
        marking.setMarking((Place) place, number);
  } } }
  return marking;
}
```

The method for computing the marking from the current annotation of the application is similar. It is shown in Listing 4.20. The only difference is that the value of the marking of each place is not taken from the model of the net, but from the `Marking` annotations of the current annotation of the application. This current annotation is obtained by calling `getNetAnnotations().getCurrent()` the call `getObjectAnnotations()` returns a list of all the individual object annotations. For each annotation

it is checked whether it is a `Marking` annotation, and the underlying object is obtained. If the underlying object is a place, the value of the `Marking` annotation is set as marking for that place.

Listing 4.20: Implementation of the `computeMarking()` method

```
private NetMarking computeMarking() {
  NetMarking marking = new NetMarking();
  for (ObjectAnnotation annotation:
4       getNetAnnotations().getCurrent().getObjectAnnotations()) {
    if (annotation instanceof Marking) {
      Marking markingAnnotation = (Marking) annotation;
      Object object = markingAnnotation.getObject();
      int value = markingAnnotation.getValue();
9       if (object instanceof Place &&  value > 0) {
        Place place = (Place) object;
        marking.setMarking(place, value);
  } } }
  return marking;
14 }
```

The most intricate method is `computeAnnotation()`, which takes a `NetMarking` and computes a new net annotation (`NetAnnotation`) representing this marking and also indicating the enabled transitions and the preventing arcs in this marking. The implementation of this method is shown in Listings 4.21 and 4.22 (we needed to split this listing up in two). In addition to obtaining the `FlatAccess`, the method initially creates a new and empty `NetAnnotation` by using the factory (from the ePNK). And it sets the net annotation to the Petri net of the application.

Then, there are two major loops. The first (in Listing 4.21) deals with annotating enabled transitions and their arcs; the second (in Listing 4.22) deals with annotating the places with the markings. When a transition is enabled, an `EnabledTransition` object is created (using the factory which was generated from our new annotation model). The object of this annotation is set to the transition and added to the net annotations, the `enabled` attribute is set to `true` or not; if there are preventing arcs, these are annotated with an `InvolvedArc` annotation, initially setting them to `active`. Note that these `InvolvedArc` annotations are attached to the respective `EnabledTransition`. At last, all the reference transitions pointing to the enabled transitions are also annotated with an `EnabledTransition` annotation.

Listing 4.21: Implementation of `computeAnnotation()` (part 1)

```
private NetAnnotation computeAnnotation(NetMarking marking) {
  FlatAccess flatAccess = this.getFlatAccess();

  NetAnnotation annotation =
    NetannotationsFactory.eINSTANCE.createNetAnnotation();
  annotation.setNet(getPetrinet());
  for (Object object: flatAccess.getTransitions()) {
    if (object instanceof Transition) {
      Transition transition = (Transition) object;
      if (isWeaklyEnabled(marking, transition)) {
        EnabledTransition enabledTransition =
          TechsimannotationsFactory.eINSTANCE.
          createEnabledTransition();
        enabledTransition.setObject(transition);
        annotation.getObjectAnnotations().add(enabledTransition);
        Collection<Arc> preventingArcs =
          this.preventingArcs(marking, transition);
        if (preventingArcs.isEmpty()) {
          enabledTransition.setEnabled(true);
        } else {
          enabledTransition.setEnabled(false);
          for (Arc arc : preventingArcs) {
            InvolvedArc involvedArc =
              TechsimannotationsFactory.eINSTANCE.
              createInvolvedArc();
            involvedArc.setObject(arc);
            involvedArc.setTarget(enabledTransition);
            involvedArc.setActive(true);
            annotation.getObjectAnnotations().add(involvedArc);
        } }
        for (RefTransition refTransition:
            flatAccess.getRefTransitions(transition)) {
          EnabledTransition enabledTransition2 =
            TechsimannotationsFactory.eINSTANCE.
            createEnabledTransition();
          enabledTransition2.setObject(refTransition);
          enabledTransition2.setResolve(enabledTransition);
          enabledTransition2.setEnabled(enabledTransition.
            isEnabled());
          annotation.getObjectAnnotations().add(enabledTransition2);
  } } } }
```

Listing 4.22: Implementation of `computeAnnotation()` (part 2)

```
  for (Place place: marking.getSupport()) {
    int m = marking.getMarking(place);
    if (m > 0) {
      Marking mAnnotation =
        TechsimannotationsFactory.eINSTANCE.createMarking();
      mAnnotation.setObject(place);
      mAnnotation.setValue(m);
      annotation.getObjectAnnotations().add(mAnnotation);
      for (RefPlace refPlace: flatAccess.getRefPlaces(place)) {
        Marking mAnnotation2 =
          TechsimannotationsFactory.eINSTANCE.createMarking();
        mAnnotation2.setObject(refPlace);
        mAnnotation2.setValue(m);
        annotation.getObjectAnnotations().add(mAnnotation2);
  } } }
  return annotation;
}
```

The second loop (in Listing 4.22) annotates each place that has at least one token with a `Marking` annotation – and all reference places referring to that place get such an annotation too.

Note that all the methods discussed above, are pure functions; they do not change the state of the application at all. At some point, of course, the simulator needs to change the state (current marking) of the net. To this end, the simulator implements another `fireTransition()` method with a different signature than the one from before. This method will be called when the end-user actually fires a transition, which we discuss later in Sect. 4.3.5.5. This second `fireTranstion()` method is shown in Listing 4.23. It has a transition as a parameter and two sets of arcs, which are the arcs which the user had selected to be inactive; the first are the ingoing inactive arcs, the second are the outgoing inactive arcs. Actually, our implementation needs the ingoing arcs only. But since other applications might need both, we chose to have this parameter here, just to indicate the possibility. First, the `fireTransition()` method computes the marking from the current annotation, by using the method, which we had discussed before. Then, the arcs that would prevent its firing in the current marking are computed, and the inactive arcs are removed from it. If the set is empty, the transition is actually enabled – and fired. Then, the first

`fireTransition()` computes the next marking, and a new net annotation is computed from it with `computeAnnotation()`, which we had discussed before. At last, the new net annotation is added to the application (which will implicitly present it to the user – by mechanisms discussed later). There is some subtlety though: the application maintains a sequence of net annotations, which reflects the sequence of transitions and resulting markings the user has fired. The user can, by using the applications GUI elements, navigate back and forth in this sequence. So the current marking might not be the last marking in the sequence. When the user fires a transition in a marking, which is not the last one, we must delete all net annotations after the current one. Then, we add the new net annotation as the current one (which will implicitly be added at the end of the sequence).

Listing 4.23: Implementation of the `fireTransition()` method

```
boolean fireTransition(Transition transition,
    Collection<Arc> inactiveInArcs,
    Collection<Arc> inactiveOutArcs) {
  NetMarking marking1 = this.computeMarking();

  Collection<Arc> preventors =
    this.preventingArcs(marking1, transition);
  preventors.removeAll(inactiveInArcs);
  if (this.isWeaklyEnabled(marking1, transition) &&
      preventors.isEmpty()) {
    NetMarking marking2 = this.fireTransition(marking1, transition);
    NetAnnotation netAnnotation = this.computeAnnotation(marking2);

    this.deleteNetAnnotationAfterCurrent();
    this.addNetAnnotationAsCurrent(netAnnotation);
    return true;
  }
  return false;
}
```

4.3.5.5 Action handlers

The actions of an application that can be triggered by the end-user are defined by *action handlers*, which are registered with the application itself. An action handler provides methods, which will be called when the user presses, double clicks or releases a mouse button on some annotation of a

Petri net object. The implementation of these methods define what should happen in this case. The action handler can decide to ignore the action, in which case it would return `false` in order to indicate that it did not handle the event, allowing other registered action handlers to kick in. In case the action handler has handled the event, the action handler should return `true`.

Listings 4.24 and 4.25 show the implementation of the class `EnabledTransitionHandler`, which defines what happens when the end-user double clicks on a transition with an `EnabledTransition` annotation. It ignores single mouse presses and mouse releases, since the respective methods always return `false` (see lines 14–24). Note that all these handler methods take two parameters, a `MouseEvent` coming from Eclipse's Draw2D, and an `ObjectAnnotation` of the ePNK. Note that the `ObjectAnnotation` will typically be of a type defined by your application.

Listing 4.24: Implementation of the `EnabledTransitionHandler` (part 1)

```
1  package org.pnml.tools.epnk.tutorials.app.simulator.application;

   // [...]

   public class EnabledTransitionHandler implements IActionHandler {
6
     private TechnicalSimulator application;

     public EnabledTransitionHandler(TechnicalSimulator application) {
       super();
11     this.application = application;
     }

     @Override
     public boolean mousePressed(MouseEvent arg0,
16       ObjectAnnotation annotation) {
       return false;
     }

     @Override
21   public boolean mouseReleased(MouseEvent arg0,
         ObjectAnnotation annotation) {
       return false;
     }
```

The interesting method in the `EnabledTransitionHandler` is the `mouse`

Listing 4.25: Implementation of the `EnabledTransitionHandler` (part 2)

```
  @Override
  public boolean mouseDoubleClicked(MouseEvent arg0,
      ObjectAnnotation annotation) {
    NetAnnotations netAnnotations = application.getNetAnnotations();
    NetAnnotation current = netAnnotations.getCurrent();
    if (current.getObjectAnnotations().contains(annotation)) {
      Object object = annotation.getObject();
      if (object instanceof TransitionNode) {
        object = NetFunctions.resolve((TransitionNode) object);
      }
      if (object instanceof Transition &&
          annotation instanceof EnabledTransition) {
        Transition transition = (Transition) object;
        EnabledTransition enabledTransition =
          (EnabledTransition) annotation;
        if (enabledTransition.isEnabled()) {
          Collection<Arc> inactiveInArcs = new HashSet<Arc>();
          for (InvolvedArc a: enabledTransition.getIn()) {
            Object o = a.getObject();
            if (o instanceof Arc && !a.isActive()) {
              inactiveInArcs.add((Arc) o);
          } }
          Collection<Arc> inactiveOutArcs = new HashSet<Arc>();
          for (InvolvedArc a: enabledTransition.getOut()) {
            Object o = a.getObject();
            if (o instanceof Arc && !a.isActive()) {
              inactiveOutArcs.add((Arc) o);
          } }
          return application.fireTransition(transition,
              inactiveInArcs,
              inactiveOutArcs);
    } } }
    return false;
  }

}
```

`DoubleClicked()` method (lines 26–59 in Listings 4.25), which issues the `fireTransition()` method of the simulator application. Implemented in a defensive way, this method first checks whether the provided object annotation is actually in the current annotation. Then it checks whether the annotated object is a transition (maybe resolving a reference transition) and the annotation is an `EnabledTransition` annotation. Then it computes which of the incoming and out-going arcs are inactive. The transition and the sets of these arcs are then provide as parameters when the `fireTransition()` method of the application is called, which will do the "heavy lifting".

There is one other action of the end-user that our simulator application needs to handle: the end-user deactivating or activating an arc, which might prevent a weakly enabled transition from firing. In principle, this could be implemented within the same action handler as firing the transition. To keep things separate, however, we have chosen to implement a separate `InvolvedArcHandler`, which is shown in Listing 4.26. This handler, handles a single mouse press, which is implemented in the `mousePressed()` method on an `InvolvedArc` annotation; all other events and other annotations will be ignored (the respective methods returning `fase` are omitted from the code in Listing 4.26). In case the involved annotation is an `InvolvedArc` annotation, the `active` attribute of this annotation is toggled, and for the attached `EnabledTransition` annotation, we recompute its activation status: if all `InvolvedArc`s are inactive, the transition is enabled; if at least one `InvolvedArc` is `active`, the transition is not enabled. If the enabledness of the transition has changed, the `enabled` attribute of the `EnabledTransition` annotation and all the ones referring to it are updated.

At last, all `NetAnnotation`s after the current one are deleted – just to make sure that all the net annotations of the application form a consistent firing sequence. In case this operation actually deletes a net annotation, the ePNK will automatically update the presentation of the annotations. But, in case nothing changes, we need to issue an update of the presentation of the annotations explicitly by calling `update()` on the application.

4.3.5.6 Presentation handler

Up to now, our application was defined by adding, changing, and updating annotations. In this section, we discuss how to define how annotations are actually shown to the end-user. This is implemented by one or more `PresentationHandler`s. Actually, for very simple applications, the default `PresentationHandler` provided by the ePNK might me enough already. But, as soon as you want to use different colors or different shapes, an

Listing 4.26: Implementation of the `InvolvedArcHandler`

```
package org.pnml.tools.epnk.tutorials.app.simulator.application;
// [...]
public class InvolvedArcHandler implements IActionHandler {
  private TechnicalSimulator application;

  public InvolvedArcHandler(TechnicalSimulator application) {
    super();
    this.application = application;
  }

  @Override
  public boolean mousePressed(MouseEvent arg0,
      ObjectAnnotation annotation) {
    if (annotation instanceof InvolvedArc) {
      InvolvedArc involvedArc = (InvolvedArc) annotation;
      involvedArc.setActive(!involvedArc.isActive());
      EnabledTransition transition = involvedArc.getTarget();
      if (transition != null) {
        boolean active = transition.isEnabled();
        boolean result = true;
        for (InvolvedArc other: transition.getIn()) {
          if (other.isActive()) {
            result = false;
            break;
        } }
        if (active != result) {
          transition.setEnabled(result);
          for (EnabledTransition refTrans: transition.getRefs()) {
            refTrans.setEnabled(result);
        } }
        int size = application.getNetAnnotations().
            getNetAnnotations().size();
        application.deleteNetAnnotationAfterCurrent();
        if (size == application.getNetAnnotations().
              getNetAnnotations().size()) {
          application.update();
        }
        return true;
    } }
    return false;
  } // [...]
```

application needs to implement its own `PresentationHandler`s.

In our case, `Marking` annotations for places should be shown as a text label to the top-right corner of the places. This, will actually be handled by the default presentation handler, which will show all textual annotations as a blue label to the top-right of the corresponding element. For `EnabledTransitions` and `InvolvedArc` annotations, we need to define a dedicated presentation handler, since the colour changes depending on the annotation's attributes. Transitions that are weakly enabled only should be shown in grey, transitions that are truly enabled (maybe by the user deactivating the preventing arcs) should be shown in red. Involved arcs that are activated (and therefore preventing the transition from firing) should be shown in grey; deactivated arcs should be shown in red (reminding the end-user that firing them might deviate from the usual behaviour).

Listings 4.27 and 4.28 show the implementation of the class `TechnicalAnnotationsPresentationHandler` of our implementation. The presentation `handler()` method has two parameters: an `ObjectAnnotation` and an `AbstractGraphicalEditPart`. The `ObjectAnnotation` is the one which the handler should provide an graphical representation for, and the `AbstractGraphicalEditPart` provides access to the editor graphics of the underlying Petri net element (which is a concept from GEF). The implementation handles two main cases: the first case handles `EnabledTransition` annotations, the other `InvolvedArc` annotations. For an `EnabledTransition` annotation, the method creates a `RectangleOverlay`, and sets its colours according to the value of the `enabled` attribute to red or grey (using Eclipse SWT's colour constants). The `RectangleOverlay` is defined by the ePNK and is supposed to be an overlay over an existing figure – underlying the `GraphicalEditPart`. For an `InvolvedArc`, the method creates a `PolylineOverlay`, and sets its colours according to the value of the `active` attribute to grey or red. Similar to the `RectangleOverlay`, access to the underlying graphical representation of the arc is via the `ConnectionNodeEditPart`. Note that depending on whether the underlying object is a node or an arc, the `AbstractGraphicalEditPart` needs to be cast to either `GraphicalEditPart` or `ConnectionNodeEditPart`. In either case, the overlays adjust their size and position to the underlying graphical object – even when the user changes the size and position later on.

Note that the handler can also return `null`, which indicates that it does does not have a graphical representation for the annotation in the given situation; in that case, an other handler might provide one. If no handler can provide a representation, the annotation is not shown at all. But, typically the default presentation handler takes care of them – unless the default

Listing 4.27: Implementation of the presentation handler (part 1)

```
package org.pnml.tools.epnk.tutorials.app.simulator.application;

// [...]
public class TechnicalAnnotationsPresentationHandler
    implements IPresentationHandler {

  @Override
  public IFigure handle(ObjectAnnotation annotation,
      AbstractGraphicalEditPart editPart) {
    if (annotation instanceof EnabledTransition) {
      if (editPart instanceof GraphicalEditPart) {
        GraphicalEditPart graphicalEditPart =
          (GraphicalEditPart) editPart;
        java.lang.Object modelObject =
          graphicalEditPart.resolveSemanticElement();
        if (modelObject instanceof TransitionNode) {
          RectangleOverlay overlay =
            new RectangleOverlay(graphicalEditPart);
          if (((EnabledTransition) annotation).isEnabled()) {
            overlay.setForegroundColor(ColorConstants.red);
            overlay.setBackgroundColor(ColorConstants.red);
          } else {
            overlay.setForegroundColor(ColorConstants.lightGray);
            overlay.setBackgroundColor(ColorConstants.lightGray);
          }
          return overlay;
      } }
```

Listing 4.28: Implementation of the presentation handler (part 2)

```
    } else if (annotation instanceof InvolvedArc) {
      InvolvedArc involvedArc = (InvolvedArc) annotation;
      if (editPart instanceof ConnectionNodeEditPart) {
        ConnectionNodeEditPart connectionEditPart =
          (ConnectionNodeEditPart) editPart;
        java.lang.Object modelObject =
          connectionEditPart.resolveSemanticElement();
        if (modelObject instanceof Arc) {
          PolylineOverlay overlay =
            new PolylineOverlay(connectionEditPart);
          if (involvedArc.isActive()) {
            overlay.setForegroundColor(ColorConstants.lightGray);
            overlay.setBackgroundColor(ColorConstants.lightGray);
          } else {
            overlay.setForegroundColor(ColorConstants.red);
            overlay.setBackgroundColor(ColorConstants.red);
          }
          return overlay;
    } } }
    return null;
  }
}
```

presentation handler is explicitly removed from the application.

Note that the default annotation handlers will provide representation for all object annotations, which will be overlays in red; if the annotation is a `TextualAnnotation` and the underlying object is a node, the value of the `TextualAnnotation` is shown to the top-right of the node in blue.

4.3.5.7 Combining the pieces

Above, we have discussed the nost relevant bits and pieces of our simulator for our technical Petri net type. Next, we show how to combine these bits and pieces into a working application. Listings 4.29 and4.30show the remaining methods implemented in the class `TechnicalSimulator`, which implements the simulator application. Note that the omissions in line 40 indicate the left out methods, which we have discussed in Sect. 4.3.5.3 and Sect. 4.3.5.4 already.

The constructor of this class sets the name, and then creates and adds the action handlers and the presentation handler, which we had discussed above. It also initializes a listener class, which will take care of notifying a user when the end-user modifies the net on which this application is running. But, we do not discuss this class here. The method `getFlatAccess()` provides access to an instance of `FlatAccess` for the net of the application; note that we register the `NetChangeListener` with this instance, since this instance will be notified when the underlying net changes, and it will notify other registered adapters when this happens. But, we don't discuss this in more detail here.

The `initializeContents()` method creates the first net annotation, which represents the initial marking in our case. To this end, it uses the methods which we had discussed earlier: it computes the `initialMarking()` and computes the initial net annotation from it (`computeAnnotation()`), adds it to the application's net annotations and makes it the current annotation.

At last, there are two more technical methods: `isSavable()` indicates that the net annotations of this application can be saved. In that case, the "ePNK: Applications" view will allow the end-user to save and load the current situation of the simulator by enabling the respective buttons in the "ePNK: applications" view. The method `shutDown()` is called when the application is shut down. It must release all resource which the application had acquired. In our case, it is enough to unregister the adapter from the instance of the `FlatAccess` (otherwise changing the net would trigger a notification even after the application shut down).

Listing 4.29: Implementation of the simulator application (part 1)

```
1  package org.pnml.tools.epnk.tutorials.app.simulator.application;

   \\ [...]

   public class TechnicalSimulator extends ApplicationWithUIManager {
6
     FlatAccess flatAccess;
     private NetChangeListener adapter;

     public TechnicalSimulator(PetriNet petrinet) {
11     super(petrinet);
       getNetAnnotations().setName("A simple technical simulator");
       ApplicationUIManager manager = this.getPresentationManager();
       manager.addActionHandler(new EnabledTransitionHandler(this));
       manager.addActionHandler(new InvolvedArcHandler(this));
16     manager.addPresentationHandler(
         new TechnicalAnnotationsPresentationHandler());

       adapter = new NetChangeListener(this);
     }
21
     public FlatAccess getFlatAccess() {
       if (flatAccess == null) {
         flatAccess = FlatAccess.getFlatAccess(this.getPetrinet());
         if (adapter != null) {
26         flatAccess.addInvalidationListener(adapter);
         }
       }
       return flatAccess;
     }
31
     @Override
     protected void initializeContents() {
       NetMarking initialMarking = computeInitialMarking();
       NetAnnotation initialAnnotation =
36       computeAnnotation(initialMarking);
       this.getNetAnnotations().getNetAnnotations().
         add(initialAnnotation);
       this.getNetAnnotations().setCurrent(initialAnnotation);
     }
```

Listing 4.30: Implementation of the simulator application (part 2)

```
  \\ [...]

  @Override
  public boolean isSavable() {
    return true;
  }

  @Override
  protected void shutDown() {
    super.shutDown();
    if (flatAccess != null && adapter != null) {
      flatAccess.removeInvalidationListener(adapter);
      flatAccess = null;
  } }

}
```

In order to plug in the application to the ePNK, we need to imlement one other class: a factory, which can create new instances of the application on a given net. Listing 4.31, shows the implementation of this class. The most important method is `startApplication()`, which creates a new application on the given net. Moreover, there is a method `isApplicable()`, which checks for a given net, whether it would be applicable for that net. In our case, it returns `true` if the type of the net is an instance of our `TechnicalNetType`. But in some cases, this method might also check some other consistency criteria that must be met before the application could be started.

4.3.5.8 Plugging in the application

The last step for making the ePNK run our new application is registering the application factory as en extension to the ePNK. The easiest way to do that is adding an XML snippet to the `plugin.xml` of the project with the simulator. This snippet is shown in Listing 4.32, where the parts indicated in red, can be freely changed. The `class` attribute needs to refer to the fully qualified Java name of the application factory.

Once you did this last step, the application should work with the ePNK. To check this, you should start the runtime workbench of Eclipse again and

Listing 4.31: Implementation of the application factory

```
package org.pnml.tools.epnk.tutorials.app.simulator.application;

// [...]

public class TechnicalSimulatorFactory extends ApplicationFactory {

  public TechnicalSimulatorFactory() {
    super();
  }

  @Override
  public String getName() {
    return "Technical Simulator (Tutorial)";
  }

  @Override
  public String getDescription() {
    return "A technical simulator used in the ePNK tutorial";
  }

  @Override
  public boolean isApplicable(PetriNet net) {
    return net.getType() instanceof TechnicalNetType;
  }

  @Override
  public ApplicationWithUIManager startApplication(PetriNet net) {
    return new TechnicalSimulator(net);
  }

}
```

Listing 4.32: Snippet from `plugin.xml` for plugging in the application

```
<extension
      id="org.pnml.tools.epnk.tutorials.app.simulator.application"
      name="Technical Simulator (Tutorial)"
      point="org.pnml.tools.epnk.applications.applicationfactory">
   <applicationfactory
         class="org.pnml.tools.epnk.tutorials.app.simulator.
application.TechnicalSimulatorFactory"
         description="A simple simulator used as technical example
in the ePNK tutorial">
   </applicationfactory>
</extension>
```

create a Petri net of the new type in the runtime workbench of Eclipse. Open the graphical editor on a page of this net. Then, in the "ePNK: Applications" view, your new application should show up in the drop down menu for applications. If you select it, it should start up on the selected net.

Chapter 5

Installation

This chapter discusses the installation of the ePNK (version 1.0.0). Readers who are interested in getting an idea of what the ePNK is and who do not want to work with the PNK right away can skip this chapter.

5.1 Prerequisites

In order to install the ePNK, you need to have Java 1.6 (or higher) and Eclipse 3.7 (Indigo) or Eclipse 4.2 (Juno) installed on your computer. In this version of the manual, we discuss the installation of the ePNK version 1.0.0 only. For installing other versions of the ePNK or for installing it on other versions of Eclipse, you might find information on the *ePNK installation page*[1].

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

If you are new to Eclipse, it is recommended that you install the *Eclipse Classic* version. Download this Eclipse version for your operating system from `http://www.eclipse.org/downloads/` and extract the downloaded file to some directory; after the extraction, you will find a folder named "eclipse" in this directory, and in this folder, you will find 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 overview of the Eclipse Integrated Development Environment (IDE) at `http://www.vogella.de/articles/Eclipse/article.html`. Once you have installed and started Eclipse, you will find much more information on Eclipse in the "Workbench

[1] `http://www2.imm.dtu.dk/~ekki/projects/ePNK/install-details.html`

User Guide" in the Eclipse help: You can open it via the "Help" menu in the Eclipse toolbar under "Help Contents".

5.2 Installing the ePNK in Eclipse

Once you have installed Eclipse, you can install the ePNK from the Eclipse workbench. To this end, the ePNK is made available via an *Eclipse update site*: `http://www2.imm.dtu.dk/~ekki/projects/ePNK/indigo/update/`

In order to install the ePNK from there to your Eclipse installation, you should proceed as follows (after you have started it and selected a workspace):

1. In the Eclipse toolbar, select "Help" → "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") and the URL

   `http://www2.imm.dtu.dk/~ekki/projects/ePNK/indigo/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. From there, you can select the features of the ePNK you want.

   For working with this manual, you should at least select the following features from the category "ePNK Features":

   - ePNK Basic Extensions 1.0.0
   - ePNK Core 1.0.0
   - ePNK HLPNGs 1.0.0
   - ePNK Tutorial 1.0.0

   If you intend to import high-level nets from other tools than the ePNK, it is recommended that you also install the feature

   - ePNK: HLPNG Label Serialisation (experimental) 0.2.0

   from category "ePNK Experimental Projects".

   If you want to simulate high-level nets, you should also select the feature

- ePNK: HLPNG Simulator 0.1.1

from category "HLPNG Simulator".

You will not need the features from the "ECNO Projects" category, which are a project in their own right (see [16] for more information on the ECNO project). Since they are based on the ePNK, they are deployed from the same update site.

4. After you have selected the features you want, make sure that the box "Contact all update sites during install to find required software" is checked; this will guarantee that all additional features from Eclipse that the ePNK requires will be automatically installed together with the ePNK features (EMF, GMF, Xtext, etc.).

   Then press press okay.

5. Follow through the installation process (don't forget to accept the license agreement).

   Note: If you get an error of the kind

   ```
   Cannot complete the install because one
   or more required items could not be found.
   ...
   ```

   you probably forgot to check the box "Contact all update sites during install to find required software" or have selected a wrong combination of features. In that case, go back and select the right combination as explained above.

6. Then, the selected features of the ePNK and all other required features will be installed; it is a good idea to restart Eclipse after that (Eclipse will ask you to do that anyway).

In case you intend to develop new functions and, in particular, new Petri net types for the ePNK, you might want to install the tools necessary for that purpose already now – while at it. You need to install the "EMF Modeling Framework SDK" and the "Ecore Tools SDK" from the standard Eclipse update site. The details are described in Sect. 3.1.2.

Chapter 6

Experience and outlook

With version 1.0.0, the ePNK has reached a mature state and it should be useful for end users who want to use its graphical editor for creating PNML files and for using the simple function of the ePNK as they are. The ePNK should be even more useful for developers who want to implement new Petri net types and new functionality. In particular, it should be an ideal platform for scientists who quickly want to test new functions and still want a graphical editor that is nicely embedded to an IDE. Actually, we use the ePNK ourselves for implementing the tool support for the Event Coordination Notation (ECNO) [16].

Some of the plans for future extensions of the ePNK are discussed in Sect. 6.2. Before discussing the future plans, we briefly discuss the past in Sect. 6.1: the experiences with developing the ePNK in a model based way and in particular with using EMF, GMF, and some related technologies

6.1 Experiences with MBSE

There are many Petri net tools out there already. Therefore, implementing yet another one needed some additional motivation. When developing the ePNK, this additional motivation was to gain some more experience with the use of EMF and model-based development. To this end, we kept a detailed log of how much time was spent on which parts of the development (up to the first major release of version 0.9.1, the log accounts for minutes).

Eventually, we might break down the time and experiences made in more detail. Here we give an overview of the major steps and the rough time spent on the major parts of the ePNK only:

40h The (roughly) first 40 hours were spent on making a first model of

the PNML core model and implementing the extensions necessary for plugging in new Petri net types and for hooking into the serialisation mechanisms of Eclipse and EMF, so that it could be configured – and on using this new configuration mechanism for implementing the PNML syntax for serialisation instead of standard XMI. Of these 40 hours, about 10 hours were spent on debugging Eclipse's and EMF's serialisation mechanism in order to understand this serialisation mechanism, which unfortunately is not very well documented.

After these first 40 hours, a basic version of the ePNK was working – not supporting HLPNGs yet and without any graphical editor.

20h The next 20 hours were spent on implementing the framework for tool specific extensions – and ignoring unknown tool specifc extensions, as well as on implementing the validation mechanisms and most of the constraints for the PNML core model[1].

50h About 50 hours were spent on implementing HLPNGs, which includes making the models for most of the sorts and operators of HLPNGs, implementing a type system for checking correct typing, and for validating correctness, and implementing parsing and linking functions. These 50 hours include the time spent on adjusting the mechanisms of the ePNK so that it could deal with parsing and linking structured Petri net types. The parser itself is based on Xtext.

Implementing a basic version of HLPNGs with only a few but representative sorts and operators took roughly 20 hours.

80h Implementing the GMF editor in a generic way and properly integrating it with the EMF tree editor, and the parsing mechanisms for structured labels took about 80 hours. This time includes a lot of time investigating and experimenting with different options in achieving this.

30h The remaining 30 hours were spent on implementing some functions as examples (used for the tutorials), on adding some of the last remaining sorts to the definition of HLPNGs, as well as on cleaning up the code and fixing some errors.

[1]This concerns not only the constraints that are explicitly formulated as OCL constraints in the models of ISO/IEC 15909-2; this concerns all the constraints that are stated somewhere in the text of the standard, such as forbidden cycles between reference nodes, etc.

Altogether, it took about 5 $\frac{1}{2}$ weeks working time (spent scattered over about 7 month) to implement version 0.9.1 of the ePNK, which was the first stable version of the ePNK. The core part of the ePNK was implemented in 60 hours. About the same time went into implementing HLPNGs. The implementation of the graphical editor took a major part of the time (80 hours). The reason for that was that GMF itself was not made for building generic editors, and we had to find ways of bringing genericity into GMF – and we had to work around several GMF problems and quirks. Still, using GMF might have saved us a significant amount of time considering the overall functionality that we get for free by a GMF generated graphical editor – and its smooth integration with the Eclipse IDE.

The overall experience was that the parts of the ePNK that concern EMF only worked very smoothly and the use of EMF significantly sped up the development process – for a developer who has some experience with EMF and some of its more advanced concepts already. Working with GMF was more tedious and required much more experience and much more endurance – using the debugger digging in the inner workings of GMF in order to find out how some things work. This is partly due to the fact that GMF was lacking the concepts for generic editors; but, genericity aside, GMF requires much more experience than EMF in order to use it with benefit.

6.2 Future plans

The ePNK can be and is used for different kinds of applications – and it makes it easy to quickly implement new Petri net types and new functions and application. For making these functions more usable and also for easing the fast creation of Petri nets with the ePNK editors, some extension of the ePNK would be useful.

In this section, we give an overview of some extensions that are planned for the ePNK in the future. These extensions concern ePNK's flexibility and ease of use as well as some additional extension mechanisms. The order indicates some priorities, and might be roughly the order in which the features are implemented – but, since nobody is paid for the work, it needs to be seen how things turn out:

- Right now, the ePNK does not "know" the functions and applications that are available for the ePNK. These are plugged in to the Eclipse platform as actions or commands – and initiated by the Eclipse platform. It would be nice if there was an ePNK view that would, for a

selected Petri net, show all the available functions and applications, which could be started from there by a single mouse click.

To this end, an explicit extension point to plug in functions and applications for the ePNK needs to be defined. This extension point could also provide some means to give some meta information about the function or application, saying to which net types it applies and on its characteristics.

Such a plug-in mechanism along with a view for showing the functions and applications available for the selected Petri net will be implemented in a future version of the ePNK.

- Right now, the annotations are shown by marking the elements with a red overlay. Changing this is possible but quite complicated.

  A future version of the ePNK will support a mechanism for applications to provide a presentation descriptor, which defines how annotations should be shown (with some reasonable default implementation). And this descriptor should also be able to define how the end user can interact with the annotated elements and define actions that are initiated by that.

- Right now, adding all the needed labels to a Petri net element of a more complex net types such as HLPNGs with the ePNK editor is quite tedious: First, each label must be created, then the label must be connected to the element, and its type must be selected.

  In a future version of the ePNK, an action will be installed that, e. g. on a double click on a node, will add all the labels required by the Petri net type for that node.

- Right now only the ePNK tree editor will show the correct "dirty flag" after a change of the model. And the PNML document can be saved only from this tree editor.

  In a future version of the ePNK, the "dirty flag" will be updated in all editors that are open on that document and saving (in particular with CTRL-S) will work from any of these editors.

- Right now, the mapping of the model elements of the ePNK and of Petri net types to their XML representation must be programmed – if it needs to be changed. This means programming large tables, which is boring, tedious and error prone.

In a future version of the ePNK, we might define an extension point for such XML mapping that directly takes a table instead of "programming the table".

- Right now, the ePNK comes with its own specific and fixedly implemented concrete syntax for the labels of HLPNGs. Since this syntax is not mandated by ISO/IEC 15909-2, different tools might use different concrete syntax for these labels.

  It would be nice, if different versions of such concrete syntax for labels of HLPNGs could be plugged in to the ePNK, from which the user could choose.

- Up to now, the ePNK supports basic and structured PNML [30] only. Modular PNML is not supported by the ePNK yet – nor is it part of ISO/IEC 15909-2:2011.

  In the long term, the ePNK should support also modular PNML (which was proposed in [30] and some aspects worked out in more detail in [14] already). Implementing the EMF models would actually not be a big issue – implementing the graphical editor of the ePNK so that it works for both PNML as of ISO/IEC 15909-2:2011 as well as for modular PNML is the actual challenge here – and the reason for not having started on this endeavor yet.

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, 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] R. C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. The Eclipse Series. Addison-Wesley Professional, Mar. 2009.

[6] 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.

[7] 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.

[8] ISO/IEC. Systems and software engineering – High-level Petri nets – Part 2: Transfer format, International Standard ISO/IEC 15909-2:2011, Feb. 2011.

[9] K. Jensen and L. M. Kristensen. *Coloured Petri Nets – Modelling and Validation of Concurrent Systems*. Springer-Verlag, 2009.

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

[11] 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.

[12] 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.

[13] 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.

[14] E. Kindler. Modular PNML revisited: Some ideas for strict typing. In *Proc. AWPN 2007, Koblenz, Germany*, pages 20–25, Sept. 2007.

[15] E. Kindler. Model-based software engineering and process-aware information systems. In K. Jensen and W. van der Aalst, editors, *Transactions on Petri Nets and Other Models of Concurrency II: Special Issue on Concurrency in Process-Aware Information Systems*, volume 5460 of *LNCS*, pages 27–45. Springer-Verlag, 2009.

[16] E. Kindler. Modelling local and global behaviour: Petri nets and event coordination. *Transactions on Petri Nets and Other Models of Concurrency*, 6:71–93, 2012.

[17] E. Kindler. Coordinating interactions: The Event Coordination Notation. Technical Report DTU Compute Technical Report 2014-05, DTU Compute, Kgs. Lyngby, Denmark, May 2014.

[18] 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.

[19] E. Kindler and W. Reisig. Algebraic system nets for modelling distributed algorithms. *Petri Net Newsletter*, 51:16–31, Dec. 1996.

[20] E. Kindler, W. Reisig, H. Völzer, and R. Walter. Petri net based verification of distributed algorithms: An example. *Formal Aspects of Computing*, 9:409–424, 1997.

[21] 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.

[22] M. Laganeckas. A simulator for high-level Petri nets: Model-based design and implementation. Master's thesis, IMM-M.Sc.-2012-101, DTU Informatics, Technical University of Denmark, Sept. 2012.

[23] OMG. Meta Object Facility (MOF) specification, version 1.4.1. Technical Report formal/05-05-05, The Object Management Group, Inc., May 2005.

[24] W. Reisig. *Elements of Distributed Algorithms — Modeling and Analysis with Petri Nets*. Springer, 1998.

[25] W. Reisig, E. Kindler, T. Vesper, H. Völzer, and R. Walter. Distributed algorithms for networks of agents. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets II: Applications*, volume 1492 of *LNCS*, pages 331–385. Springer, 1998.

[26] G. Rozenberg. Behaviour of elementary net systems. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and Their Properties*, volume 254 of *LNCS*, pages 60–94. Springer-Verlag, 1987.

[27] P. H. Starke and H.-M. Hanisch. Analysis of signal/event nets. In *Emerging Technologies and Factory Automation (ETFA '97), Proceedings, 6^{th} International Conference on*, pages 253–257. IEEE, Sept. 1997.

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

[29] P. Thiagarajan. Elementary net systems. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and Their Properties*, volume 254 of *LNCS*, pages 26–59. Springer-Verlag, 1987.

[30] 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.

[31] M. Weber, R. Walter, H. Völzer, T. Vesper, W. Reisig, S. Peuker, E. Kindler, J. Freiheit, and J. Desel. DAWN: Petrinetzmodelle zur Verifikation Verteilter Algorithmen. Informatik-Bericht 88, Humboldt-Universität zu Berlin, Dec. 1997.

Index

`AbstractEPNKAction`, *see* ePNK
`AbstractEPNKJob`, *see* ePNK
Action, *see* ePNK
    `AbstractEPNKAction`
active application, **43**
Add default labels, *see* ePNK
Annotation, *see* ePNK, **2**
API, *see* ePNK
Application, *see* ePNK
Applications view, *see* ePNK
Arc, **2**
Arc annotation (HLPNGs), **30**
Arc decoration, *see* ePNK
Association element, *see* ePNK
Attribute, *see* ePNK, **2**, **22–24**, **52**

bend point, **26**
bundled association element,
    see ePNK

Canvas, *see* Canvas
Child element, *see* Eclipse
Command framework, *see* Eclipse
Command handler, *see* Eclipse
Condition, **29**
Constraint, 4
Constraints, *see* EMF, *see* ePNK
context independent element,
    see ePNK
Convenience classes, *see* ePNK
Curved arc, **27**

Declaration, **29**
Decoration, *see* GMF
Development workbench, *see* Eclipse
Diagram information, *see* ePNK
Dialog, *see* Eclipse
Dot net, *see* ePNK

`eAllContents()`, *see* EMF
Echo algorithm, 51
`eClass()`, *see* EMF
Eclipse, **9–12**
    Action, 63
    Child element, **15**
    Command framework, **88**
    Command handler, 63
    Default editor, **11**
    Development workbench, **59–60**
    Dialog, 63
    Editing domain, **88**
    Editor, **11**
    Extension, **69, 70**
    Extension point, **70**
    Feature, **156**
    IDE, **9–12**, 58, 231
    Installation, **231**
    `ISelectionListener`, **66**
    Job, **40**, 63
    Menu bar, **9**
    Outline view, 11, **12**
    Package explorer, **10**

Perspective, 9, **12**
Plug-in Development perspective, **59**
Plug-ins view, **59**
`plugin.xml`, **69**
Problems view, **12**, 17, 18
Progress indicator, 40
Progress view, 40
Properties view, 11, **12**
Resouce, **14**
Resource, **69**, **88**
Resource set, **69**, **72**, **88**
Runtime workbench, **59**–**60**
Selection listener, 66
Tool bar, **9**
Update site, **156**
View, **12**, 63, **65**–**66**
`ViewPart`, **66**
Wizard, 63, **71**
Workbench, **9**, 10
`eContainer()`, *see* EMF
Ecore model, *see* EMF
Edit code, *see* EMF
Edit part, **89**
Editing domain, *see* Eclipse
EMF, 63
`AbstractModelConstraint`, **125**
API (generated from model), **86**–**87**
batch constraint, **114**
constraint provider, **114**
Constraints, **112**–**115**
Container, *see* EMF
Container class, **130**
`eAllContents()`, **87**
`eClass()`, **87**
`eContainer()`, **87**
Ecore model, **86**
Edit code, **109**
`EObject`, **87**
`eResource()`, **88**
Factory, **72**, **87**
Generator model, **109**
`getContents()`, **69**, **88**
`getContents`, **72**
`getResource()`, **69**
`getResourceSet()`, **88**
getter method, **69**, **86**
live constraint, **114**
Model code, **109**
Object class, **130**
Package class, 128
Reference, **86**
`save()`, **72**
`save()`, **88**
setter method, **69**, **86**
`validate()`, **125**
Validation, **114**
Empty (net type), **28**
`EmptyType`, *see* ePNK
`EObject`, *see* EMF
ePNK
`AbstractEPNKAction`, **77**, **93**, **152**
`AbstractEPNKJob`, **78**, **93**, **152**
Action handler, **97**
Add default labels (menu), **22**
Annotation, **62**, **94**–**96**, **153,** **154**
API, **69**, **72**–**76**
Application, 6, **37**–**38**, **41**–**44**, **57**, **63**, **93**–**104**, 105, **153**, 154
Applications view, 37, **41**–**44**
Arc decoration, **141**
`ArcFigure`, **141**
Association element, **133**, **151**
Attribute, **62**, **115**–**117**
`Attribute`, **115**

bundled association element, **135**
`canCreateObject()`, **130**
Canvas, **19**
Clear image cache, **27**
Constraints, **124–128**, **151**
context independent element, **135**
Convenience classes, **92–93**, **151**
`createAttributeObject()`, **132**
`createObject()`, **130**
`createToolInfo()`, **148**
Customizing graphics, **153**
Data types, **150**
Decorations, **153**
Developer, **6**
Diagram information, **151**
Dot net, **28**
`EmptyType`, 62
Factory, **130**
Figures, **153**
`FlatAccess`, **82**, **92**, **151**
Function, **37–38**, **63–86**
`getFlatAccess()`, **92**
`getglobalLinks()`, **123**
`getLinker()`, **123**
`getRefFeature()`, **123**
`getStructuralFeature()`, **122**
`getStructuralFeature`, **124**
`getSymbolDef()`, **124**
`getSymbolUses()`, **124**
Graphical editor, 14, **19–27**
graphical features, **25–27**
`GraphicalExtension`, **143**
HLPNG, **28**, **154–155**
ID, **63**
`ID`, **123**
id, 14, **18**
`IGraphicalExtension`, **146**
Image cache, **27**
Installation, **232–233**
Interaction description, 105
`IPNMLFactory`, **130**
Java constraint, **124–128**
Label, **62**
`Label`, **108**
Label proxy, **150**
`Linker`, **123**
Net annotation, **101–104**
Net label, **24**
`NetFunctions`, **93**, **151**
Object annotation, **101–104**
Page label, **24**, **138**
Page label proxy, **150**
`parse()`, **122**
Petri net type, **91–92**
Petri net type definition, **105–138**
`PetriNet`, **108**, **122**
`PetriNetType`, 62, **106**, **122**
`PetriNetTypeExtensions`, **74**
PNML core model, **61–63**
`PnmlcoremodelFactory`, **72**
PNTD, 151
PNTD extension point, **110–112**
Presentation description, 104
Presentation handler, **96–97**
PTNet, **28–29**
`PTNet`, **105–115**, 152
Reference place, 25
Reference transition, 25
`registerExtended PNMLMetaData()`, **128**
`resolve()`, **92**
`showLabelOnPage()`, **138**
standard feature, **137**
structured label, **122–124**

structured Petri net type, **122–124**, **150**
`StructuredLabel`, **122**
`StructuredPetriNetType`, **123**
Symbol, **119**
Symbol definition, **122**
Symbol use, **122**
`SymbolDef`, **121**, **123**
`SymbolUse`, **123**
`SymbolUse`, **121**
`SymbolUseMapping`, **123**
Symmetric net, **28**
`text`, **108**
Tool specific extension, 151
Tool specific information, **146–149**
`ToolInfo`, **147**
`ToolspecificExtension Factory`, **148**
Tree editor, 13, **14–19**
Type registry, **74**
`Unparsed`, 55
`update()`, **141**, **143**
Update site, **232**
User, **6**, 9
Validation, **17–19**, 22, 151
XML mapping, **128–137**
`eResource()`, *see* EMF
Extension, *see* Eclipse
Extension point, *see* Eclipse

Factory, *see* EMF, *see* ePNK
Feature, *see* Eclipse
Figure class, *see* GMF
`FlatAccess`, *see* ePNK
Flattening (of a net), **25**
Function, *see* ePNK

Generator model, *see* EMF
genmodel, *see* EMF: Generator model
`getContents()`, *see* EMF
`getglobalLinks()`, *see* ePNK
`getLinker()`, *see* ePNK
`getRefFeature()`, *see* ePNK
`getResource()`, *see* EMF
`getResourceSet()`, *see* EMF
`getStructuralFeature()`, *see* ePNK
getter method, *see* EMF
GMF
Connection, **141**
Decoration, **141**
Figure class, **139**
`fillShape()`, **143**
`outlineShape()`, **143**
`setForeGroundColor()`, **141**
`setLineStyle()`, **141**
Graphical features, *see* ePNK

High-level net schema, 44
High-level Petri net graph, *see* HLPNG
High-level Petri net schema, *see* HLPNGS
High-level Petri nets, **28**
HLPNG, *see* ePNK, **9**, 14, **29–37**
HLPNG Label Serialisation, **30**, **37**
HLPNGS, **48**

ID, *see* ePNK
id, *see* ePNK
IDE, *see* Eclipse
Image, **27**
Image cache, *see* ePNK
Integrated Development Environment, *see* IDE
intermediate point, **26**
`ISelectionListener`, *see* Eclipse

Job, *see* Eclipse, *see* ePNK
`AbstractEPNKJob`, **76–86**

Label, *see* ePNK, **2**, 19, **20–22**, **62**
Simple, **21**
Structured, **22**
`Label`, *see* ePNK
Label proxy, *see* ePNK
`Linker`, *see* ePNK

Marking, **29**
Menu bar, *see* Eclipse
Minimal distance algorithm, 48
Model checker, 38–40, 153
Model code, *see* EMF

Name, *see* Petri net
Net annotation, *see* ePNK
Net label, *see* ePNK, 109, **122**
`NetFunctions`, *see* ePNK
Network algorithm, 48
Network editor, **50**
Network model, 50
Node, **2**

Object (of a Petri net), **2**
Object annotation, *see* ePNK
OCL constraint, 4
Operator, **29**
Outline view, *see* Eclipse

P/T-System, **105–115**
P/T-system, **4**
P/T-Systems, **28**
Package class, *see* EMF
Package explorer, *see* Eclipse
Page, **2**, **24–25**
Page label, *see* ePNK, **19**
Page label proxy, *see* ePNK
`parse()`, *see* ePNK
Perspective, *see* Eclipse
Petri net
Name, **3**
Petri Net Markup Language, *see* PNML
Petri net type, *see* ePNK, 2
Petri net type definition, *see* ePNK, *see* PNTD
`PetriNet`, *see* ePNK
`PetriNetType`, *see* ePNK
Place, **2**
Place/Transition-System, *see* P/T-System
Plug-in Development perspective, *see* Eclipse
Plug-ins view, *see* Eclipse
PNML, 1, **2–6**
XML format, **5–6**
PNML Core Model, **150**
PNML core model, *see* ePNK, **2–4**, 69, 90–91
PNML Document, **13**
`PnmlcoremodelFactory`, *see* ePNK
PNTD, 1, **4**, **27**
PNX Document, **13**
Polyline arc, **27**
Problems view, *see* Eclipse
Properties view, *see* Eclipse
PTNet, *see* ePNK
`PtnetFactory`, *see* ePNK

Reference, *see* EMF
Reference place, *see* ePNK, **2**
Reference transition, *see* ePNK, **2**
Resource, *see* Eclipse
Resource set, *see* Eclipse
Runtime workbench, *see* Eclipse

`save()`, *see* EMF
SE-Nets, **115–119**
SE-nets, 22, 37, **154**

setter method, *see* EMF
Sieve of Eratosthenes, 44
Signal event nets, *see* SE-nets
Signal/Event-systems, 37
Simulation view, **44**
Simulator, **44**–**51**
    Back button, **46**
    for network algorithms, **48**
    Forward button, **46**
    Pause button, **46**
    Play button, **46**
    Simulation speed, **46**
    Stop button, **46**
Simulator (P/T-nets), **41**
Smoothness (of an arc), **27**
Sort, **29**
standard feature, *see* ePNK
structured label, *see* ePNK
structured Petri net type, *see* ePNK
`StructuredLabel`, *see* ePNK
`StructuredPetriNetType`,
        see ePNK
Sub-page, **24**
Symbol, *see* ePNK
Symbol definition, *see* ePNK
Symbol use, *see* ePNK
`SymbolDef`, *see* ePNK
`SymbolUse`, *see* ePNK
Symmetric net, *see* ePNK

Token positions, **152**
Tool specific information, 1,
        see ePNK
Toolbar, *see* Eclipse
`ToolInfo`, *see* ePNK
Transition, **2**
Type, **29**

Update site, *see* Eclipse, *see* ePNK

Validation, *see* EMF, *see* ePNK
Varialble, **29**
View, *see* Eclipse
`ViewPart`, *see* Eclipse

Wizard, *see* Eclipse
Workbench, *see* Eclipse

XML mapping, *see* ePNK