A universal PNML Tool

Lukasz Zoglowek

Kongens Lyngby 2008
The PNML (Petri Net Markup Language) is an interchange format for Petri nets, which is currently being standadized. PNML is not restricted to a particular kind or version of Petri net, but can be used for every Petri net type. To this end, PNML allows the definition of new Petri net types.

The definition of PNML makes use of Model-based Software Engineering technology, and there is an implementation of PNML based on EMF, which is called the PNML Framework. The PNML Framework, however, needs to be recompiled whenever a new Petri net type is defined, and it provides an API only. It does not come with a graphical editor.

In this thesis, a universal PNML tool is developed in which new Petri net types can be plugged in without recompiling the tool for every new type. Moreover, the universal PNML tool should come with a graphical editor that allows to create and edit Petri nets of any (plugged-in) type. Furthermore, the universal PNML tool should be able to interact with applications running on that Petri net, and to allow the applications to visualize the results in the graphical editor.

For some features of new Petri net types, there are specific graphical representations. In order to allow the universal PNML tool to represent these features in this way, the type concept of PNML types needs to be enhanced with a concept for defining the graphical appearance of some features of a new Petri net type. The concepts for defining these appearances should be designed in this thesis and implemented in the universal PNML tool.
This master thesis was prepared at Informatics Mathematical Modelling, the Technical University of Denmark within the period March 2008 - August 2008.

The goal of this document is to present the complete analysis, design and implementation of universal Petri net tool. The thesis will present the reader the main requirements and problems for constructing the flexible solution for designing Petri nets. Furthermore, the report is going to show how the modeling technologies can leverage building of PNML application. Finally, upon gathered specification the proposal for design and implementation of universal PNML tool will be given.

The CD is also included which consists of executable and source code.

Lyngby, September 2008

Lukasz Zoglowek
Acknowledgements

I would like to thank my supervisor Ekkart Kindler who was very helpful with guidance, advices and delivering constructive criticism.
Contents

Summary i

Preface iii

Acknowledgements v

1 Introduction 1

1.1 Motivation ........................................ 2

2 Background 5

2.1 Models in software development ..................... 5

2.2 The Eclipse platform ................................ 10

2.3 The Eclipse Modeling Technologies .................... 12

2.4 The ISO/IEC 15909 standard ........................ 17

2.5 PNML Concepts ..................................... 17

2.6 The PNML Framework ............................... 24
3 Analysis and design of the PNML tool 29
  3.1 Analysis ........................................... 29
  3.2 Design ............................................. 42

4 Implementation 55
  4.1 Extending generated code .............................. 56
  4.2 PNML Serialization ................................ 57
  4.3 Loading Petri net types .............................. 64
  4.4 Editor support for Petri net types ................. 71
  4.5 Graphical extensions .............................. 80
  4.6 Conclusions ........................................ 80

5 Future work 83

6 Related Work 87
  6.1 Petri Net Kernel .................................. 87

7 Conclusions 91

8 Glossary 95

A The GMF Notation meta-model 97

B Creating Petri net type - Step by step example 99
Petri nets are a mathematical formalism, which is used to represent discrete distributed systems [13, 27, 28]. The name originates from Carl Adam Petri, who in his dissertation in 1962 [24], started the discussion about a general architecture for asynchronous distributed systems. Petri nets, as one knows them nowadays, were introduced three years later in Petri’s talk ”Fundamentals on the description of discrete processes”[25].

The range of applications for Petri nets is very wide, they are usually used for studying concurrent, asynchronous, non-deterministic and stochastic systems. Petri nets allow designing systems on a high level of abstraction, which makes them implementation independent. Petri nets are frequently used in the early stages of software development. Mostly they are used as prototypes which can be utilized to test and prove various requirements for a system. This can include proofs for deadlock freedom and verification for different properties in the network.

Due to the extensive applications of Petri nets, many different specialized Petri net types (PN types) were developed. The basic structure of Petri net consists of places, transitions and arcs. A Petri net type is a variant of a Petri net, which augments its basic structure with specialized elements. These elements can include simple text annotations, types or algebraic expressions. These low level formalisms were abstracted by defining so called High-Level Petri nets.
High-Level Petri nets added the concepts of complex data types and allowed using algebraic expressions for annotating network nodes. The concrete examples (identified and thoroughly described in [18]) of High-Level Petri nets include Coloured Petri Net [17], Predicate-Transition Nets [14] and Algebraic Petri Nets [26]. Apart from them there exist many other types which take into consideration time, complex hierarchies, capacities etc.

Unfortunately, the diversity of Petri nets types created many incompatibility issues. The ISO standard ISO/IEC 15909, which is currently under development, is going to unify the technology for Petri nets. More specifically, according to [15], this standard will:

1. allow developers/engineers to use the same terminology
2. ease the data exchange through common unambiguous XML format: the PNML (Petri Net Markup Language) The PNML structure is defined in Part 2 of the standard
3. enable developing extensions for well defined basic Petri net types (defined in UML). This is mostly the goal of Part 3 of the standard

More information about the PNML technology will be covered in Chapter 2.5.1. The standard and its different parts is described in Chapter 2.4.

1.1 Motivation

The motivation for this thesis is to create a universal PNML tool. A PNML tool is a software which is the realization of the Petri Net ISO/IEC 15909-2 standard. The end users of this software are tool developers (Petri net experts) and Petri net designers. The Petri net designers aim to model Petri net of various Petri net types. The Petri net experts, on the other hand, requires the possibility of Petri net types customization and adding new types. A universal PNML tool must therefore support many Petri net types and it must be easily extensible with new types. Moreover, adding new types should be very simple for the users, without even requiring programming skills.

1.1.1 High-level goals

The main features of the PNML tool are (partially mentioned in [16]):
1.1 Motivation

1. Allow end-users to manipulate Petri net models. This means that the PNML tool should provide graphical editor which allows developer constructing Petri nets of different types.

2. Enable developers to create and plug-in their own Petri net types. The PNML tool should ease the tool developers in defining new types without deep knowledge of the programming details. Furthermore, all editors should be aware of any new extensions and automatically adjust its modelling tools and properties. What is more, the tool developers should be able to customize the graphical appearance of nodes and connections for the new Petri net types.

3. Saving and loading models to and from the PNML format. Persisting models to PNML must be also possible for documents which use new Petri net types defined by tool developers.

The work presented in this thesis will concentrate mostly on the implementation of ISO/IEC 15909 Part 2. However, part of the thesis will deal also with plugging-in new Petri net types. The thesis will study concepts and implementation issues of this problem. Therefore, it will help advancing the development of the Part 3 of the standard.

1.1.2 Existing solutions

An implementation of the Petri net standard supporting some of above-mentioned features already exist and is called the PNML Framework [16].

The PNML Framework is an API library developed using the Eclipse Modeling Framework (EMF), which comes with easy to use PNML import and export capabilities. Apart from it, the PNML Framework includes an EMF meta-model for Place/Transition nets (which conforms to the UML meta-model defined in the ISO/IEC 15909 standard). This meta-model can be used to generate simple tree-based Petri-net editor in Eclipse. More advanced graphical editors can also be generated, but some knowledge of Graphical Modeling Framework (GMF) is required.

Unfortunately, the PNML Framework is not easily extensible. If one would like to extend or create a new Petri Net type, according to [16], not only the new meta-model needs to be created but also the code needs to be re-generated and re-compiled. Obviously, any compile time errors would need to be resolved by developers. What is more, some extensions may require modification of the code generation templates, which could be very complex, time-consuming and
error-prone task. The detailed description of the PNML Framework is covered in Section 2.6.

Having mentioned the problems with the PNML Framework the new approach for a universal PNML tool is proposed. The incentives for creating this universal PNML tool are the following:

1. Allow easy plugging-in new Petri net types. This means that new types can be loaded as UML models without re-generation and re-compilation of any component. In other words, the developer would not need to program anything to create a new Petri net type.

2. Enable easy diagram modeling by providing an adaptable editor. The new tool won’t be only an API but the complete graphical editor. Consequently, the user will not need to generate and compile it first. The graphical editor will be very flexible, adjusting its modelling features to Petri net type of any Petri net document. Furthermore, the new tool will provide a way to plug-in graphical artifacts, which will be used to represent Petri net nodes and arcs.

Finally, the new PNML solution should be platform independent and implemented using mature tools and reliable design techniques.
Chapter 2

Background

This chapter introduces the principal concepts, which create the basis for developing the PNML tool. Firstly, the definitions of models and modeling are described and the dominant benefits over traditional code-based software development are shown. The PNML tool will be developed in the Eclipse platform which has very rich support for modeling technologies (so called Eclipse Modeling Technologies - EMT). The modeling frameworks will significantly leverage the amount of work needed to implement a PNML editor. What is more, models will be used for defining and exchanging the Petri net types.

Secondly, detailed description is presented for PNML concepts, which are formally enclosed in the Petri net ISO standard - Part 2([9]). In the ISO document Petri net models are defined as UML diagrams. It will be shown how existing projects (The PNML Framework) utilizes the Eclipse modeling techniques to represent these concepts.

2.1 Models in software development

Software development in particular of business-scale solutions has never been an easy task. Many of the difficulties lie in the high complexity of these kind of
Background

Software developers need to deal with number of problems including: understanding broad problem space, constraints for time and technology and constant changes in system’s requirement and design. What is more, big projects require involvement of many teams which causes usually a lot of management and scheduling issues.

There are many techniques to tackle some of the above-mentioned problems. One of the most common is using the notion of a model and a process called modeling. The model is a way of hiding low level details of physical system and consequently presenting only relevant concepts to engineers. Therefore, models firstly, help in understanding the real-world systems. Secondly they ease in communication among engineers by providing the common notations and properties. Depending on the context different level of details represented by models, may be important for developers. Due to this fact, the model transformations are quite common which can convert among various model views (levels of abstractions).

Using modeling and model transformations approaches is usually referred to Model Driven Development (MDD) or Model Driven Engineering (MDE).

2.1.1 Models and Modeling

The process of modeling may adopt different forms with regard to interaction between a model and a code. Those forms are keyed out in the section to give an overview and show some threats when using concrete approaches.

Figure 2.1 illustrates the current practices used in model-driven development (MDD). The first and most commonly used is Code Only. For this form it is questionable to call it MDD as models do not exist formally. The models, represented usually as architectural diagrams, are informal and sometimes even not documented. This means that there is almost no coupling between a code and high level models. This cause problems especially with the software evolution. The code undergoes constant changes while architectural models remains usually not updated. This fact makes it much more difficult in understanding complex details of business logic.

The second type of modeling practice is defined in [10] as Code Visualisation. Nowadays, many of development environment (for example Eclipse, Visual Studio) provides the way of extracting the system architecture from the source code. Usually, architectural models are represented as UML-like class or sequence diagrams. Furthermore, some of the tools allow run-time synchronization between model (diagram) and source code view. This enables that some changes made
A roundtrip engineering (RTE) is a type of development practice, where transformations for models and code happens constantly in both directions. Usually the process starts with some high level design for which the code is generated. After that, the implementation is changed according to system requirement and transformed to model again. Next the model is refined and the process is being repeated. This technique has the inimitable advantage that the model and the source code are consistent with each other. IBM Rational Rose is one of the tool, which support the roundtrip engineering.

Model-centric solution relies on generating source code from well-defined models. These models, may include very specific information about code generation for example: business logic, presentation and persistence of elements. Most frequently, generation is performed using code-templates, which can be adjusted by software engineers. For model-centric solution the model is the main artifact that is being changed. This solution is adopted in the Eclipse Modeling Framework (EMF) and will be mainly used in the implementation stage of the PNML tool.

Model-only approach concentrates mostly on understanding specific problem domain. The main application of this technique is used in prototypical or early analysis stage of software systems.
2.1.2 Model Driven Architecture

During development of the PNML tool different models will be designed. In order to understand the hierarchy of various models and transformation among them this section will give an overview of Model driven architecture (MDA).

MDA is the style of model driven development which defines guidelines of how to specify the models, notations and transformation rules. MDA concepts are standardised by the Object Management Group (OMG). It provides the foundation for system interoperability by introducing following modeling standards: Unified Modeling Language (UML), Meta-Object Facility (MOF) and Common Warehouse Metamodel (CWM). As pointed out in [11] the main MDA design principles are the following:

- Models defined using concrete and formal notation are the key for understanding complex applications
- Systems should be build around set of models, which are imposed by a number of model transformations. These transformations should be organised into layers.
- Describing models in a set of meta-models which allows integration and transformation among models
- Model-based concepts requires international standards

The main goal of MDA platform is to decouple design from architecture. The design should reflect only functional requirements for the system, while architecture should capture more technical concepts like: scalability, reliability and performance.

In order to support these principles OMG proposed the set of layers and transformation for MDA which is presented in Figure 2.2. Computation Independent Model (CIM) is used to capture only very high level business related domain elements without showing details about structure of the system. It should be modelled by domain experts (business experts). Platform Independent Model (PIM) represents the constructions of the system, however, without dealing with implementation details. The next stage of view is Platform Specific Model (PSM). It combines the specification of PIM together with Platform specific information. Examples of PIM may include models concrete to technologies like: CORBA, J2EE etc. The most detailed models called Implementation Specific Models (ISM) refers to run-time and implementation models.
The PNML models (which are presented in Section 2.5) are situated as Platform Independent Models in depicted hierarchy. The PNML models are not pure CIM models as they are defined using UML, which captures much design information about the system. On the other hand, the models from the PNML Framework presented in Section 2.6.2 will be categorized as Platform Specific Models as they are concrete to Eclipse Modeling Technology. The Section 2.6.2 will also show that the transformation from PIM to PSM may not be always straightforward as there may be some model elements or relationships not easily defined in specific platforms.

![Layers of Model Driven Architecture](image)

Finally it is worth to mention the importance of model transformation. The OMG specification its own standard language for transformation called the QVT (Queries, Views, Transformations).

### 2.1.3 The Unified Modeling Language

As already mentioned the Petri net ISO standard defines PNML models using UML models. The Unified Modeling Language (UML) is a standard defined at the Object Management Group (OMG) which is a visual specification language for object modeling. The UML was invented to provide a unified framework for specification, modeling and documentation of software processes.
The UML was created in 1994 by Grady Booch and Jim Rumbaugh to define a common diagramming notation for their methods - the Booch and Object Modeling Technique (OMT). The OMG adopted the UML in 1997 and continued development and refining the standard. In the recent years, the UML got very popular and became an industry standard. It is important to point out that the UML is general purpose modeling language and it is not constraint only to software engineering. The UML has been successfully applied to business processes, organisational modeling and even hardware design.

The UML system model can be represented using different views called the UML diagrams. These diagrams can be categorized as:

- Functional requirements. These diagrams illustrate the requirements, goals and users of the system. The most commonly used diagram are use-case diagrams.

- Static structural. Class, component and deployment diagrams are example of illustrating the structure of the system with respect to different perspectives. The PNML models are described using UML structural class diagrams. There are presented and described in Section 2.5.

- Dynamic behavior. Dynamic behavior emphasizes the actions which may happen in the system. The diagrams which model that behavior are following: sequence, activity, state charts diagrams etc.

Although, the UML is broadly used, it is also being criticized. The main problems are the language complexity and number of redundant constructs on the diagrams. It is therefore difficult to learn and to adopt.

The current version of the UML standard is 2.1.2 [23].

2.2 The Eclipse platform

The Eclipse is the software platform which consists of many application frameworks, tools and runtimes for software developers. Its main use is as an integrated development environment (IDE). The Eclipse is an open-source software written primarily in Java. Furthermore the Eclipse Platform is a Rich Client Platform (RCP), which means the developers can easily extend the development environment and build their own applications based on an existing platform.
2.2 The Eclipse platform

The PNML tool will be developed in the Eclipse environment. The reasons for choosing specifically this platform are the following:

- The Eclipse through its plug-in architecture is very easily extensible. Consequently, it is easy to add a new type of editor for Petri net and plug in various extensions to it (for example new Petri net types). The details of Eclipse plug-in architecture are described in the next sub-section.

- Support for modeling frameworks. The Eclipse Modeling Technologies (MDT) are well developed frameworks, which allow generating code for models, plug-ins, editors etc. More information about MDT is covered in Section 2.3.

- Eclipse is an open-source project with huge community working on it and extending it constantly.

2.2.1 The Eclipse Plug-in architecture

As described in the high-level goals for the PNML tool (Section 1.1), the software must be easily extensible with new Petri net types. In order to achieve that goal the Eclipse plug-in architecture will be utilized.

The Eclipse is an extensible platform which allows contributing to it by building sets of pluggable components, called plug-ins. The basic Eclipse plug-in mechanism is that each plug-in can add processing elements to another plug-in. Except some small run-time classes every feature in Eclipse is built as a plug-in. This makes the extension to Eclipse equal with any other default features.

![Image of the Eclipse plug-in support](image-url)
Figure 2.3 depicts the typical situation when a new plug-in extends some existing plug-ins. The Workbench and Workspace are the essential and required plug-ins in Eclipse and serves as extension points to new plug-ins. The extension point in Eclipse terminology is a set of slots that other plug-ins can extend and consequently augment the functionality of the host-plug-in. For example the Workbench serves as extension point for modifying the user interface. The Workspace plug-in, on the other hand, consists of extension points for interacting with files and resources. The Workbench and Workspace are just examples of built-in plug-ins. However, a user-defined plug-in can also define new extension point for other plug-ins. All extension points are defined as XML documents.

2.3 The Eclipse Modeling Technologies

The Eclipse Modeling Technologies (MDT) are collections of opens-source tools and frameworks to support model driven engineering within the Eclipse platform. MDT are at the moment one of the biggest modeling solutions available for developers. The direct competitor for it is the DSL Toolkit integrated in Microsoft Visual Studio. However, due to the fact that Eclipse architecture is much more extensible and open-source it was decided to use MDT instead of DSL tools from Microsoft. Furthermore, the PNML Framework (described in Section 2.6) makes use of MDT so it will be possible to re-use some of the PNML models.

The goal of Eclipse Modeling Technologies is to provide tools and support in the following areas:

- **Abstract Syntax Development** The framework supports editing, testing, validating, and refactoring models.

- **Concrete Syntax Development** The MDT has specific tools for creating textual and graphical concrete syntax for abstract syntax. This includes editors for general-purpose languages (UML) and domain specific languages (DSL).

- **Model Transformation** The MDT supports many industrial standards, in particular the OMG’s Query, View, Transformation (QVT) Specification

- **Supporting Industry Standards** Lastly the MDT supports bunch of open standards. The main includes OMG Standards (MOF, UML, MDA, XMI, QVT), XML Schema Definition (XSD) and many more.
Here the details of each modeling framework will be described.

2.3 The Eclipse Modeling Technologies

2.3.1 The Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) is a framework and code generation facility which allows for creation of tools and applications based on a structured model [12]. It will form the base for implementation the PNML tool. Therefore, this section will describe the basic of this framework.

The instance models in EMF are designed using metamodels, which uses similar notation to UML Class Diagrams. More specifically EMF metamodels are called Ecore. Ecore originates from Meta Object Facility (MOF) standard defined by OMG. However, comparing to MOF, Ecore uses only core subset of core MOF API. After Ecore architecture was introduced MOF was still evolving. In 2006 OMG standardised variant of MOF called EMOF (Essential MOF) which is (apart from some naming convention) very similar to Ecore.

For code generation EMF requires Ecore metamodel as an input. It can be designed using simple UML diagram tool provided in EMF or imported from external UML tool (for example Rational Rose). What is more, Ecore document can also be generated from XML Schema document and specially annotated Java classes (more details about this technique can be found in [2] or [12]).

Once the Ecore metamodel is set EMF can generate corresponding code. Apart from simple class structure, generated code offers also other facilities like:

- Model change notification mechanism
- Persistence implementation - by default EMF serialize (that is stores objects to file) all documents using XMI (XML Metadata Exchange) standard (it is also used for persisting ECore models). However, if the model is defined using XML Schema, EMF will persist all object as an XML document conforming to that schema.
- Model validation framework. It provides support for defining constraint providers for EMF models. Apart from it, it is possible to plug-in model traversal algorithm, constraint parsers, validation listeners and many more. Details of EMF Validation Framework are described in [3]
- Reflective API, which is used to manage objects generically

The above-mentioned features are provided by the core EMF framework. Apart from it the EMF includes the EMF.Edit framework. EMF.Edit extends the
framework by adding adapter classes which allows viewing and editing the model instance. The simple editor is a tree-based model editor and supports many standard commands like: undo-redo and copy-paste. Furthermore, EMF.Edit adds support for displaying property sheet information for specific model elements. What is more, the generated code is in the form of an Eclipse plug-in, which conforms to the standard way of extending the Eclipse Platform.

EMF and EMF.Edit implement the Model View Controller (MVC) design pattern, where the EMF generated code acts as a model and EMF.Edit serves as a controller. The connection between model and controller is possible due to generated by EMF notification mechanism.

The generated code for EMF and EMF.Editor can be customized by, firstly, modifying numbers of different flags on the model. The details of generation control flags one may find in [2]. Secondly, the generated code can be edited by hand and then subsequent code-regeneration will preserve the customer code. Thirdly, the default EMF templates for code generation can be modified and adjusted for concrete problem.

2.3.2 The Graphical Editing Framework

EMF and EMF.Edit can be used to produce tree-based PNML editor. However, the requirement for the PNML tool is to build a graphical tool. The reason is that it is much easier to visualize the Petri net as graphs then trees. The Graphical Editing Framework together with the Graphical Modeling Framework with will be utilized to ease the development of a graphical editor.

The Graphical Editing Framework (GEF) [22] enables developers to create rich and consistent graphical representations and editing environment for existing models. Draw2d framework, which is standard 2D drawing framework based on SWT, is used for all graphical visualizations. The graphical editors can be build for nearly every model and allows for easy model modification (changing structure of the model, modifying properties etc..). These modifications can be handled using common functions like drag and drop, copy paste and commands from toolbars or menus.

Draw2D is the lightweight graphical framework hosted in SWT canvas heavyweight control. Lightweight means in this situation that Draw2D elements do not require separate handles from operation system. The advantage is that the structure of many Draw2D elements is seen to OS as one heavyweight SWT component. This solution uses much less resources, especially when many elements needs to be displayed. Furthermore, Draw2D is also more flexible as it
allows creation of arbitrary shaped objects in windowing system, which usually are restricted to rectangular components.

Figure 2.4 illustrates high level view on GEF framework. Generally speaking, GEF framework provides the link between the model and the view. What is more, the framework consists of tools, which capture the input events and transform them into specific requests. Requests and Commands are the main concepts for communications in GEF, they encapsulate interactions and the effect on the model.

Figure 2.4: High level view on GEF [4]

2.3.3 The Graphical Modeling Framework

The Graphical Modeling Framework (GMF) is the framework used for developing visual design and modeling platforms in Eclipse [6]. The main goal of GMF is to reduce the gap between EMF and GEF. The input to the framework is the domain model linked with diagram definition. As an output GMF will generate complete visual editor implemented as a plug-in in Eclipse.

Figure 2.5 shows the cycles during GMF development. The first three core models are: domain model (developed in EMF), graphical definition and tooling definition model. Graphical definition model contains information related to graphical artifacts, which will be shown as a view of the domain model. The
one important fact about graphical model is that it is completely disconnected from the domain model, therefore, it can be shared across different projects. Tooling definition, on the other hand, specifies the design of menus, toolbars, elements on the palette etc.

The next step is defining the mapping among the three mentioned models. After that, the GMF transforms the mapping model into a generator model, which includes implementation specific details. Finally the code for plug-in based graphical editor can be generated.

Generated editor supports following features: Properties window, Palette, Overview, Zoom, Navigator, Outline, Keyboard bindings, Drag And Drop, Layout and Filter views.

Figure 2.5: GMF-based development overview [7]

GMF is very powerful framework which can generate a lot of difficult and error-prone code. The intermediate allows for many customizations in the generated editors. Unfortunately, the major problem with the GMF at the moment, is the lack of documentation. The GMF Project website offers only some tutorials and examples. However, in order to deepen the knowledge about GMF, one needs to perform a lot of experiments and examine generated code.
2.4 The ISO/IEC 15909 standard

Having described the most fundamental technologies which will be used for implementing the Pnml tool, the concepts concerning the standard and especially PNML will be presented.

As mentioned in the introduction the ISO/IEC 15909 standard for Petri net is currently under standardisation process. The standard consists of three main parts:

1. **Part 1**
   This part [8] is already international standard and was published in December 2004. It describes the semantic model, graphical model and the mapping between them for High-Level Petri Nets. It includes also the basic conventions for High-Level Petri Net Graphs (HLPNGs).

2. **Part 2**
   Provide the description of exchange format for Petri nets called PNML. It includes the core, basic elements which are common to all Petri nets and their corresponding PNML syntax. This part also includes description of two concrete syntax Place/Transition Nets and High-Level Petri Nets. This part will become international standard in by the end of 2008.

3. **Part 3**
   The last part defines how the Petri nets can be extended. The part, however, is currently under development and will be finished when PNML standard will be fully defined. The document will focus on formalising the Petri Net Type Definition (PNTD), which is a metadata defining legal labels for Petri Net type.

The next chapter concentrates especially on the second part of the standard.

2.5 PNML Concepts

This section will present the main concepts for PNML. This description is based on [9] and [19].
2.5.1 Principles

The ISO/IEC 15909 Part 2 includes specification of transfer format called Petri Net Markup Language (PNML). As described in [9] the main design principles are:

1. Flexibility - meaning that the format should allow any type of Petri net be stored as PNML document. This is achieved by designing Petri net graph as labelled graph. Labels can store additional Petri net specific information and can be attached to various elements of the network (nodes, arcs, pages or the whole network).

2. Unambiguity - is assured by uniquely assigning type to the Petri net. The definition of Petri net type is provided by *Petri Net Type Definition* (PNTD)

3. Compatibility - information exchange should be possible among Petri nets with different types. This is possible because Petri net standard models the graph as labelled directed graphs. *Labels* are special objects which convey specific Petri net information. Labels can be attached to various elements in the network.

The basic model for Petri nets is called *PNML Core*. The Petri net types can extend the *PNML Core* only by providing new labels. The *PNML Core* itself does not specify any concrete labels, therefore it does not represent any kind of Petri net. It acts as a basis for concrete implementation of Petri net types. Figure 2.6 depicts an example UML package structure of Petri nets. The basic structure is defined as PNML Core model package. *P/T net* which extends form the core model is an example of a concrete Petri net type. Apart from P/T net types the standard defines also High-level Petri Net Graphs (HLPNGs) and Symmetric Nets. As depicted in Figure 2.6 every Symmetric Net is also of HLPNG type.

2.5.2 Core Model

The UML class diagram of the Petri net core model is illustrated in the Figure 2.7. Here the most important classes will be described.

A *Petri net document* (PetriNetDoc) represents the document the meets the requirements of PNML Core Model. *Petri net document* consists of unique id and a type. It consists of many PetriNets.
A Petri net (PetriNet) aggregates many pages. Each page consists of a number of objects. The object is basically any element in Petri net graph structure. Each object has unique id and can consists of graphical information which defines dimensions, shapes, size of the object. Places, transitions and arcs are concrete instances of Petri net objects. Places and transitions can be connected through arcs therefore they were generalized to a node.

In order to deal with complex Petri nets: pages, reference nodes and reference transitions were introduced. For large Petri nets is may be useful to split the diagram into several pages. PNML Core model does not allow arcs connecting nodes from different pages (expressed as an OCL expression) due to problem of drawing it graphically in the editor. Instead reference nodes are defines which link to the place or transition element, which is situated on different page.

Labels can convey some extra meaning to some object. PNML Core distinguishes two types of labels: annotations and attributes. Annotations are composed of the data that can be displayed as a text for example name, initial marking of a place, arc annotations, transition conditions etc. Attributes comprise some extra information. Attributes may have effect on visual representation of an object (for example defining shape or colour). The only concrete label defined in the core model is a name.

Graphical information comprises the information about position of a node, positions of arc’s intermediate points. For annotation graphical information define the position relative to object having that annotation. Graphical information define also much more properties like size, colour, and shape of arcs and nodes.

Tool specific information is an optional element which consists of some special information required by a particular application processing PNML document. Tool specific information is defined along with the name and version of the tool.
2.5.3 Petri net type definitions

Next, the definitions of two concrete Petri net types will be described P/T and HLPNG nets. Each Petri Net Type Definition (PNTD) consists of a separate UML package which must directly or indirectly (through intermediate packages) extend the PNML core model. This reference-extension is defined in UML notation as "merge" connector. It means that source elements definitions from the core model are expanded to include the elements in the concrete Petri net model. Apart from required merge reference, a concrete Petri net type may import any elements from external packages. Lastly, Petri net type definitions
may include OCL constraints with extra requirements.

As already mentioned the new Petri net type can extend the core model only by providing new labels. Therefore the PNTD must contain at least one new class which sub-classes Annotation or Attribute class. According to the standard this new label must contain a ”text” reference to value of the label or a ”structure” element reference used for representing the value as a abstract syntax tree.

The Petri net type definitions presented below, aim to give the general structural overview of specialized Petri net types. It is not the goal to present the semantic meaning of these extensions.

2.5.3.1 Place/Transition Model

UML metamodel for Place/Transition nets is illustrated in the Figure 2.8. This Petri net type defines two new labels (annotations): PTMarking and PTAnnotation. PTMarking is attached to place and defines initial marking which is non-negative integer. Each arc can have PTAnnotation label that consist of positive integer and represents the inscription for the arc.

In P/T net type definition it is forbidden to connect two places or transitions together. This is expressed by OCL constraint.

2.5.3.2 High-Level Petri Net Graphs

A HLPNG extends the core PNML model by adding algebraic expressions and complex data types. The HLPNG is a Petri net type which consists of declarations and terms. The declarations are the basis for constructing terms. The declarations define types, functions and variables. The declarations can be attached to Petri net or pages. Places can be annotated with a type and transitions with a condition. The HLPNG metamodel is depicted in Figure 2.9.

The definitions of terms, sorts and declarations are included in the special package called Terms. The metamodel for Terms one may find in [9]. Here we do not presents the details of the HLPNG Petri net type due to high complexity of this network. Furthermore, in the PNML tool, the tool developer, can specify arbitrary metamodel structure which can be attached to labels. Therefore, the implementation and design of the application won’t be bound with any of those concrete Petri net types definition.
2.5.4 The PNML Syntax

The PNML Syntax is defined by mapping between Petri net type metamodel classes and XML elements’ names. The mapping tables for Core, P/T net and HLPNG are included in [9]. PNML schema is based on RELAX-NG. It was chosen because it is more flexible and easy to maintenance then XML schema proposed by W3C.
Figure 2.9: High-Level Petri Net Graphs metamodel [9]
2.6 The PNML Framework

The PNML Framework is currently a prototype of the implementation for ISO/IEC 15909 Part 2. The first release was published in March 2006. The current version of PNML Framework is 1.2.4 (Released 2008-02-06).

2.6.1 Goals of the PNML Framework

The main aim of the PNML Framework is to provide import and export features of PNML models. Figure 2.10 illustrates the most important scenario, in which PNML standard allows for interoperability between different Petri net tools (in the Figure represented as Tool A and Tool B). In another words, PNML format allows reading and writing documents which were created and opened in different tools. PNML framework is built as a standalone library API. It is mainly thought to be used by tool developers who would like adapt PNML syntax to their existing tools but would not like to deal with low level implementation details of PNML.

Figure 2.10, apart from load save features, depicts also create and fetch operations. *Create* means that developer must translate its Petri net entities into instances of PNML Framework metamodel objects. *Fetch* mechanism is used to retrieve loaded PNML elements.

![Figure 2.10: Goals for PNML Framework [15]](image)
2.6 The PNML Framework

2.6.2 Design

The PNML Framework API was constructed using the EMF technology. Each Petri net type is described as a separate Ecore metamodel and appropriate classes are generated through the EMF. In order to include implementation of PNML requirements, templates which are used for generating code which in turn provides support for PNML format, had to be extended. This means that obtained code is adjusted to support PNML standard.

Figure 2.11 illustrates the implementation of core metamodel in Ecore presented in Section 2.5.2. It can observed in the Figure 2.11 some differences between those two diagrams. These differences are present because the diagram presented in the Figure 2.11 is a Platform Specific Model for the EMF technology.

First of all, there are some connections which are not present explicitly in core model defined by standard that is the aggregations to NodeGraphics and AnnotationGraphics. This was introduced because those associations were defined in the ISO standard as OCL constraint. Currently, the PNML Framework does not express OCL constraints declaratively, instead, the conditions are hardcoded in the metamodel. According to [16] at the time when PNML was developed no mature OCL technology existed to implement these constraints in EMF.

Secondly, each class in the core-diagram is defined as abstract (its name is in italic). This design decision prevents from instantiation any elements from the core (as already mentioned in the core model does not have any Petri net type itself). Unfortunately, this design has the big disadvantage that each concrete Petri net type must not only define the concrete Labels but also all concrete classes which can be instantiated according to the PNML standard (that is for example: Page, Place, Transition, Arc etc). This makes all Petri Net types definitions unnecessary big.

Last difference is the lack of text association to XML Schema type element in the Name model element. The text element is instead specified as attribute of the element. This is caused by the fact that it is not possible in Ecore to have a reference to a data-type directly.

The P/T net metamodel is depicted in Figure 2.12. Apart from the new labels (PtMarking and PtArcAnnotation), also other fundamental elements needs to be redefined as concrete classes. This design was chosen to overcome problems with a UML merge concept. The merge concept is used in Petri net standard for extending the core metamodel (please refer to Figure 2.6). The problem is that the EMF does not natively support merge functionality, therefore, the PNML framework uses inheritance to extend elements from the model. This means, for
example, that in order to add new aggregation *initialMarking* to *place* node, it is necessary to override the abstract implementation of Place and then in the new class (PtPlace) define the demanded association (see Figure 2.13). This solution, unfortunately, requires creation of many duplicates elements with very similar names.

### 2.6.3 Supported metamodels and extensibility

At the moment PNML Framework supports only P/T Petri net type definition. The new version is going to add implementation of Symmetric Nets. The next releases will be able to generate specific APIs for local variation of Petri nets (not specified in the standard).
2.6 The PNML Framework

Figure 2.12: P/T Petri net type metamodel designed in the PNML Framework [15]

Figure 2.13: Solution to merge problem in the PNML Framework. Classes Place and Annotation are defined in PNML core package.
2.6.4 The PNML Framework as a universal PNML tool

The PNML Framework does not fulfill the high-level requirements that were set in the motivation section (Chapter 1.1). Firstly, the PNML Framework is not released with any graphical editor. It is an API library, which means the user can either manipulate models programmatically or integrate the PNML framework with some existing Petri net tool. It is, however, possible to generate the graphical editor using presented EMF model. Unfortunately this requires from tool developer strong skills in the EMF and GMF technologies.

Secondly, defining the new Petri Net type requires re-generation and re-compiling the code for the framework. This task may be difficult and can cause many build errors, which tool developers would need to resolve by himself. Also for some, more complex Petri net types, the tool developer may be forced to change the templates, which is a very time consuming task.

What is more, the current version supports currently only one concrete Petri net type: Place/ Transition Nets (P/T nets). Furthermore, the current version of PNML framework has problem with serialization and de-serialization to PNML when the editors is generated for P/T type. Instead, a document is persisted using default in EMF: XML Mapping Interchange (XMI) standard.

Lastly, creating the Petri net editor is not documented in the PNML Framework. Also the source code provided with PNML Framework is not complete and does not include custom templates, which were used to generate the import/export API.
Chapter 3 presents the analysis and design for a universal PNML tool, which later in the text will be called the PNML tool. The analysis section describes the problem statement and requirements (functional and non-functional) for the system. The design part consists of the models used for generating editor and discussions about various design decisions, which were made during the development of the software.

3.1 Analysis

This section describes the analysis of the PNML tool. First the high level goals are identified. After that, the functional and non-functional requirements are described. In order to clarify some the functional features the UML use-case diagram is also included.
3.1.1 Problem Description

The PNML tool is an application, whose aim is to allow end-users to model different types of Petri nets, create and edit new Petri net types. The high level goals of the application were already mentioned in Chapter 1.1. Also the comparison to existing solution (the PNML Framework) was presented in 1.1 and in more details in 2.6.4. Summing it up, the three main high level goals were identified:

1. **Graphical editor supporting various Petri net types and able to accept any new valid Petri net type**
   The graphical editor will be integrated into the Eclipse integrated development environment (IDE). Figure 3.1 depicts the view on sample Petri net model instance. Integrating the editor with Eclipse gives a possibility of re-using many components (views) integrated in the platform. The most important of them is "Palette view", which consists of various elements which can be dragged and dropped onto the editor area. The second view frequently used is the "Property view" (shown at the bottom of Figure 3.1). Property view allows for changing different parameters of a selected element in the editor window. Lastly, the view on the left in Figure 3.1 shows "Package Explorer", which is used to manipulate different files in the Workspace.

   The graphical PNML editor allows manipulating (moving, resizing etc.) places and transitions and connecting them with arcs. What is more, the user is able to attach Petri net type specific labels to pages, places, transitions and arcs. Those Petri net type specific labels, will be called through rest of document the dynamic labels.

   The PNML editor must be aware of PNML document’s Petri net type. This is important because the editor must allow the user to add and manipulate only valid labels specific to given Petri net type. For example, for a document which has a P/T net type (defined in Chapter 2.5.3.1), the editor must constraint the user so he can add ”inscription” label to arcs and ”initialMarking” to places only. Furthermore, as defined in the aggregation’s cardinalities the user cannot add more then one ”inscription” or ”initialMarking” to the same element in the graph.

2. **Possibility to create Petri net types in easy way, which does not require programming**
   In order to create Petri net type extensions the PNML tool will re-use the plug-in architecture of Eclipse (Eclipse plug-in architecture is described in Section 2.2.1). The main requirement is that the user is not required to program while creating the Petri net type definition (PNTD). In order
3.1 Analysis

to define the PNTD the user will need to define the new meta-model (in Ecore format) similar to one presented in Section 2.5.3.1 in Figure 2.8. Optionally, the user can also plug-in graphical figures together with PNTD. Finally, the graphical figures can be dynamic that is change its shape when new labels or elements/connections are added/removed. This, however, will require some programming skills.

3. Persisting the instance models to PNML format

For the purpose of being compatible with other Petri net tools, the models must be persisted in PNML format, which is described in the ISO/IEC 15909-2 standard. The persistence mechanism must be aware of how to serialize the dynamic labels specified in the Petri net types.

![Graphical editor - fully integrated in the Eclipse environment](image)

Figure 3.1: Graphical editor - fully integrated in the Eclipse environment
3.1.2 Actors

Detailed specification of the functional requirements will be easier if the main actors will be identified. From the description of high level goals in the previous Chapter it is possible to distinguish two main users. First of them is a person who is only responsible for modelling Petri nets and use only installed Petri net types. The second (more experienced user) can create and install the new types of Petri nets. Below, more elaborate description of actors is given:

1. Tool developer - is a person, who creates new Petri net types. He knows the concepts of Petri nets, PNML and UML very well. He might not be an expert in Eclipse programming, however, he understands the concepts of plug-in architecture of Eclipse and he has some experience with EMF (especially how to create Ecore diagrams). His main goal is to create new Petri net types easily and fast. He also would not like to do any programming while defining new Petri net type. Tool developer would also like to add new graphical definitions to nodes and arcs in the new Petri net type.

2. Petri net designer - is a person, who wants to use the PNML tool to create instances of Petri net types in graphical editor. He has no knowledge about how to create new Petri net types and about Eclipse Platform. His goal is to create new Petri net diagrams for installed Petri net types. He needs to be able to modify, save and load the Petri net diagrams.

3.1.3 Petri net types

Petri net types are described by UML diagrams which define the concrete labels and their associations to elements from the PNML core model. In order to make it easy for Tool developer to create new Petri net type, it is only required to specify valid UML diagram represented as the EMF model. Apart from the new model, the Petri net definition may optionally consists of graphical definitions for places, transitions and arcs.

UML diagram together with optional graphical extensions form the definition of new Petri net type.
3.1 Analysis

3.1.4 Functional requirements

For the purpose of better understanding the scenarios and their relations to actors, this section will present the UML use-cases and use-case diagram.

3.1.4.1 Creating new Petri net type

Creating new Petri net types is the main goal of the Tool Developer. Extending the PNML tool is done through extension points as described in 2.2.1. Therefore the first step in creating the new Petri net type is creating new Plug in project. After-that, the Tool developer needs to create new EMF Ecore model. This model as described in the standard and as mentioned in 2.5.3 must consists of new labels definitions, which derive from Annotation or Attribute class. Apart from this, the model must bind the labels with the existing elements in the Core model (pages, transitions, arcs or places). This binding is represented as adding containment reference from one element in Core model to concrete label.

The sample Ecore meta-model diagram for P/T Petri net type is depicted in Figure 3.2. At this point, the rules are not given for defining Petri net type meta-models. They will be specified in Section 3.2.2.1 after clarifying some design decision.

When the meta-model describing new type has been created, the tool developer must configure the PNML extension point. This is done through "Extensions" section in the plugin.xml file of the EMF Project. Alternatively the extension can be set-up using Eclipse "Extensions" page as depicted in Figure 3.3.

The use-case presenting given scenario is presented in the Table 3.1.
Use Case: Create new Petri net type

<table>
<thead>
<tr>
<th>Actors</th>
<th>Tool developer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preconditions</td>
<td>–</td>
</tr>
</tbody>
</table>

Main success scenario:

1. Tool developer creates new empty plug-in Project in Eclipse.
2. Tool developer creates new Petri Net type UML model (as Ecore model).
3. Tool developer adds the PNML extension point to the project.
4. Tool developer configures the PNML extension point:
   (a) Tool developer sets the name of the new Petri Net type.
   (b) Tool developer references the Ecore model.

Optional steps:

4a Tool developers adds graphical definitions to new Petri net type (please refer to “UC2 Create custom graphical definitions”).

Table 3.1: UC1: Create new Petri net type
3.1 Analysis

3.1.4.2 Create custom graphical definitions

The second use-case describes the optional scenario when tool developer adds the custom graphics definition to nodes (places or transitions) or arcs. The use-case starts by adding appropriate extension elements: TransitionFigure, PlaceFigure and/or ArcFigure. For each of them tool developer is required to specify the class which implements concrete interface. The design of those interfaces is described in the design section of this chapter (Section 3.2.2.2).

Figure 3.4: Extensions for adding custom graphical definitions

The concise form of this use-case is presented in Table 3.2.

3.1.4.3 Creating new Petri net diagram

The main actor for creating new Petri net diagram is the Petri net designer. The complete use-case is described in Table 3.3. The Petri net designer uses special page of the wizard to choose appropriate Petri net type. This wizard page is illustrated in Figure 3.15.

3.1.4.4 Editing Petri net diagram and adding labels

The following use-cases UC4 in Table 3.4 and UC5 in Table 3.5 show how the diagram can be modified by adding new nodes, arcs and labels. Adding new nodes and arcs can be performed only by dragging and dropping elements from the toolbox. However, the labels can be added in two different way either from
Add custom graphical definitions to Petri net type

Tool developer

Plug-in project has been created, new Petri net type has been configured

1. Tool developer configures the plug-in project and adds new extensions: TransitionFigure, PlaceFigure and/or ArcFigure (Figure 3.15)

2. For each of the extension Tool developer creates new class which implements specific graphical interfaces (description of them is provided in Section 3.2.2.2)

3. Tool developer adds created classes to the graphical extensions

Table 3.2: UC2: Create custom graphical definitions

Figure 3.5: Wizard page for choosing installed Petri net types

the toolbox by dragging the Label item from the toolbox or by using context menu. Figures 3.6 and 3.6 show those scenarios.

According to the ISO 15909-2 standard labels (annotations and attributes) can be added to the Petri net graph, page, arc, place or transition. The standard also proposes that annotations should be displayed as text next to the corresponding object. However, it is not mentioned how the attributes should be displayed. In the PNML tool it is assumed to treat annotations and attributes in the same
3.1 Analysis

<table>
<thead>
<tr>
<th>Use Case</th>
<th>Create new Petri net diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actors</td>
<td>Petri net designer</td>
</tr>
<tr>
<td>Preconditions</td>
<td>–</td>
</tr>
</tbody>
</table>

Main success scenario

1. Petri net designer creates new general project in Eclipse
2. Petri net designer creates new PNML diagram document
3. In the creation wizard, Petri net designer chooses the name and type of the Petri net Type (Figure 3.15)
4. System creates new empty diagram with concrete Petri net type chosen by the user

Table 3.3: UC3: Create new Petri net diagram

![Figure 3.6: Adding labels through context menu](image)

![Figure 3.7: Adding labels through drag and drop and pop-up menu](image)

way. The labels in the PNML tool can be displayed in two different ways:

- As external text to the node. This kind of label can be added to place, transition and arc. Figure 3.8 shows an example how external labels look
like. This kind of labels will be referred in this document as **external labels**.

- As nodes with a text inside. This type of labels can be attached to pages. In the text they will be called **node labels**. An example of node label is shown in Figure 3.8.

This distinction was made because a page is represented as canvas of an editor, therefore it is not possible to display it as text next to it. At the moment in the PNML tool, it is not possible to add labels to the whole Petri net graph.

![Figure 3.8](image)

Figure 3.8: Figure presents two versions of labels: external labels to nodes and arc, and global node labels for pages

<table>
<thead>
<tr>
<th>Use Case</th>
<th>Edit Petri net diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Actors</strong></td>
<td>Petri net designer</td>
</tr>
<tr>
<td><strong>Preconditions</strong></td>
<td>Petri net diagram is created</td>
</tr>
<tr>
<td><strong>Main success scenario</strong></td>
<td>1. Petri net designer chooses an element (node or arc) from the toolbox and drags it to the document window</td>
</tr>
<tr>
<td></td>
<td>2. System creates the element on the editor area.</td>
</tr>
</tbody>
</table>

Table 3.4: UC4: Edit Petri net diagram
### 3.1 Analysis

#### Use Case
Add Petri net type specific labels to created nodes

#### Actors
Petri net designer

#### Preconditions
Petri net diagram is created, a Place node is added in the Petri net diagram

#### Main success scenario

1. Petri net designer selects an element (page, transition, place or arc) and chooses to create Petri net type specific label
2. System list the possible labels (if any) which can be added to selected element
3. Petri net designer chooses one of possible labels
4. System creates Petri net type specific label as a Label node (for page) or as external label (for transition, place or arcs)

<table>
<thead>
<tr>
<th>Use Case</th>
<th>Save Petri net diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actors</td>
<td>Petri net designer</td>
</tr>
<tr>
<td>Preconditions</td>
<td>Petri net diagram is created</td>
</tr>
</tbody>
</table>

1. Petri net designer chooses to save the diagram
2. System persist the model in the PNML format

<table>
<thead>
<tr>
<th>Use Case</th>
<th>Save Petri net diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actors</td>
<td>Petri net designer</td>
</tr>
<tr>
<td>Preconditions</td>
<td>Petri net diagram is created</td>
</tr>
</tbody>
</table>

#### Table 3.5: UC5: Add Petri net type specific labels to created nodes

#### 3.1.4.5 Saving and loading Petri net documents

Use-cases 3.6 and reftbl:UC7 illustrate basic saving and loading scenarios. It could happen that while loading, the Petri net type of the loaded document is not recognized by the system. If this happens system should inform the user about it.

<table>
<thead>
<tr>
<th>Use Case</th>
<th>Save Petri net diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actors</td>
<td>Petri net designer</td>
</tr>
<tr>
<td>Preconditions</td>
<td>Petri net diagram is created</td>
</tr>
</tbody>
</table>

1. Petri net designer chooses to save the diagram
2. System persist the model in the PNML format

<table>
<thead>
<tr>
<th>Use Case</th>
<th>Save Petri net diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actors</td>
<td>Petri net designer</td>
</tr>
<tr>
<td>Preconditions</td>
<td>Petri net diagram is created</td>
</tr>
</tbody>
</table>

#### Table 3.6: UC6: Save Petri net diagram
**Use Case** | Load Petri net diagram  
---|---  
**Actors** | Petri net designer  
**Preconditions** | Petri net diagram is created  
**Main success scenario** | 1. Petri net designer chooses to open existing pnml file  
| 2. System loads the file and open PNML diagram  
**Variations** | 2a The Petri net type of pnml file is not installed. System signals it by showing an error  

Table 3.7: UC7: Load Petri net diagram

### 3.1.4.6 Use-case diagram

Figure 3.9 depicts diagram for identified use-cases and the relation between them. It is presented here to sum-up all use-cases and show an overview of their dependencies.

![Use case diagram](image)
3.1.5 Non-functional requirements

As a non-functional requirements it is especially addressed the simplicity for creating new Petri net types. As already mentioned, the key-point is that Tool developer would not do any programming at all. Furthermore, the Ecore models for Petri nets should easy to build and transform from the UML diagram presented in the standard.

Simplicity applies also to other elements from the Petri net type definition. However, especially for defining dynamic figures, which shapes alters according to changes in the model, programming is unavoidable.

Lastly, the code and all models should be developed with maintainability and extensibility in mind. It is very likely that the software produced in this thesis will be developed further as the Petri net ISO standard is still not 100% complete.
3.2 Design

The design section presents conceptual design details for the PNML tool. This section will cover description of concepts used for realization of the system. Furthermore, more details will be given about extension point design and specifying valid meta-models for new Petri net type definition.

3.2.1 The PNML Core Model

The PNML core model defines the basis for any Petri net type. The platform independent model for the PNML core model is presented in Figure 2.7 Chapter 2.5.2. Here the platform specific model is presented which is based on EMF technology. The PSM model was re-used from the PNML Framework project and only some tweaks were applied to it. The main goal of these changes was to make this model more similar to the PNML Core model defined in the 15909-2 ISO standard. The conceptual parts are shown in Figure 3.10, whereas the graphical representation is depicted in Figure 3.11.

Similar to the PNML framework model (presented in Figure 2.11) the model uses attribute "text" on Name element instead of association to XMLSchema type. This follows from the fact that in EMF it is not possible to create a reference to from class to data-type. In order to solve this problem, it would be necessary to define wrapper classes for every data-type. This would be, however, too time consuming task.

Secondly, the graphical part of the model does not use the OCL constraints, instead the concrete graphical classes are associated to appropriate conceptual classes. This means that a Node has direct reference to NodeGraphics instead of abstract class Graphics. It is important to mention that these differences does not change in any way the semantic meaning of the model and external representation.

The next difference to the PNML Framework core model is that the not every class is marked as abstract. In the PNML Framework’s version of the core model all classes were marked as abstract to avoid creating instances of a PNML core model (which should not be possible because it does not contain Petri net type in itself). This have, however, big disadvantage that new Petri net type models have to sub-class a lot of abstract classes from the core model. The PNML tool uses different approach and it will prevent the users from creating core model instance through the graphical editor user interface. The core model presented in Figure 3.10 defines the same abstract classes as the core model in the ISO
3.2 Design

3.2.2 Design of Petri net type extension point

As described in the analysis part adding new Petri net types will consists of defining new plug-in and configuring the PNML extension point. The design of this extension point is given in this chapter in order to introduce the reader with technical details how the new types and the graphical elements can be
Figure 3.11: The graphical elements of a core package

plugged-in.

Figure 3.12 depicts the structure of the PNML extension point. The extension point itself is defined using extension point schema (which is an XML schema with xsd extension). The top level element is common for all extension points in Eclipse. It contains the required field: name which is the id of a plug-in which defines the extension point. The rest of elements are specific only to the PNML tool.

Each extension element can contain many petriNetType elements. The PetriNetType is the entry for defining new Petri net type. It comprises of a name of Petri net type and a reference to meta-model in Ecore format. The PetriNetType can contain optional implementation of custom graphical elements for places (PlaceFigure), transitions (TransitionFigure) or arcs (ArcFigure). Each of them must specify attribute class, which is the reference to Java class implementing concrete interface: IPnmlFigure or IPnmlConnectionFigure. These interfaces will be described in the Section 3.2.2.2.

The interesting feature of the Eclipse extension point is the possibility to require that the class implements the concrete interface. Therefore, the tool developer will be enforced to provide classes which implement IPnmlFigure or IPnmlConnectionFigure. Unfortunately, it is not possible in the schema to enforce that the meta-model is an Ecore file. Therefore, the run-time checks are required to validate it.
### 3.2 Design

#### 3.2.2.1 Design of the Petri net type models

This section is intended to give the rules how to design new Petri net type Ecore model. As mentioned in the analysis (especially Chapter 2.5.3) the PT type model must consists of two fundamental elements:

1. New labels definitions (at least one) - classes which derive from PNML core *Annotation* or *Attribute* class. They might contain either a "text" attribute representing concrete textual information (for example String or Integer) or "structure" containment, which may refer to abstract syntax tree representation (this is the case in HLPNG, for example definition of expressions, operations etc.)

2. Containment references from the PNML core elements to the new labels

This section will especially focus on the rules for the second point: defining reference classes to PNML core and containment references to labels.

One approach for defining Petri net type meta-models was already discussed while describing the PNML Framework in Chapter 2.6.2 and 2.11. Especially
2.11 showed some possible problems when solving the "merge" package problem for new Petri net type meta-models. The main issue while using EMF is that it does not support merge functionality by default, consequently it is not possible to make a reference to a class from different package. The PNML tool will use the same solution as PNML Framework that is: using inheritance mechanism. The Figure 3.13 shows an example P/T Net Petri net type defined in the PNML tool. The same as in the PNML Framework PTPlace and PTArc must be subclasses of Place and Arc respectively. PTPlace and PTArc will be called the **PNML core reference classes** further in the text. Furthermore, text containment are represented as text attribute. The one subtle difference in the structure is that Place and Arc classes are no longer abstract. However, the way how the PNML tool handles this model is completely different.

![Figure 3.13: Valid P/T net type](image)

The PTPlace and PTArc serve only as a reference to Place and Arc classes respectively. Hence, in a Petri net model with P/T net type, the user will not be allowed to create the instances of PTPlace and PTArc. This would cause duplication of elements (PTPlace-Place and PTArc-Arc) and unnecessary overhead. The way a PNML tool interprets the model is by looking at the references. For example for initialMarking from Figure 3.13 informs the tool that a Place can have one optional label (that is the meaning of cardinality 0..1) called "initialMarking" with integer text attribute. It is important to point out here that all labels are stored through "labels" containment (it is containment defined on PnObject, please refer to Figure 3.10).
Name ambiguities

The ISO 15909-2 standard states that for new labels defined in a Petri net type the references-roles names will be used in PNML syntax. It means that for initial marking, the element name will be called "initialMarking" not "PTMarking". This fact has an influence what names can be used for roles compositions in order to not cause ambiguities. Figure 3.14 shows possible combinations. In the case of variant a) the Petri net type should be considered invalid. This is due to the fact, that it would be impossible to find out which label to create while reading the PNML documents. In another words, Place with Label1 and Place with Label2 attached will have exactly the same PNML document representation. Furthermore, this Petri net type would be invalid with respect to Ecore validation rules. In Ecore model, all the the references within the class (and its super-classes) must have unique name.

The combinations b) and d) are valid Petri net types. Although, in each of them the role name is the same it can be clearly distinguished what kind of label should be used for concrete role and PNML core reference class. The validity of combination c) is from Figure 3.14 is disputable. On the one hand, it does not cause any ambiguities in the PNML syntax but it forms an invalid Ecore model. It was decided that the PNML tool will consider only valid Ecore models. Consequently, a) and c) will be considered invalid. The variation e) is also considered invalid as its re-defines the name containment from the PNML core model.

The ISO 15909-2 does not state which classes may be referenced (in the case of the PNML tool - sub-classed) from the PNML core model and contain aggregation to labels. Of course, those classes must contain the containment to abstract class Label from the core model. However, it is not formulated in the standard, whether it is possible to attach label to abstract classes for example PnObject or Node. The PNML tool allows for defining labels, which are associated with PnObject and its all sub-classes. This decision is consistent for example with Name label defined in the PNML core package, which is associated with PnObject. Please, note that the PNML tool does not allow adding labels to the PetriNet class, at the moment (as mentioned in 3.1.4.4).

Allowing, the abstract classes to be referenced in the Petri net type model, unfortunately, can produce another naming ambiguities. An example is shown in Figure 3.14 as variation f). Although, it is valid Ecore model (the Place class does not inherit from PnObject in this model), the PNML tool would consider Place as having two references with the same name. Therefore variation f) must be considered wrong.
Having mentioned the problems mainly with naming ambiguities, the following rules will be required for the PNML core reference classes and the containments:

1. *The PNML core reference classes* must be sub-classed of exactly one class: PnObject or its derived classes (Ecore supports multi-inheritance).

2. Ecore model must valid with respect to EMF rules, that is two containments with the same name cannot be defined in same class.

3. In order to prevent situation from Figure 3.14 combination f), after merging the information from the PNML core and the new Petri net type, any of the classes can have two containments with the same name.

![Figure 3.14: Valid (b, d) and invalid (a, c, e, f) names for aggregations according to the PNML tool](image-url)
### 3.2.2.2 Design of Graphical interfaces

The interfaces used for implementing graphical definitions are presented in the Figure 3.15. In those classes `getFigure` is called for the first time the node or arc is drawn (it returns the appropriate Draw2d object). The `handleNotificationEvent` needs to be implemented if the developer wishes to alter the figure on some changes in the model (for example new arcs, labels added etc). Those two interfaces differs only with return type for `getFigure` method, which in case of arcs must return special Draw2d type `PolylineConnection` instead of generic `IFigure`.

In both method, also a model object is added as parameter. This important if graphical figures depends on the model object properties.

It should be mentioned also, that this thesis investigated other ways for representing graphics then simple Java interfaces. The experiments were made with SVG format. SVG (Scalable Vector Graphics) is an XML specification for describing two dimensional vector graphics. The big advantage of using SVG images, would be that the tool developer would not have to program a single interface. Unfortunately the SVG is still not mature technology in GMF and GEF, therefore it was not used in the PNML tool.

<table>
<thead>
<tr>
<th>IPnmlFigure</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#getFigure(EObject): IFigure</code></td>
</tr>
<tr>
<td><code>#handleNotificationEvent(Notification): boolean</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IPnmlConnectionFigure</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#getFigure(EOBJECT): PolylineConnection</code></td>
</tr>
<tr>
<td><code>#handleNotificationEvent(Notification): boolean</code></td>
</tr>
</tbody>
</table>

Figure 3.15: Interfaces for specifying custom graphics programmatically

### 3.2.3 Graphical editor

This section will concentrate on design of graphical editor in Eclipse. This design consists mainly of different GMF models (mapping, tooling, graphical and generator), which were discussed in Chapter 2.3.3. This section won’t give
very detailed description of the models. It will rather focus on some parts which were important to enable support for dynamic labels and PNML persistence.

3.2.3.1 Tooling and Graphical model

Tooling model defines the creation tools for four elements: Arc, Transition, Place, and Label. The graphical model consists of three figures which describes arcs as poly-lines, places and transitions as rectangles. For labels there are two graphical representations depicted in Figure 3.16: LabelNode and ExtLabel. The first one is a representation of node labels in PNML that is labels which are associated with pages. ExtLabel, on the other hands, is defined as external label, and it will represent labels associated with nodes and arcs. External labels in GMF are special text fields which float around nodes. Furthermore, external labels are the only valid labels which can be attached to connections.

External labels for nodes, however, have one disadvantage, that is they can be placed only within some exact offset from the node. Consequently, when many nodes need to placed around one node, it could be difficult to read them. The better, but on the other hand, more complicated solution, would be to create external nodes as label nodes with additional arc which connects labels and nodes. This kind of annotations are used for example when creating a Note element (this is standard element which is available in every generated editor, as well as in Ecore diagram editor). However, it would be required probably to hide those connecting "label arcs" in order not confuse them with real arcs.

3.2.3.2 Mapping model

Mapping model is used to combine the graphical and tooling models with domain information. The mapping model for the PNML tool is presented as tree-diagram in Figure 3.17. Top node references reflect the nodes which can be created on the diagram area, whereas links mapping reflect the arcs connecting them. Node mappings are used to define what domain, graphical and tooling element will be used to represent given node. It is important to notice that each node mapping and link mapping is assigned element called Label Mapping. In GMF Label Mappings are used to represent custom labels. Custom labels in GMF are special labels whose parser implementation is supposed to be plugged in at run-time. This is valuable for design of the PNML tool, as dynamic labels cannot be resolved at compile time and it is not possible to determine how the label value should be stored.
3.2 Design

Figure 3.16: GMF graphical model for the PNML tool

Figure 3.17 shows the general structure for the mapping model, but unfortunately lacks the properties (attributes) of elements. Each Label Mapping elements contains two attributes: reference to a graphical label and boolean flag stating whether the label is editable. All labels at Figure 3.17 are editable, Transition, Place node and Link mapping labels correspond to graphical ExtLabel type, whereas Label node mapping label correspond to graphical LabelNode (both described in previous section).

3.2.3.3 Generator model

The GMF Generator model is an implementation specific model which combines all three mentioned model. It is quite complex therefore, in this section only some tweaks are presented which were applied to that model.

Figure 3.18 presents parts of the GMF Generator model, which comprises of EditPart definitions. EditParts are controllers in GMF Model-view-controller pattern, they will be described in more details in Chapter 4.4.1. In Figure 3.18 there are marked EditParts references called ExtLabelEditParts. Normally, GMF uses separate controller for each node’s label, therefore it would create
ExtLabel1EditParts, ExtLabel2EditParts etc. In the PNML tool implementation of label controller must be very generic therefore the generated code can be much simplified by re-using one controller for each external labels. Hence, in the generator model all external labels controllers were unified to one ExtLabelEditPart.

Another modifications were applied to enable persistence to PNML format. Firstly, in the properties of Gen Editor the extension was set to pnml, secondly the "Same File For Diagram And Model" attribute was set to true. The default GMF behaviour is to persist the model using two files: the domain model file and concrete model representation file (graphical information). The PNML model mixes those two data together, therefore, the adequate transformation mechanism will need to be developed. More information about problems and solution for adjusting persistence to PNML syntax are covered in Chapter 4.2.
Figure 3.18: Generator model definition
The previous chapter presented the design of the PNML tool by presenting different EMF and GMF models. When they are defined one can use the powerful EMF and GMF code generators to obtain the implementation. There are three main projects which are the result of the code generation:

- **EMF Model Project** - project containing interfaces for classes and their implementation for the PNML Core model. More information about what functionality EMF is generating is described in Chapter 2.3.1

- **EMF.Edit Project** - project containing classes (*Item Providers*) which serve as controllers in the EMF.Edit framework (for details please refer to Chapter 2.3.1). The PNML tool will use *Item Providers* mostly for obtaining information for property sheets.

- **GMF diagram Project** - this project containing the implementation for the graphical editor

When all the projects are generated one can run the graphical editor. It should be possible to create new PNML files and model simple Petri nets which consist of transitions, places and arcs. Of course, serialization will be done using the default to EMF XMI format, and it will not be possible to add any labels to the diagram.
This chapter will describe the custom implementation which had to be added to support the following features:

1. Serialization to PNML - Section 4.2
2. Loading Petri net types - Section 4.3
3. Adding support for Petri net types in the graphical editor - Section 4.4
4. Adding custom graphical figures - Section 4.5

4.1 Extending generated code

Before the description of the concrete implementation details, it might be worth to list some common practices in extending the generated code. In EMF projects there exist two basic techniques. The first one is using the Eclipse extension points. It is a very elegant solution, which does not require modification of a generated code. Instead, the developer creates a new plug-in project and adds the appropriate extensions. Unfortunately, the extension points are usually limited to what the whole Eclipse UI provides, that is workbench UI, views, commands etc. For example, the developer can add new menu command or a button a toolbar. However, there are no specific extension points to the behaviour of EMF project for example custom serialization or object’s notification support.

The second much more flexible way of customizing generated code is inferring in the code directly. This means that a developer makes direct modification to the source code. However, there is a problem if the code was modified, and after some time the developer wishes to regenerate the code for the modified model, then his custom code will be overwritten. To overcome that issue EMF introduced technology called JMerge. The developer is able to specify the attribute \texttt{@generated Not} on classes/method/xml elements. If JMerge finds this attribute it will skip the generation of code for the whole class, method or xml element.

As mentioned modifying the generated code is very flexible and tempting for programmers, however it can be a source of many problems. It can happen, for example, that when the code needs to be re-generated it conflicts with the changes added previously by developer. If the generation for this method would be skipped the project could not compile or not work as expected. Fortunately, the EMF provides solution for this problem. If JMerge encounters a \texttt{@generated Not} method which would conflict with generated code, it will look for the method with the same name plus the \texttt{Gen} suffix. If this method exists, the
implementation will be generated in this method instead. For example, the listing below show how sample method `getText()` could be customized using `Gen` pattern:

```java
/**
 * @generated Not
 */
public String getText(Object object)
{
    // custom code goes here
    return this.getTextGen(object);
}

/**
 * @generated Not
 */
public String getTextGen(Object object)
{
    // here the code is generated by the EMF
}
```

The other ways of extending generated code refer to GMF projects. The GMF offers so called services, to whom many controllers delegate various tasks. Those services contain a dynamic list of providers which are defined in the GMF extension point. The providers can form a hierarchy and be categorized according to priorities. Therefore, the developer may add its own provider to add a new functionality or define a new provider with higher priority then existing one to override default behaviour. More details, and the concrete example will be shown in Section 4.4.

It the case of the PNML tool, one may think that it would be necessary to generate the code only once, because the PNML core model is well defined and rather unchangeable. Therefore marking the code just with `@ generated Not` should not do any harm. However, it could happen that the GMF models are altered in the future to add new functional features from the standard. Consequently, the code might be re-generated. In general, it use good practice to use the presented techniques in this section to assure high maintainability of the final software.

### 4.2 PNML Serialization

By default, the EMF adds serialization support in the XMI format only (please refer to Chapter 2.3.1 for details). The PNML syntax, however, is based on the Relax NG schema, which is not supported by the EMF itself. It was therefore a challenge to produce a valid PNML serialization mechanism in the limited
amount of time for this project. Unfortunately, the persistence algorithm could not be ported from the PNML Framework. That is due the fact that this algorithm was embedded into the JET templates, which were still not in a mature implementation state. It would be required therefore to extend those templates, which requires deep knowledge of the JET technology. What is more, it would be necessary to generate the code only once for the PNML core package, hence, developing the set of templates seemed to be an overcomplicated task.

The other option for enabling the PNML persistence was to implement a reader/writer from scratch using XML DOM or SAX parsers and then populate the EMF model. Unfortunately, this solution requires a lot of low level programming, first by adding a PNML parser support and secondly to add binding to EMF objects. The implementation would also need to be aware of element’s namespaces, prefixes and references among different elements.

Fortunately there is a way of customizing the built in EMF serialization API. The implementation extension is called the Extended meta-data and will be described in the subsequent chapters. It is beneficial to re-use the EMF persistence API because of the following reasons:

1. The EMF persistence API comes with a solution based on the SAX parser. Consequently it is more efficient then for example DOM parsers especially for large models.

2. The EMF serialization API hides a lot of low level details like coping with name-spaces, prefixes or references.

3. The EMF serialization API contains the implementation for populating and reading EMF objects. It supports also the meaning of all attributes for example transient, volatile, required etc. which can affect the serialization.

4. The EMF adds support for advanced XML binding to Java like mixed contents and Feature Maps. Although, the current implementation does not use them, they might be important in the future work.

As one can see, re-using the EMF persistence API can significantly reduce the implementation time for the PNML serialization. More information about the EMF Persistence API and binding XML to Java, the reader can find in [20] and in [12] especially Chapter 13.2.

\(^1\)It is not required for the reader to understand the meaning of these attributes. They are listed just for information purposes. The complete description of them can be found in [12]
The next section will give a theoretical introduction to Resource and Resource factory concepts. Further, chapters 4.2.2 and 4.2.3 are going to present the implementation details of the PNML persistence mechanism. Lastly, Sections 4.2.4 and 4.2.5 will show how to use graphical information from the PNML format and integrate it into GMF’s graphical model (called Notation model).

4.2.1 Resource and resource factories

Before describing the solution for customizing the EMF persistence API, the introduction to the main persistence concepts will be given: Resources and resource factories.

Resources are in the EMF terminology containers for EObjects. Resources are an abstraction of a physical storage (for example a file), where objects are persisted. Resources are uniquely identified by means of the URI (Universal Resource Identifier). Each resource manages the list of all contained EObject that one can access by calling the `getContents` method.

The Resource interface specifies two of the most important methods `save()` and `load()` which are used for actual persistence. To both methods options can be passed which control their behaviour (these options will be used to attach the Extended meta-data).

Resource factories are used for creating resources. All resource factories are available from the global registry `Resource.Factory.Registry.INSTANCE`. This registry keeps a map of extension types to corresponding resource factories. The EMF persistence mechanism highly relies on resource factories and the global registry. Whenever, a file is going to be opened, the EMF looks for the file extension in that registry. If it exists it gets the factory and creates appropriate Resource object. If, however, the registry does not contain file extension key, the standard XMI resource factory will be used.

4.2.2 PNML Resource implementation

Figure 4.1 on page 63 illustrates the class diagram with all the classes responsible for serialization. In order to trigger the serialization through the custom classes it is necessary to add the following information to `Resource.Factory.Registry`:

```java
Resource.Factory.Registry.INSTANCE.getExtensionToFactoryMap().put("pnml", new PNMLResourceFactoryImpl());
```
Then whenever the EMF will come across a file with "pnml" extension it will use the custom \textit{PNMLResourceFactoryImpl} class. The \textit{PNMLResourceFactoryImpl} object in turns will instantiate an instance of the \textit{PNMLResourceImpl} class.

### 4.2.3 PNML Extended meta-data

The \textit{ExtendedMetaData} is an interface which allows accessing and setting extended meta-data on Ecore model elements. It is mainly used to support structure from the main model, add customization to it or provide additional information. Therefore, the functionality of this interface, supports methods for getting meta-data classes from string element names (and opposite), obtaining kind of serialization element (for example to serialize as element or as attribute), managing prefixes and namespaces and many more.

The PNML tool provides its own implementation of the \textit{ExtendedMetaData} interface called the \textit{PNMLExtendedMetaData}. The \textit{PNMLExtendedMetaData} class is added to the default save and load options in a constructor of the \textit{PNMLResourceImpl}:

```java
public PNMLResourceImpl(URI uri) {
    super(uri);
    setTrackingModification(true);
    this.metaData = new PNMLExtendedMetaData();

    this.getDefaultLoadOptions().put(  
        XMLResource.OPTION_EXTENDED_META_DATA,  
        metaData);

    this.getDefaultSaveOptions().put(  
        XMLResource.OPTION_EXTENDED_META_DATA,  
        metaData);
}
```

The main purpose of the \textit{PNMLExtendedMetaData} is to allow specific customization to the PNML syntax. More specifically:

1. it adds translation of the PNML Core classes to XML elements as defined in the standard [9] in Table 4. The mapping is implemented using the \textit{xmlMapping} field of type the \textit{XMLMap} (please refer to Figure 4.1), which maps element names to the \textit{EClass} objects.

2. it enables using proper EMF features for saving. In XMI the XML elements’ names correspond to names of references. In the PNML syntax XML elements correspond to the meta classes names. Table 4.1 shows an
example of serialization the same model using XMI and the PNML syntax. At each level of the XML document, the PNMLExtendedMetaData translates the reference name to appropriate name of the class, for example the corePnmlpage reference is mapped to the reference type that is a page class.

3. consists of logic which finds appropriate element which needs to be created in the model. For example in Table 4.1, when serialization logic encounters the graphics element it determines that the NodeGraphics object must be created. It is because the containing element (Place) is a subclass of the Node, consequently the only valid class deriving from the Graphics is the NodeGraphics. In another words, based on the name (graphics) and containing type (Place) the PNMLExtendedMetaData must uniquely determine the meta class.

Apart from the PNMLExtendedMetaData Figure 4.1 illustrates also another helper classes: PNMLLoadImpl, PNMLSaveImpl, SAXPNMLHandler, MergePackage and MergeFactory. The first three are used to enable persisting element without special "xsi:type" attribute. This special attribute is used to determine the concrete meta-class when a reference is specified to an abstract class. For example "type" attribute is used on the XMI representation in Table 4.1 for determining that corePnmlpnObjects refers to a Place class.

Lastly, there are some disproportion with respect to name-spaces between PNML and XMI. In EMF each package (Ecore model) is represented by a different name-space. This could be seen in Table 4.1. PTMarking class comes form separate package, therefore it is contained in separate name-space (ptnet prefix). In the PNML syntax, however, the core model types and new loaded labels have exactly the same name-space. In order to solve that issue a virtual package was developed called the MergePackage. It acts as a wrapper for the PNML core package and dynamically loaded package with new labels. The MergePackage has a fixed name-space "http://www.pnml.org/version-2009/grammar/pnml" and delegates method calls to appropriate package (either core model or dynamic package with labels). Together with the MergePackage also a factory called the MergeFactory was defined. It works in the same way as the MergePackage, by delegating calls to either the core or the dynamic package.

### 4.2.4 GMF Notation Model

The PNML core model is a mixed model which contains both domain and concrete (graphical) information. In GMF framework those representations are separated. The GMF uses the special notation meta model which provides
Table 4.1: Comparison of XMI and the PNML syntax. The xml fragment above is in XMI syntax, the one below in PNML syntax

the concrete link between EMF and GEF. The most important feature of the Notation meta model is the independence and isolation from the domain model. This enable the use of any domain model, not only implemented in EMF. The diagram for notation meta-model is presented in the appendix in Figure A.1.

The notation meta-model is the integral part of the generated code for diagram. Therefore it is not possible to exchange it to different model without huge implementation changes. In order to accommodate the graphical part of PNML, the PNML model must be transformed to the notation representation. This process must be performed on every loading or creating a document. The implementation of the PNML tool includes a class `NotationModelHelper` which performs the creation of notation model instances. The creation of model at
4.2 PNML Serialization

Run-time is performed by using the *NotationPackage* and the *NotationFactory* classes. These are generated EMF classes for the Notation Ecore meta-model.

However, the transformation must be also enabled in the other direction, for example when a new place is created its graphical information must be updated according to notation model. Furthermore, those two information must be synchronized and always consistent. Special notification mechanism is established to assure this consistency. It is described in next section.

Figure 4.1: Classes used for PNML serialization
4.2.5 Notification mechanisms

Special class was designed called *GraphicElementModificationListener* which is subscribed to all notification changes of the Notation model. Its main role it to update the graphical elements of the PNML whenever corresponding notation element changes.

In this way, all the information will be always synchronized between the PNML and the GMF model. There is one, more issue however, when the user modifies directly the PNML text document, the diagram should be updated accordingly. This scenario is however supported by GMF, as it tries to load the file again, whenever someone changes it. When the file is loaded *NotationModelHelper* creates the new notation model. Consequently, the graphical information will be updated.

4.3 Loading Petri net types

Loading new Petri net types is the functionality which allows reading and understanding plugged-in Petri net types. As described in the design chapter the Petri net types are defined using the Eclipse extension point. The following section will describe how the PNML tool manages this extension point and how it stores the Petri net types. Furthermore, it will be described what is the logic for understanding the meta-models provided with the Petri net types.

4.3.1 Managing extension point

Figure 4.2 depicts a class diagram of the utility classes used for loading and managing Petri net types. The *ExtensionPointService* class is a helper singleton service used for obtaining information about plugged-in Petri net types. The *ExtensionPointService* stores also the list of all types which were registered. The *ExtensionPointService* is modelled as a singleton and is initialized for the first time when it is called by other classes. The *ExtensionPointService* encapsulates information which is available to all PNML diagrams. The *PetriNetType* is an object representation for the Petri net type. It comprises the same information as in extension point: name, Petri net type model URI and graphical definitions. Graphical definitions will be covered in more details in Chapter 4.5.
4.3 Loading Petri net types

4.3.2 Loading meta-model packages

Loading and analysis the Petri net type model will be performed by using special EMF Dynamic API. Therefore, before the solution for the PNML tool, the EMF Dynamic API will be shortly introduced.
4.3.2.1 EMF Dynamic API

Chapter 2.3.1 gave an introduction to EMF and focused mainly on the EMF code generation features. However, EMF consists also of an API called Dynamic EMF or Reflective EMF. This API allows for exploring, modifying and creating models at run-time, without the need of generating the code.

The Reflective EMF API is especially useful when:

- No type-safe interfaces and classes are needed. For example, the object data is very simple and needs to be shared among different projects. In that case, for every model change, the code will need to be re-generated for every application.

- The model is not known during the development. In that case it is not possible to create static EMF classes. The model must be created and inspected at run-time in this situation.

The dynamic EMF API is mainly implemented through EObject interface. The functionality of EObject includes:

1. Exploring the object. It possible to list all references, attributes, annotations, super-classes etc of the EObject instance. Using the method eClass(), one can retrieve also the meta-data for an object

2. Accessing the data of an object. By using the methods eGet() and eSet() it is possible to retrieve and modify object’s data

3. The EObject implements the Notifier interface which allows for monitoring object data.

The creation of new objects can be performed by using the EcoreFactory class. The EcoreFactory allows for creating all the Ecore types that is: EClasses, EAttributes, EPackages etc. Therefore, it is possible to create models at run-time, without the need for creating Ecore files.

The down-side of dynamic EMF API is that manipulating the model is much slower then in the generated classes. There is, however workaround for this situation. If the efficiency is an important factor, the EMF generator could be invoked at run-time. After that, the application can dynamically load the created classes (similar solution is used for example in the JIT compiler). This solution, unfortunately, requires much more, implementation effort.
4.3 Loading Petri net types

4.3.2.2 Validating the model

The PetriNetTypeManager illustrated on diagram 4.2 is another helper singleton which analyses the Petri net type on lower level of details. The PetriNetTypeManager class obtains all Petri net information from the ExtensionPointService singleton. For each PetriNetType class it loads the Ecore meta-model and initialize the class PetriNetTypeInfo.

The PetriNetTypeInfo in turns analyzes and validates the Petri net type meta-model. Furthermore, the PetriNetTypeInfo establishes the valid dynamic labels for the Petri net type and stores the information for which classes from the PNML core they are valid. The PetriNetTypeInfo checks the following conditions on Petri net type meta-model (as described in design section in Chapter 3.2.2.1):

1. that the model is valid according to basic EMF constraint. In order to check it EMF utility class is used called Diagnostician
2. if at least one new label definition exists in the model.
3. that all labels definitions sub-class from Annotation or Attribute PNML core classes and consist of either "text" attribute or a "structure" containment
4. whether classes which contain containment to label are super-types PnObject or its subclasses
5. verifies that there are no ambiguity situation in naming as described in Chapter 3.2.2.1. That is, that the merge classes will not contain references with the same names.

If PetriNetTypeInfo finds that at least one of these conditions is invalid it will throw the PetriNetModelValidationException and log the error to the plug-in logger.

4.3.2.3 Analysing Petri net type models

The PetriNetManager class in its init() method loads the dynamic model by using the EMF dynamic API. The result of loading the Ecore model is a Resource which contains the EPackage object. By using the EPackage, it is possible to browse all the classes, references etc defined in this package. The
EPackage object is passed to PetriNetTypeInfo, which performs the analysis of the model.

In order to show how PetriNetTypeInfo works an example will be presented. Figure 4.3 illustrates sample Petri net type model, which is a little bit extended version of P/T type model. The goal of the PetriNetTypeInfo is apart from validation, collecting all the defined labels in Petri net type (dynamic labels) and establishing the mapping which label is applicable to which PNML core class. This mapping is defined as pnObjectToLabels hash-table, where the key is a PnObject EClass and the value is the list of dynamic labels.

![Sample Petri net type meta-model](image)

Figure 4.3: Sample Petri net type meta-model

The PetriNetTypeInfo contains also the method canLabelBeAdded which for the given PnObject and label type checks if the label can be attached to given Petri net object instance. The method canLabelBeAdded investigates the labels of the PnObject instance and compare the data to reference cardinality. However, here the main problem occurs that the PetriNetTypeInfo must be able to get an EReference instance for the given label to validate that cardinalities are met. For example, in our modified P/T type example let’s assume that user adds one "initialMarking" and one "finalMarking" to the an instance of Place class for example MyPlace. Because, all labels are stored through the PnObject labels containment (please refer to the PNML core model from Chapter 3.2.1), the MyPlace will be assigned following list of labels [PTMarking, PTMarking]. Now if the user would like to check if he can add another "initialMarking", he can call canLabelBeAdded through PetriNetInfo. However, the PetriNetInfo, which has the knowledge of the list [PTMarking, PTMarking] only, has no longer the data by using which references were those labels added. It is therefore necessary that the MyPlace object allows obtaining additional data about references which were used to add dynamic labels.

There could be many solution to embed reference information in labels. The most simple ones are:
1. Creating separate class in the model with will encapsulate both EReference and the Label objects. This solution, however, requires modification of the PNML core model, which consequently, would brake the consistency between the PNML core model defined in the standard. Introducing new class will also require some tweaks in the serialization mechanism.

2. Extending the PnObject class by defining a hash-table attribute which would map EReference to Label instances. This modification could be done without any changes in the model. Unfortunately, it requires writing custom code for PnObject, and finding the way of passing the EReference to PnObject, whenever a label needs to be created. Instead of simple hash-table PnObject might use also a data-type called FeatureMap. FeatureMap could be used to map EMF features (that is classes, references, attributes etc) into another EMF features. Big advantage of using FeatureMap is that is has a built in support in EMF Persistence API.

3. Transforming the Petri net type model, by creating for each reference new label type at run-time. Each of those new labels will be annotated with a reference type. Using this solution, requires the less coding. There are very little changes required with respect to creating new labels and persisting the model. Furthermore, thanks to the Dynamic EMF API, creating new label types is easy and straightforward.

Here only the simple and straightforward solutions were discussed. Of course one could think about implementing native merge mechanism in EMF, however, it would not be realistic within this project time frame.

The third solution was chosen to solve the problem for label’s references. This is because, it does not require changes in the model (keeps it consistent with the standard) and does not need a lot of programming and modification of persistence mechanism. The choice was also dictated by the limited amount of time. Therefore, the future versions of the PNML tool, may change this implementation detail.

Figure 4.4 shows the transformed Petri Net type model from Figure 4.3. The goal is to first create 1-1 relationship for references to labels. Therefore for each containment new label is created which subclasses from original label. The subclassing gives additional advantage that the derived classes inherit the attributes ("text" attribute if exists) or "structure" reference. The name of new label is a combination of a PNML Reference class name and a containment text.

The second step is annotating new labels with containment information. This is done by attaching EAnnotation class. EAnnotation allows for adding ex-
Implementation

extra information to the model elements. It consists of a source (which is a String value) and a list of references to elements in the model. Annotations may also consist of details entries, however they are not used in the implementation. As depicted in Figure 4.4 each annotation is assigned a source http://www.pnml.org/RoleReference. What is more important, but unfortunately not visible in the figure, the reference of annotation is set to point to appropriate reference.

There could be also other solution for this problem, instead of setting annotation on the class, the special attribute could be added. However this would require more work, because this attribute should be hidden from the user (for example in the property window).

The access to the role from the label is very simple and looks as follows:

```java
EAnnotation annotation = label.getEAnnotation("http://www.pnml.org/RoleReference");
EReference eReference = (ERefERENCE) annotation.getReferences().get(0);
```

Figure 4.4: Transformed Petri net type meta-model

Coming back to the example, which was mentioned in the beginning of this section. If user adds new labels "initialMarking" and "finalMarking" to instance of Place (MyPlace). The MyPlace object, in the proposed solution, will contain the list of labels: [PTPlace_initialMarking, PTPlace_finalMarking]. It will be also very easy to obtain the EReference instance by means of provided annotations.
4.4 Editor support for Petri net types

Having described how Petri net types are loaded, the description will be given how the editor supports the new Petri net types. This section will show the solution to following implementation problems:

- How to register dynamic labels in GMF editor so that it is possible to perform add/edit/delete operations
- How the creation/deletion command are performed, and how to force that only valid labels can be created
- How to add support for dynamic context menu

Before the discussion about the solution for these problems, short introduction to the implementation details of Model View Controller will be presented in GEF and GMF.

4.4.1 Model View Controller Implementation

The MVC architecture of GEF was already presented in chapter 2.3.2. The architecture consists of three main elements: model (the EMF model instance), graphics (visual representation in Drawd2) and controllers. The controller element in GEF/GMF is called EditPart. EditParts are the main elements of the GEF application. Apart from linking the model to the view EditParts creates also its own structure. In most of the cases this structure of EditParts is the same as the model hierarchy. However, in some situations it may be necessary that some models elements are not displayed to the users, or some non-model elements are shown in the view. Then the structure of EditParts will differ. Figure 4.5 presents the simplest case of 1-1 mapping between model and the view.

There are three types of EditParts. First (GraphicalEditParts) provides controller for nodes, second (ConnectionEditPart) represents the connection between two EditParts. The third one (TreeEditPart) is used for providing tree-based representation (for example in the outline view).

The main responsibilities of the EditParts are:

- Creation and maintaining a view
• Creation and maintaining children EditParts
• Creation and maintaining connections EditParts
• Supporting edit operations on the model

![Diagram of EditParts](image)

Figure 4.5: The structure of EditParts [4]

The PNML tool includes the following EditParts implementation:

• **ArcEditPart** - EditPart for arc element
• **ExtLabelEditPart** - EditPart representing external label
• **LabelEditPart** - EditPart representing node label
• **PageEditPart** - EditPart for a page element
• **PlaceEditPart** - EditPart for place
• **TransitionEditPart** - EditPart for transition
• **WrapLabelEditPart** - EditPart of the text which is contained in the node label

The GMF technology introduces the separation of model into two parts: semantic and notation model. Semantic model is a domain model and it is the equivalent model in EMF. Notation model represents the shapes and connections displayed in the editor. The notation model was already introduced in 4.2.4. Because of an existence of the notation model, the MVC architecture had to be modified appropriately. The GMF view on MVC paradigm is depicted in Figure 4.6. Due to the fact, that the PNML tool is developed in GMF, the rest of the text will deal mostly with GMF-specific concepts.
The communication between the model and the view is performed by sending and receiving requests and commands. The EditParts does not manage request and commands by themselves, but they delegate them to specific edit policies. This design has the benefit, that if one needs to adjust the behaviour of EditPart he is only required to install new or modified edit policy. What is more, GMF defines special extension service called EditPolicy Service, which allow installing policies at run-time for any EditPart.

The next sub-section, will give details of edit-policies, requests and commands.

### 4.4.1.1 Edit policies

Edit policies are parts of EditParts which are responsible for editing functionalities. As mentioned above, the main task of EditPolicies is to execute requests. Apart from it, EditPolicies may also forward request to more appropriate EditParts.

EditPolicies are categorized into roles. Each EditPart can have only one EditPolicy per role. The most important roles which will be used in the PNML editor are following:

1. Creational Role - Creation edit policies are responsible for creating model
elements both from notation and semantic model

2. Component Role - Component edit-policy deals with delete requests. This edit policy concerns both the notation and the semantic model.

3. Semantic Role - Semantic edit policies deal with creating and deleting elements from semantic model only

4. Canonical Role - The aim of canonical edit policies is to keep the notation model in sync with semantic information. Canonical edit policies subscribe to relevant semantic model events and send notation create/delete requests to keep the notation model updated.

5. Container Role - Container edit parts consists of logic for supporting EditParts which can be contained in other EditParts.

### 4.4.1.2 Requests and Commands

Requests are object used for communications in GEF. The requests are typically used to demand the creation of new model object and keeping track of objects’ locations or sizes. Requests are forwarded to EditParts but handled by appropriate EditPolicies.

Commands are objects which modify the model. They provide support for undo-redo operations, execution limitations, combining and chaining.

### 4.4.2 The GMF Extensible Type Registry

The EMF Ecore concepts are not sufficient source of information to built a graphical editor application. The GMF introduces so called Extensible Type Registry which is a special application classification system alternative to Ecore. This system augments the domain model information with editor-specific details like:

- Icon and display name
- Defining editing behaviour
- Extending editing behaviour for already defined types

The type, which consists of domain-specific concepts and editor specific detail is called the Element Type. Each element type define how domain class should
be displayed in the editor and how it could be edited (created, deleted, modified etc.). The GMF distinguishes two kinds of element types:

- **metamodel type**
  Metamodel types correspond to single domain EClass element. The editing behaviour is defined by providing special class-factory called edit helper. The purpose of edit helper is to allow constructing editing commands before the default semantic edit-policy. Consequently, the behaviour may be changed without modification of the semantic edit policy. The metamodel type can be also provided with an icon and display name.

- **specialization type**
  Specialization type are types used for extending already defined metamodel types. There could be many specialization types specified for single metamodel type. Each specialization type can define so called edit helper advice class. The aim of helper advices is to decorate the default edit behaviour provided by edit helper.

The GMF maintains the element types in a registry called Element Type Registry. The element types can be defined using extension point or at run-time.

### 4.4.3 The PNML Element types

The PNML tool defines a helper class which manages the element types called the DkdtuimmnmlElementTypes. The generated GMF editor creates a new metamodel type, using GMF extension point, for each top level node defined in the gmfmap model. What is more, each metamodel type is provided with so called semantic hint\(^2\) information which defines the id of an EditPart, which should be used to manage that type. In the case of the PNML tool, by default GMF creates metamodel types for classes: Arc, Place, Label, Transition and Page.

In order to allow editing functionalities for dynamic labels, for each of them new metamodel type needs to be created at run-time. This task is performed by a method registerElementTypes defined in the DkdtuimmnmlElementTypes class. Each of the new metamodel type must also be equipped with the semantic hint, in the same way as types defined in the extension point. In order to allow creating metamodel types at run-time special class was created called the HintedMetamodelType. The HintedMetamodelType extends the GMF standard

\(^2\)“Semantic hint” notion is quite confusing here. It does not have anything to do with semantic data but rather visual id
class MetamodelType and also implement IHintedType interface which supports the semantic hint.

The new label element types can be created with two different semantic hints. The first one is the id of LabelEditPart and the second ExtLabelEditPart. This means that label element type can be either associated with a node label or an external label. The method registerElementTypes made that decision based on the PNML core model class which is associated with the label. For example for Page, the logic of registerElementTypes will create only element type with node label hint. On the other hand, for Place the metamodel type will be configured with external label edit part id. For PnObject both label element types will be created.

The generated metamodel types (that is page, place, label, transition and arc) are assigned a separate edit helper class. It can be used for changing editing functionalities and allow creating dynamic labels. The edit helpers contain the logic to check whether a label can be added to an element. If this is true, an appropriate label command is returned (ExtLabelCreateCommand or LabelCreateCommand - depending on the type of label). This implementation, which uses edit helpers has many advantages. First of all, edit helpers are the single points where it is checked if the label can be added to the node. This condition will be verified on every operation which uses SemanticEditPolicy. It is therefore guaranteed that no matter how the creation for label will be triggered, only valid labels will be added to PNML elements. Secondly, the generated semantic edit policies does not have to be changed and the implementation for labels is isolated from the rest of the code. Figure 4.7 explains the chain of sending the messages, while creating new labels, and shows how the edit helpers are used.

4.4.4 Customizing Edit Parts

Some of the generated EditParts of the PNML tool required some customization in order to support proper labels editing. The customization was performed by adding new edit policies and modifying some of the generated policies.

For Arc, Place and Transition the following edit policies were installed:

1. CreationEditPolicy to allow creating new external labels
2. PnObjectCanonicalEditPolicy. This edit policy is used to keep the external labels view synchronized with their semantic representation
3. ContainerEditPolicy. This edit policy must be installed to enable support
For external labels that is for ExtLabelEditPart following edit policies were added:

1. ComponentEditPolicy. Installing that policy overwrites the standard for labels edit part the VisibilityComponentEditPart. The VisibilityComponentEditPart prevents from deleting element from the model, instead it just hides the label (changes the visibility attribute in notation model). By installing standard policy implementation, the notation together with semantic element can be removed properly.

2. ExtLabelSemanticEditPolicy for a semantic role. This allows creation and deleting labels from the PNML core model

Note, that it was not needed to change any edit policies for PageEditPart and NodeEditPart. That is because label node was defined in gmfmap model as top level node. Therefore all edit-policies for creating/deleting were automatically generated.

The edit policies were installed using the GMF Service mechanism described in Section 4.1. The EditPolicy Service was used for that purpose and two
custom providers were created `PnmlEditPolicyProvider` and `ExtLabelEditPolicyProvider`. The first provider installs policies for arc, place and transition `EditPart`, the second one for external labels. By installing all policies through those providers the generated code was not modified, consequently it led to cleaner design. The providers were specified by using the GMF extension point as shown on Listing 4.1

Listing 4.1: Fragment of plugin.xml which uses GMF extension point to add custom ExtLabelEditPolicyProvider

```xml
<editpolicyProvider
  class="pnmlcore.diagram.providers.ExtLabelEditPolicyProvider">
  <Priority
    name="Lowest">
  </Priority>
  <context/>
  <object
    class="pnmlcore.diagram.edit.parts.ExtLabelEditPart"
    id="ExtLabelEditPart">
  </object>
  <context
    editparts="ExtLabelEditPart">
  </context>
</editpolicyProvider>
```

4.4.5 Palette and context menu support

The editor palette support creating four elements: places, transitions, arcs, and labels. However, by default, the label creation tool is associated with a PNML core Label class. The Label class is abstract, therefore, the label palette tool will not give any effect. In order to plug-in dynamic labels, the method `createLabel4CreationTool` from the `DkdtuimmpnmlPaletteFactory` was modified. The modification includes adding all registered element types for all Petri net types. Although, this solution works correctly, it is not efficient. In the future to the palette only labels valid for current Petri net type should be added.

The dynamic context menu entry ”Add label” was implemented in class `DkdtuimmpnmlDiagramActionBarContributor`, which is an implementation of the `IEditorActionBarContributor` interface. The `IEditorActionBarContributor` allows the developers to contribute menus or commands to toolbars, context menus, status line and so forth. The class `DkdtuimmpnmlDiagramActionBarContributor` overrides the method `menuAboutToShow`, which is executed before every context menu is displayed. The `DkdtuimmpnmlDiagramActionBarContributor`
also subscribes itself to selection events. Consequently, before the menu is to be shown the `DkdtummpnmlDiagramActionBarContributor` knows exactly what element is selected on a diagram. The implementation of the `menuAboutToShow` method obtains the information what labels are available for selected elements. Based on this it populates the context menu.

If the new label needs to be created `CreateLabelAction` is executed. `CreateLabelAction` creates appropriate request and obtain the creation command.

### 4.4.6 Parsers for dynamic labels

Chapter 3.2.3.2 described the mapping model of the PNML tool and mentioned that so-called GMF custom labels were used for text fields in external and node labels. The custom labels in GMF require the developer to attach his own parser implementation. This parser is used to parse the entered text in the label and store it in EMF object.

The GMF has its own extension service called `ParserService` which allows for registering classes which implements `ParserProvider` interface. The PNML tool has its own implementation of `ParserProvider` called the `LabelParserProvider`. The `LabelParserProvider` provides the parser for external and node labels. The following listing presents how the PNML tool registers new `ParserProvider`.

```xml
<extension
  point="org.eclipse.gmf.runtime.common.ui.services.parserProviders">
  <ParserProvider
    class="pnmlcore.diagram.providers.LabelParserProvider">
    <Priority name="Lowest"/>
  </ParserProvider>
</extension>
```

The `ParserProvider` interface defines a method called `getParser` which for the given EMF object returns appropriate parser representation. The PNML tool for labels which have a `text` attribute return the instance of `MessageFormatParser`. The `MessageFormatParser` is a GMF generated parser which supports saving information to EMF attributes specified in the constructor. These attributes could of any EMF data-type kind. For example integer, double, string, but also enumeration. The PNML tool returns the instance of `MessageFormatParser` with `text` attribute if applicable.

For the labels with a `structure` reference, the tool developer must plug-in his own `ParserProvider` and implement special `IParser` interface. Unfortunately at the moment, it is not possible to define parsers definitions in the PNML extension point. This is one extension that will be mentioned in Chapter 5.
4.5 Graphical extensions

Figure 4.8 depicts the classes which includes the implementation for custom graphical extensions. The graphical definitions can be specified to arcs, places and transitions. EditParts are the classes responsible for managing graphical artifacts. EditParts create and encapsulate the graphical figures as Drawd2d objects (Drawd2d was mentioned in Chapter 2.3.2).

Figure 4.8 shows the view on package pnmlcore.diagram.edit.parts and illustrates only methods and attributes necessary for describing the PNML graphical extensions. In order to introduce the new graphical appearance for PNML elements the ArcEditPart, PlaceEditPart and TransitionEditPart had to be adjusted. Each of those EditParts contains a pnmlFigure attribute, which is the realization of either IPnmlConnectionFigure or IPnmlFigure. The pnmlFigure of each EditPart is created in its constructor.

The constructor of each EditPart uses the PetriNetTypeManager to obtain the Petri net type specific instance of the PetriNetTypeGraphicalDef. The PetriNetTypeGraphicalDef is a factory for creating the graphical definition objects from the extension point. Each edit part presented in Figure 4.8 uses appropriate method to create an an PNML figure definition object.

In order to display the custom PNML figures correctly, two methods were changed in the implementation of each EditPart. CreateNodeShape (or CreateConnectionFigure) forward the create request to IPnmlFigure.getFigure(..) method. The second adjusted method is handleNotificationEvent, which notifies also the IPnmlFigure realization by calling the method of the same name.

It is clear to notice that there is 1-1 correspondence between the EditPart instance and the IPnmlFigure realization instance.

Chapter 3.2.2.2 mentioned also an approach where the SVG figures could be plugged-in instead in the code. Currently, there exist an implementation for SVG graphics in GMF. Unfortunately, it is not very stable yet. Also the performance of SVG renderers is quite poor at the moment.

4.6 Conclusions

This chapter described the most important changes that were made to generated editor in order to support pluggable Petri net types. Apart, from the imple-
4.6 Conclusions

It should be stressed that some implementation solutions were a big challenge because not every part of EMF, GEF and especially GMF is well documented. For example, there is only one article describing the EMF Persistence API mentioned in Section 4.2. To make things worst this article mentions the EMF extended meta-data very briefly. Therefore, to fully understand those concepts a lot of debugging effort was put. Another, tricky part was understanding the GMF Extensible Types Registry. Without defining the PNML element types the implementation would be very difficult and complex, because element types are used in request/command operations, and are understood by all edit policies. Finally, the new types of edit policies introduced in the GMF are not well documented. Therefore, many different tests with installing different edit
policies on EditParts were performed to obtain the demanded behaviour.

The implementation presented in this chapter addresses all the functional requirements described in the Chapter 3.1 (Analysis of the PNML tool). However, because of the time constraint not every aspect was implemented efficient enough, or using the best design practices. For example, in Chapter 4.3.2.3 (Analysis of Petri net models) different solutions were presented for solving the problem of encapsulating role information in the the label. For the current solution, the simplest solution was chosen, however, the future versions of the PNML tool might try using more efficient approach. Furthermore, the performance was not considered while registering the PNML dynamic label element types for example to the palette (as described in Chapter 4.4.5). Lastly, some decision could be implemented in more elegant way. For example the modification to Palette, instead of changing the generated code, might use the GMF extension Service called PaletteSevice.

The current solution of the PNML tool lacks also of automated test implementation. As the future releases will add more and more new features, it will be extremely important to test all the changes through unit-tests. Moreover, one could also try to write automated GUI tests, which would be of a great value, for validating the graphical editor behaviour.
The chapter will list and describe the directions in which the PNML tool could be extended. Those extensions will be categorized into two groups. The first one will consider the functional requirements which need to be implemented to support the PNML ISO standard. This group will mention mostly these tasks, in which tools and methods to complete them are already specified and only the implementation effort is needed. The second group is going to sketch and highlight the research areas for making the PNML easy to extend for new Petri net types.

The next versions of the PNML tool should add the following functional features:

1. **Add support to all elements from the PNML core model**
   The graphical editor should support multiple pages, and consequently reference places and transitions. Next, the editor should enable users to add labels to the whole Petri net graph (that is to the `PetriNet` class). Apart from this, the editor ought to add a way for showing and optionally adding/modifying tool specific information (that is providing support for `ToolInfo` class). Finally, the support for all graphical information should be added like: dimensions, fonts, line styles etc. The current version of the PNML tool, only considers the positions of Petri net nodes. The implementation of the whole PNML graphical elements should not cause any conceptual problems. It is mostly programming of transformation to
Notation model, where all of these graphical concepts are present.

2. **Easy plugging-in parsers**
   For the Petri net type which uses the *structure* reference on labels, the PNTD must include the information about how to parse that kind of labels. Although, in the current version of the PNML tool, attaching custom parser is possible, it requires high level of programming knowledge. The future work on the PNML tool requires re-factoring these interfaces and abstracting it so that the tool developer does not have to deal with low level GMF details.

3. **Adding support for SVG graphics**
   The SVG graphics support would be one possible way of leveraging the effort for programming graphical extensions by tool developers. As mentioned in Chapters 3.2.2.2 and 4.5, GMF has a built in support for rendering SVG images, however, the technology is not mature enough at the moment. However, in future releases of the PNML, when SVG implementation will be stable it could be possible to add this support quite easily.

4. **Enabling more features of the editor**
   At the moment, the PNML tool makes use only of some limited functionality offered by the Eclipse IDE. The future versions of the tool might add support for navigator window, assistant provider and many more.

The future work on the universal PNML tool could try investigating the following areas:

1. **Finding graphical representation for Petri net element types**
   One of the goal for the PNML tool, is that the user, who creates new Petri net types does not need to program anything. This means, that also specifying graphical definition for nodes should not be based on programming. Therefore, some easy to define format (probably XML) should be used. This thesis experimented only with SVG graphical representation, but also other solutions should be investigated.

   Usually there is a trade-off between using ready graphical standard formats and creating new ones. On one side it is may be beneficial to use standard and already defined approaches (for example SVG) because there are usually many tools, which provides support for it. On the other hand, the defined standards might be too complex or too inflexible for visualizing Petri net nodes. This thesis unfortunately, because of time constraint, did not investigate this area deep enough.
2. **Defining arbitrary high level structures in PNTD**
   The standard ISO 15909-2 defines the High-Level Core Structure in which the annotation has a reference to the complex data structures. For example, the standard defines the *Terms* package, which class structure can represent conditions, declarations and high level annotations. Furthermore, the ISO 15909-2 defines exactly the PNML syntax for all defined packages in that standard including sample *Terms* package. The Part 3 of the standard will describe how the new Petri net types with arbitrary complex package structures could be plugged-in. This causes a problem how to define the PNML mapping for the new packages. In other words, the plugged-in packages may need to be extended with serialization specific information. The way how to do this, should be the area of investigation. The possible solutions may include annotating the model, or specifying the PNML syntax mapping as a separate document.

3. **Support for Relax NG schema**
   The PNML format is defined using the Relax-NG schema. The ISO 15909-2 standard includes the definitions of all Petri net types by using Relax-NG grammar documents. Therefore, it would be consistent to create the schema documents also for new plugged-in Petri net types. Having generated the schema documents, gives also the advantage that the PNML documents can be validated against it. The behaviour would be also consistent with EMF, where for each Ecore model, the framework can generate XML schema document.

4. **Defining parsers without need of programming**
   As mentioned already, the PNTD will have to include the parsers, for each annotation type, which defines the *structure* containment. The way how it could be done in simple case is by plugging in implementation of some well defined interface. This approach was mentioned in the beginning of this section. Unfortunately, implementing parsers is quite tedious and time consuming task. Therefore, it should be investigated, how this could be optimized, so that the user does not have to perform a lot of programming.
Future work
Related Work

So far, the related work of Petri net tools was limited mostly to the PNML framework. In this chapter, another project is presented called the Petri Net Kernel (PNK). The PNK was developed at Humboldt-Universität within years 1998-2002 that is before the PNML Framework and ISO 15909 was defined.

6.1 Petri Net Kernel

The Petri Net Kernel [21] is an infrastructure for building Petri net tools. The Petri Net Kernel was developed before the definition of the ISO 15909-2 and implementation of the PNML Framework.

The goals of the PNK are very similar to the ones from the PNML tool. The PNK aims to build an infrastructure which is easy to use by people which are not experts in programming. The PNK enables building the tool from the standard or application modules without coding anything. In the other words, the user is able to plug-in different application modules and in the end obtain fully functional Petri net tool. On the other hand, the PNK is flexible enough to be able to define completely new Petri net types. However, in this case programming is required.
The Petri net tool comprises number of application modules, which are controlled by the Petri Net Kernel. Application modules may perform various tasks, for example visualize or serialize the Petri net. The editor is also special type of an application module. The application control is a subsystem of PNK which coordinated all application modules and different nets. Figure 6.1 presents an overview of PNK architecture. Figure 6.1a depicts a Petri net tool which consists of some Petri Net types and number of application modules. Figure 6.1 shows in turn example of Petri net which has one specific Petri net type and application modules running on that type.

The PNK consists of number of interfaces which establish a contract how the application modules should be implemented. For example, there exist a net interface, which can be used to access and modify the Petri net. The editor interface, on the other hand, enables users to implement the new editor module.

The Petri Net Type Definition is defined a a collection of labels. More specifically Petri net type consists of two parts: implementation of classes representing labels and XML document listing all possible labels. The implementation for labels includes mostly the parsing methods which are used for converting string to internal representation. The example of XML document for Petri net type is presented in Figure 6.2. This definition adds new labels for kernel classes Place and Arc. The attributes NaturalNumber and NaturalNumber1\(^1\) refer to implementation classes which are responsible for parsing natural numbers.

\(^1\)The NaturalNumber1 differs from the NaturalNumber class by default value, which instead of 0 is 1.
The Petri Net Kernel does not use model-based software technologies (like Eclipse Modeling Technologies), because they were not mature enough when the PNK was being created. What is more, the graphical editor for PNK was built completely from scratch, which is a big development effort. Also because of the lack of ready plug-in architecture, the Petri Net Kernel had to develop its own extension mechanism. By analysing projects like the PNK one can notice how much time can be saved by using modeling technologies, code generation and integration with existing development environments (for example the Eclipse).
Chapter 7

Conclusions

This thesis described the analysis, design and implementation of the universal PNML tool. The new tool is built using model-based software engineering technologies: EMF, GEF and GMF and integrated in the Eclipse platform.

The new PNML tool comparing to the existing solutions (the PNML Framework) offers much better extensibility. The tool developer is able to create and plug-in his own Petri net type without programming anything and re-compiling existing solution. Furthermore, tool developers can easily define their own graphical representation in the Petri net type definition. However, in the current version of the PNML tool the user must program these graphical extensions.

The PNML tools offers also the graphical editor, which is fully integrated in the Eclipse IDE. By using the editor the Petri net designer can create and modify Petri net documents, which can be persistent to the PNML format. The graphical editor is also aware of the Petri net document type. Consequently, it allows creating, and modifying the elements (labels) valid to specific plugged in Petri net type.

Of course within the limited amount of time, it was not possible to create the full implementation of the ISO 15909-2 standard. The current version of the PNML tool supports plugging in simple kinds of Petri net types, where labels
Conclusions

refers to EMF simple data types (through the text attribute). At the moment the graphical editor does not support creating all the possible elements from the PNML core model (like multiple pages, reference nodes, tool specific information etc). Many of the future improvements and extensions were mentioned in Chapter 5.

It must be, however, highlighted that the current solution creates a solid basis for the future extensions. The PNML tool has been implemented with the extensibility idea in mind, and it tried to keep consistent architecture according to the Eclipse modeling frameworks. The PNML tool found a possible solutions to many problems. The most important of them were: PNML serialization, plugging-in new Petri net types and implementing Petri net type aware graphical editor.

Many different technologies were used to provide the realization of the PNML tool. The most important of them includes the EMF, GEF and GMF. Without them implementing the persistence mechanism and the graphical editor would take much more time then planned for this thesis. Using these frameworks required discovering and understanding many new concepts, in the beginning. However, spending this time was not wasted, as the Eclipse Modeling Technologies generated the most of the PNML tool implementation. This implementation, then was customized using many extensible techniques to add support for dynamic, plugged-in Petri net types.

Loading and analysing Petri net types models was possible through the dynamic EMF technology (Section 4.3.2.1). Dynamic EMF allowed plugging in the whole Ecore models, so that the tool developer does not have to program anything. Basically, he needs only to draw his new Petri net type in the Ecore diagram editor. Unfortunately, there are some problems with this solution as EMF does not support so called merge package concept as used in the standard for defining Petri net type models. Therefore, there some disproportion between those models, and some rules which need to be obeyed (Section 3.2.2.1 describes this in details).

The PNML tools takes also advantage of the plug-in Eclipse architecture. By using it, it is easy to specify the Petri net type definition extension point which comprises: the Petri net type model and optionally the graphical definition. Adding new Petri net type definition relies therefore on creating a new plug-in which extends the PNML tool. This solution has number of advantages, as new Petri net type plug-in can be easily installed, deployed and shared.

This master thesis gave me a lot of insights in the area of model-based software development. This works includes many different frameworks, applications and standards which are currently widely used in various research projects and in
industry. Within the limited time of this project, it was a big challenge to master all the technologies and apply them to the real world problem of building the universal PNML tool.
**Glossary**

**PNML** - Petri Net Markup Language. It is an XML based format, used for describing Petri net models. It’s syntax is defined in ISO 15909-2 standard.

**Petri net type** - represents concrete variant of a Petri net. It is defined as a extension to Core Petri net model in the form of UML diagram

**Petri net type model** - it is a model which can be represented as UML class diagram, which consists of label definitions. Furthermore, the Petri net type model must consists of references, which bind the new labels with the core PNML classes.

**Petri net type definition** - it is a specification of Petri net type. In the PNML tool it consists of Petri net type model and optional graphical definition

**Concrete Petri net type** - Petri net type which extends the core PNML model with at least one concrete label

**Serialization** - saving object to a physical storage for example a file. Deserialization is the opposite process.

**EMF persistence API** - it is a library which support serializations of EMF models

**Tool developer** - it is a user of the PNML tool. His aim is to create and modify the Petri net types.
**Petri net designer** - it is a user of the PNML tool. His goal is to model various Petri net documents based on different Petri net types.

**Dynamic labels** - labels defined in new Petri net type. For example for P/T net type, the dynamic labels are PTMarking and PTAnnotation (please refer to Figure 2.8 on page 22)

**External labels** - labels which are added to arcs, places or nodes. They are displayed as external text to the object

**Global node labels (node labels)** - labels which are added to pages. They are displayed as a node on a page

**PNML core reference classes** - these are the classes defined in the Petri net type model, which reference to some classes from the PNML core model
Appendix A

The GMF Notation meta-model
Figure A.1: The GMF Notation meta-model
B.0.1 Problem description

This chapter will show how the new Petri net type can be developed. As an example a Petri net type based on P/T net will be used. Except of initial marking and inscription labels, the new labels will be defined: arc types and place/transition types. The UML diagram of this new Petri net type is depicted in Figure B.1.

The type attribute is going to control the graphical representations of Petri net objects. The Figure B.2 shows the details of how different elements should look like when appropriate attributes are applied. The inhibitor arcs for example are denoted with extra circle in the end. On the other hand, the queues nodes are drawn with vertical lines in inside.
Figure B.1: The UML diagram for P/T net type with inhibitors and queue attributes

Figure B.2: The different graphical representation according to type attribute
B.0.2 Creating new Petri net type

Here we show the steps required to create a Petri net type.

1. Create new Empty EMF Project

2. Create new Ecore model in the EMF Project
   Set the name of the Ecore model to:
   `dk.dtu.imm.pnml.ptnet.inhibitors.ecore`
   Then add a reference to PNML Core package. Right click on the ecore model and chosen "Load Resource” command. After that: Browse Registered Packages and choose "http://www.pnml.org/version-2009/grammar/pnml” as shown below:

   ![Package Selection Window](image)

3. Define the Petri net type model
   Open the Ecore model and set the following attributes on the package element:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFactory Instance</td>
<td>ptnet</td>
</tr>
<tr>
<td>Name</td>
<td>ptnet</td>
</tr>
<tr>
<td>Ns Prefix</td>
<td>ptnet</td>
</tr>
<tr>
<td>Ns URI</td>
<td><a href="http://www.pnml.org/version-2009/grammar/ptnet/inhibitors">http://www.pnml.org/version-2009/grammar/ptnet/inhibitors</a></td>
</tr>
</tbody>
</table>

   After that, right click on Ecore file and select "Initialize Ecore diagram file”. New empty diagram editor will show up. Create the following Petri net type structure:
When this is done, go back to the `dk.dtu.imm.pnml.ptnet.inhibitors.ecore` and set the rest of the fields according to the following figure:

4. Define the PNML extension point
Double click on the MANIFEST.MF under META-INF and go to dependency tab first. Click add and choose PumlEMF.diagram project as shown below:

Next, switch to Extensions tab of MANIFEST.MF file and add following extension: PumlEMF.diagram.pnml.extension. Add new PetriNetType extension element and specify the name and path to created Ecore file.

After that, as shown in the figure above the graphics extensions should be added: PlaceFigure, TransitionFigure and ArcFigure. For each of that extension a new class should be specified. The implementation of the classes called MyTransitionFigure, MyPLaceFigure and MyArcFigure is given in Section B.0.4. Finally the plugin.xml file should like this:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<?eclipse version="3.2" ?>
```
B.0.3 Testing new Petri net type

When the Petri net is defined, it is possible to test if it works as expected. What you need to do is to run or debug "New Eclipse Application". After that you can create new empty project and a new Pnml diagram file. It can be noticed that a new type was installed from the wizard page. Please make sure that P/T net type with inhibitor is chosen.

When the new diagram is created you can create places, transitions, arcs and attach various labels to them. An example network is presented below. Please also note that graphical representation changes dynamically according to the values of the labels as defined in the problem description.
The new labels can be added by either drag and drop a Label node from the palette or by using the context menu. The screenshots below show examples of these scenarios.
Please also notice, that PNML tool allow you to create only valid labels for specific type. That is for example if a place has a type label, the PNML tool, will not allow creating a new type label.

The diagram is serialized to PNML document. Below the serialized file is presented:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<pnml xmlns="http://www.pnml.org/version-2009/grammar/pnml"
      <net id="default2" type="http://www.pnml.org/version-2009/grammar/ptnet/inhibitors">
    <page id="id21142459">
      <name>
        <text>aaa</text>
      </name>
      <transition id="transition3">
        <graphics>
          <position x="429" y="253"/>
        </graphics>
      </transition>
      <place id="place2">
        <graphics>
          <position x="308" y="165"/>
        </graphics>
      </place>
      <place id="place1">
        <type>queue</type>
        <name>
          <graphics>
            <offset x="-54" y="11"/>
          </graphics>
          <text>Place</text>
        </name>
        <place>
          <arc id="arc2" source="place2" target="transition3">
            <type>
              <text>inhibitor</text>
            </type>
          </arc>
          <transition id="transition1">
            <type>queue</type>
            <name>
              <graphics>
                <offset x="-80" y="11"/>
              </graphics>
              <text>Transition</text>
            </name>
          </transition>
        </place>
    </page>
  </net>
</pnml>
```
<graphics>
  <position x="176" y="330"/>
</graphics>
</transition>
<transition id="transition2">
  <graphics>
    <position x="286" y="319"/>
  </graphics>
</transition>
<place id="place3">
  <graphics>
    <position x="429" y="319"/>
  </graphics>
</place>
<arc id="arc4" source="place3" target="transition2">
  <type>
    <text>normal</text>
  </type>
</arc>
<arc id="arc6" source="place1" target="transition1"/>
<arc id="arc1" source="place1" target="place2"/>
<arc id="arc5" source="transition1" target="transition2"/>
<arc id="arc3" source="transition3" target="place3"/>
</page>
</net>

B.0.4 Implementation classes

package ptNetExtension;

import org.eclipse.draw2d.PolylineDecoration;
import org.eclipse.draw2d.geometry.Dimension;
import org.eclipse.emf.common.notify.Notification;
import org.eclipse.emf.ecore.EEnumLiteral;
import org.eclipse.emf.ecore.EStructuralFeature;
import org.eclipse.emf.ecore.EMetaProperty;
import org.eclipse.emf.ecore.EObject;
import org.eclipse.gmf.runtime.draw2d.ui.figures.PolylineConnectionEx;
import org.eclipse.swt.graphics.Color;
import pnmlcore.Arc;
import pnmlcore.Label;
import pnmlcore.util.graphicaldef.IPnmlConnectionFigure;

public class MyArcFigure implements IPnmlConnectionFigure {

  private PolylineConnectionEx myArcFigure;
  private EllipseDecoration ellipseDecoration;
  private PolylineDecoration polylineDecoration;

  public MyArcFigure() {
    // Constructor implementation
  }
}
ellipseDecoration = new EllipseDecoration();
ellipseDecoration.setForegroundColor(new Color(null, 0, 0, 0));
ellipseDecoration.setSize(new Dimension(10, 10));
polylineDecoration = new PolylineDecoration();
}

@Override
public PolylineConnectionEx getFigure(EObject model) {
    this.myArcFigure = new PolylineConnectionEx();
    this.handleNotificationEvent(null, model);
    return this.myArcFigure;
}

@Override
public boolean handleNotificationEvent(Notification notification, EObject model) {
    if (model instanceof Arc) {
        Arc arc = (Arc) model;
        for (Label label : arc.getLabels()) {
            if (label.getRole().getName().equals("type")) {
                EEnumLiteral text = (EEnumLiteral) label.getText();
                if (text != null && text.getName().equals("inhibitor")) {
                    this.myArcFigure.setTargetDecoration(ellipseDecoration);
                    return true;
                } else if (text != null && text.getName().equals("normal")) {
                    this.myArcFigure.setTargetDecoration(polylineDecoration);
                    return true;
                } else {
                    this.myArcFigure.setTargetDecoration(null);
                    return false;
                }
            }
        }
        return false;
    }
}

class EllipseDecoration extends Ellipse implements RotatableDecoration {
    public void setLocation(Point p) {
        Dimension delta = getPreferredSize().getScaled(0.5).getNegated();
        super.setLocation(p.getTranslated(delta));
    }

    public void setReferencePoint(Point p) {
    }
}

package ptNetExtension;

import org.eclipse.draw2d.Ellipse;
```java
import org.eclipse.draw2d.IFigure;
import org.eclipse.draw2d.RectangleFigure;
import org.eclipse.draw2d.XYLayout;
import org.eclipse.draw2d.geometry.Dimension;
import org.eclipse.draw2d.geometry.Point;
import org.eclipse.emf.common.notify.Notification;
import org.eclipse.emf.ecore.EEnumLiteral;
import org.eclipse.emf.ecore.EObject;
import org.eclipse.swt.graphics.Color;
import pnmlcore.Label;
import pnmlcore.Place;
import pnmlcore.util.graphicaldef.IPnmlFigure;

public class MyPlaceFigure implements IPnmlFigure {

    private Ellipse ellipse;
    private RectangleFigure[] line;

    public MyPlaceFigure() {
        this.ellipse = new Ellipse();
        XYLayout layoutManager = new XYLayout();
        ellipse.setLayoutManager(layoutManager);

        this.line = new RectangleFigure[5];
        for (int i = 0; i < 5; i++) {
            line[i] = new RectangleFigure();
            line[i].setSize(new Dimension(2, 25));
            line[i].setLocation(new Point(8 + i * 5, 8));
            line[i].setVisible(false);
            ellipse.add(line[i]);
        }
    }

    private void changeVisibility(boolean visible) {
        for (int i = 0; i < 5; i++) {
            line[i].setVisible(visible);
        }
    }

    @Override
    public IFigure getFigure(EOBJECT model) {
        this.handleNotificationEvent(null, model);
        return ellipse;
    }

    @Override
    public boolean handleNotificationEvent(Notification notification, EObject model) {

        if (model instanceof Place) {
            Place place = (Place) model;
            if (place.getLabels() != null) {
```
for (Label label : place.getLabels()) {
    if (label.getName().equals("type")) {
        EEnumLiteral text = (EEnumLiteral) label.getText();
        if (text != null && text.getName().equals("queue")) {
            changeVisibility(true);
            return true;
        } else {
            changeVisibility(false);
            return true;
        }
    }
}

return false;

public class MyTransitionFigure implements IPnmlFigure {
    private RectangleFigure ellipse;
    private RectangleFigure[] line;

    public MyTransitionFigure() {
        this.ellipse = new RectangleFigure();
        XYLayout layoutManager = new XYLayout();
        ellipse.setLayoutManager(layoutManager);

        this.line = new RectangleFigure[5];

        for (int i = 0; i < 5; i++) {
            line[i] = new RectangleFigure();
            line[i].setSize(new Dimension(2, 25));
            line[i].setLocation(new Point(8 + i * 5, 8));
            line[i].setVisible(false);
            ellipse.add(line[i]);
        }
    }

    private void changeVisibility(boolean visible) {
        for (int i = 0; i < 5; i++) {
            line[i].setVisible(visible);
        }
    }

    @Override
    public IFigure getFigure(EOBJECT model) {
        this.handleNotificationEvent(null, model);
        return ellipse;
    }

    @Override
    public boolean handleNotificationEvent(Notification notification,
                                              EOBJECT model) {
        return true;
    }
}
if (model instanceof Transition) {
    Transition transition = (Transition) model;
    if (transition.getLabels() != null) {
        for (Label label : transition.getLabels()) {
            if (label.getRole().getName().equals("type")) {
                EEnumLiteral text = (EEnumLiteral) label.getText();
                if (text != null && text.getName().equals("queue")) {
                    changeVisibility(true);
                    return true;
                } else {
                    changeVisibility(false);
                    return true;
                }
            }
        }
    } else {
        return true;
    }
} return false;
Creating Petri net type - Step by step example
Bibliography


