Model engineering on Petri nets for ISO/IEC 15909-2: API Framework for Petri Net types metamodels

Lom Hillah, Fabrice Kordon (LIP6, Univ. P. & M. Curie),
{lom-messan.hillah,fabrice.kordon}@lip6.fr
Laure Petrucci (LIPN, Univ. Paris-XIII),
laure.petrucci@lipn.univ-paris13.fr
Nicolas Trèves (Cedric-CNAM), treves@cnam.fr

Abstract

ISO/IEC 15909-2 International Standard defines a universal XML-based transfer syntax for Petri nets. The main objective of this exchange format, namely Petri Net Markup Language (PNML), is to enable compatibility and interoperability among heterogeneous Petri net tools, while keeping expressiveness and extensibility. Hence, it shall support different types of Petri nets and be extensible to allow for future types. To make it applicable and integrable at low cost to existing tools, a software framework is proposed to the Petri net community. This framework enables fast and cost-effective tools adaptation to PNML use by bringing forth model engineering techniques through model transformation, backed on OMG's Model-Driven Architecture principles. This paper shows that the proposed approach is concerned for compliance with the standard, enhancing compatibility between tools. It is realistic and pragmatic. It copes with extensibility issues by allowing fast re-engineering, thanks to model-driven development approach and code generation.

Keywords: Petri nets, PNML, model-driven development, metamodeling, model transformation, code generation.

1 Introduction

Universal transfer syntax for Petri net tools. The standardisation process of ISO/IEC 15909 part 2 [8], aims at defining a universal transfer syntax for Petri nets. This transfer syntax, namely Petri Net Markup Language (PNML) [19], is intended to become a universal interchange format among Petri net tools. The need for this exchange format has long been expressed within the Petri net community, as discussed in [2]. The proposal made in [2] is now being achieved through the international standardisation process.

The ISO/IEC 15909 part 1 [6] is already an International Standard. It defines Petri nets through a mathematical semantic model, an abstract mathematical syntax for annotations and a graphical notation. Its purpose is to set reference definitions for Petri nets that would be acknowledged unambiguously by systems designers, organisations and Petri net tools vendors, and users working on systems and applications specifications, validation and simulation involving the use of Petri nets.

The ISO/IEC 15909 part 2 is in Working Draft state. One of the main challenges of this future standard is its applicability and tooling with reference implementations that will spread its use.

In parallel to part 2, part 3 of the standard is being started. It focuses on modularity and Petri net extensions (e.g. time).

Part 2 is therefore today the part of the standard where the most active work is expected. PNML relies on:

- the PNML Core Model, a Petri Net metamodel (represented as a UML class diagram). Generally speaking, a metamodel is a business-concern specification of a particular concept at a high level of abstraction, regardless of technological implementation choices. The Core metamodel consists of a graph structure compatible with any kind of Petri net. It sets the basis upon which to build concrete Petri nets types.
- concrete Petri nets types metamodels that are built from the Core metamodel. They extend and enrich the Core metamodel. Three main types have been defined so far by the Standard: *P/T Systems* [6], *Structured Class Nets* [7] and *High Level Nets* [6].
- a Petri Net Type Definition Interface (PNTDI), which is the *RELAX NG* [3] based mechanism (grammar) to define the syntax format for any type of Petri Net.

The PNML syntax has been defined in terms of RELAX NG [3] grammar, which is an XML technology, to enable the translation of a given Petri net into an XML representation. RELAX NG is being standardised as *ISO/IEC 19757-2*. Unlike XML Schema [16], RELAX NG technology is easier to handle and more flexible to use. So far, it is a good candidate to drive PNML syntax rules. However, it does not capture all high level semantic subtilities or constraints that are expressed on Petri nets, such as not connecting two nodes of the same kind in a P/T System (e.g., two places).

To be useful, an implementation of these concepts is required with a particular care w.r.t. three types of versatility induced by:

- the variety of Petri net types that will be considered in part 3 of the standard. This definition process has already started with the three main types defined in parts 1 and 2, as stated above.
- the numerous variants within each Petri net type that may be considered to suit particular modelling needs, such as P/T Systems with capacities, Timed nets with inhibitor arcs, etc.
- tools that are usually based on a Petri net type on which specific information may be plugged to enable the use of tool-specific features (e.g. reference to source code).

Hence, to make this Standard applicable and to cope with compatibility and extensibity requirements that are raised by its specification, this proposal is based on model engineering techniques, based on OMG's Model Driven Architecture (MDA) [11] initiative. It aims at developing a *software framework*, which is a *toolbox*, offering standard APIs to support efficient Petri net tools interoperability by empowering them with automated model transformation techniques.

The incentives for this approach. A *software framework* is proposed to tools developers to support Petri net tools interoperability. It is proposed along with the future ISO/IEC 15909-2 International Standard with the following main purposes:

- **Making ISO/IEC 15909-2 International Standard applicable**. Indeed, PNML transfer format has been designed to enable easy interchange of many variants of Petri nets between Petri net tools. Therefore, interoperability will be made possible by compatibility between the main Petri nets types and by their extensibility to other variants.
- **Easing this International Standard applicability** by providing tools designers with immediate low-cost integration of PNML to their applications and thus making their convergence to this Standard homogeneously through the proposed framework. Such an approach will ensure tools compatibility with this Standard without having to perform the tedious and costly work of dealing with whatever version of PNML. In fact, designing a program to load-/save Petri net models according to the Standard can be an error-prone task and may lead to compliance problems.
- **Providing tools developers with standard APIs** to manipulate Petri net models. These APIs are automatically generated from the corresponding Petri net type metamodel. They also have to be multi-platform.

- **Enabling Petri nets tools designers to add their own extensions** to the main Petri nets types defined by this Standard while keeping some level of conformance. This is achieved by providing tools designers with guidelines to make these extensions. Consequently, the generated APIs will adapt to these extensions.
- Eventually as a consequence of the statements above, **facilitating the Standard use in the academic and industrial worlds** through its integration into existing or newly implemented Petri net tools. Systems designers and Petri net tools users are clearly targeted by this work. This Standard and the accompanying framework set a basis for exchanging Petri nets and can be envisoned to be also integrated into applications that are designed following OMG's MDA principles.

The proposed object-oriented framework has been built by means of model engineering techniques, thanks to OMG's Model Driven Architecture (MDA) guidelines [11] and related technologies such as Meta Object Facility (MOF) [12] and its implementation through Eclipse Modelling Framework's Ecore [5] metamodelling language.

It is a *toolbox* that proposes model transformation chains to tools developers. One transformation chain starts with a conversion from a proprietary representation-based Petri net model into the framework object-oriented one and then finally into PNML. The other starts from a PNML model towards a proprietary representation-based Petri net model.

In the following, after having recalled the principles of model engineering and model transformation techniques which sets our work on PNML w.r.t. MDA initiative in section 2, we describe the methodology to build the standard APIs from each of the main Petri net types metamodels in section 3. Then we step through the code generation process implementing the proposed solution with Java programming language to produce the APIs in section 4. Afterwards, an example of the APIs use in the context of a specific Petri net modelling environment such as CPN-AMI [1] is shown in section 5. This step helps taking a tool developer point of view, regarding the API use, in order to estimate the work required from him/her. Finally, we provide in section 6 the main guidelines to build extensions to the Petri net types metamodels.

2 Context

The Petri Net Markup Language (PNML) is under standardisation process as ISO/IEC 15909 part 2. It aims to be an XML-based standard interchange format for Petri nets in order to provide interoperability between various Petri net tools. Since ISO/IEC 15909 part 3 will introduce more Petri net types, PNML should also allow for extensibility and compatibility. So, PNML should have the following capabilities:

Compatibility This PNML standard shall allow for interoperability, and be compatible with all types of Petri nets, both well-known and future ones that will be defined by the Standard.

Extensibility It shall be versatile and extensible, so as to include other variants of Petri nets developed in the future.

Applicability It must allow engineers to develop their Petri nets tools interfaces using the main high-level programming languages, e.g. Java, C++, Ada.

Compatibility is enforced by using a de facto standard: XML.

To enforce *extensibility*, we propose to systematically use a metamodelling approach starting concretely from the P/T nets metamodel which is based on the Core metamodel. Then, we provide a methodological approach to define extensions of the Core metamodel to represent higher level types of Petri nets (such as P/T Systems with inhibitor arcs, capacity places, low-level priorities, etc.). Such an approach takes place within the context of MDA [11].

To make the standard *applicable*, we propose a software framework, designed using MDA's model-driven development approach, putting into work model transformation techniques. It provides tools designers with standard APIs to manipulate Petri net models. These APIs are automatically generated in Java from the corresponding metamodel.

We now introduce model-driven development and model transformation principles.

2.1 Model-driven development and model transformation

In 2001, the OMG introduced MDA [11] that promotes system development with models as first-class artifacts. MDA introduces several successive models:

PIM (Platform Independent Model) Models specification at this level is regardless of any particular platform technological requirements. E.g., when we specify what a Petri net is, conceptually speaking, with nodes and connectors and labels on these objects, we do not care about the programming language that is going to be used to implement these concepts.

PSM (Platform Specific Model) Models specification takes into account a given platform specificities, allowing for parameterisation in order to facilitate models transformation into generated code. For instance, a banking application PIM could be specified with UML, then transformed into a PSM designed with UML profiles for EJB, thus targeting EJB technology.

To cope with transformation from one model to another, model engineering techniques propose a set of solutions that are being studied and normalised by the OMG.

Model engineering techniques are concerned with models transformations. These techniques require the definition of metamodels and sets of rules to perform a transformation from a source metamodel to a target metamodel. Such rules define the *metamodel mappings*. Then, any model compliant with the source metamodel (i.e. formalism) can be processed by a given set of rules, thus rigorously defining a transformation towards the target model. Consequently,

- PIMs can be transformed into other PIMs taking into account specific business concerns for a particular domain. E.g., how could we transform a Structured Class Net PIM into a P/T System PIM?
- PIMs can be transformed into PSMs, taking into account platform specific concerns. E.g., in our approach, we use Eclipse Modelling Framework [5] (EMF) to transform the Petri net types PIMs into PSMs targeting EMF Java-based platform.
- PSMs can be transformed into PSMs, taking into account other platforms specificities. E.g., how could we transform the EMF-based Petri net types PSMs into EJB-based PSMs?
- PSMs can be transformed into PIMs. E.g, abstracting business concerns from an application.
- PSMs can be transformed into source code. E.g., generating Java code from a Petri net type PSM using EMF.

In practice, models transformations are not easy to perform. The main problems deal with semantics. Some kinds of transformations are hardly mature (e.g., PIM to PSM, PSM to PSM) while others are at the prospective level (e.g., PSM to PIM). MDA philosophy aims at portability, interoperability and reusability through architectural separation of concerns.

Figure 1 gives an overview of model transformation principles. Generally speaking, given a source metamodel and a target metamodel, transformation rules could be input to an engine, so as to perform transformation on a given source model in order to obtain the target model. The transformation rules can be directly expressed as programs or be driven by metamodels.

Industrial practitioners acknowledge the advantages of MDA-based development techniques for designing complex systems. Separation between domain and platform specific issues, namely PIM and PSM, allow a more precise approach to the viewpoints of the system under design. The framework provided by the OMG, MOF [12], includes two viewpoints :

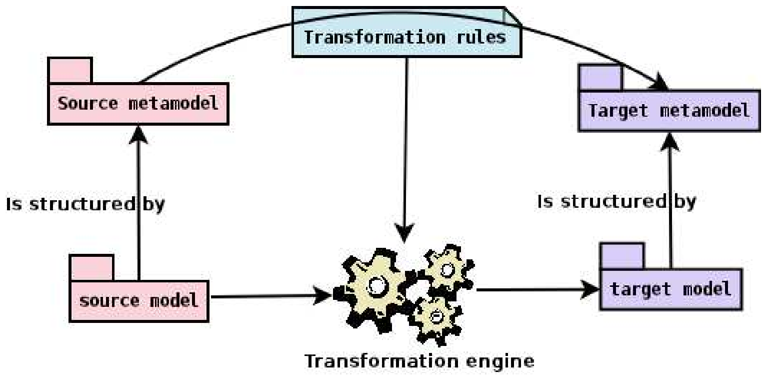

Figure 1: Simple overview of model transformation

1. *the modelling viewpoint* : it deals with the meta levels and hence provides model information for a particular field. For example, in this paper, a basic core metamodel for Petri Nets is specified, according to the ISO/IEC 15909-2 Standard. Then we provide specific extensions to this metamodel to allow for its versatility, so as to define particular Petri Net types, such as P/T Nets or Structured Class Nets.

2. *the data viewpoint* : the implementation level, from which APIs for different programming languages can be developed to meet specific requirements for the engineers' tools.

Hence, MOF aims at extensibility by providing a framework for representing any kind of metadata. The classical four-layer architecture describing the MOF concept is of interest. Here, we consider the two inner layers, metamodel layer (M2) and model layer (M1).

2.2 Metamodel specification viewpoints

Experiences such as the metamodelling of UML revealed that there are three main views, each corresponding to orthogonal goals (figure 2):

1. *abstraction*: it emphasises reusability and any type of extensibility for the specified metamodel. However, due to a high level of abstraction, it is difficult to express precise semantics in the metamodel. So, implementation is heavy and strongly reduces applicability.

   This view has been experimented with UML and leads to dramatic complexity when enabling the respect of constraints by means of OCL [14] assertions. This approach is difficult to maintain.

2. *exchange*: its purpose is to allow metamodel exchange for technical discussions between system designers.

3. *productivity*: we emphasise the structuring of the metamodel to set constraints without additional OCL assertions Then, it is possible to constrain extensibility to appropriate directions (such as avoiding to connect two places together). This view guarantees a certain level of compliance with the core metamodel. A customisation process is easier to define.

   APIs generated with such an approach are generally more accurate in terms of usability by programmers.

As shown on figure 2, we mainly target the productivity viewpoint, while remaining open to reusability and extensibility for the Petri nets types.

The metamodels sketched in the current draft version of ISO/IEC-15909 part 2 is specified at a very high level of abstraction. Since: 1) extensibility should be constrained in order to preserve the Petri net semantics, 2) APIs should be usable, we propose in the metamodels design to put emphasis on the third axis, while remaining compliant with the Standard ones without any loss of relevant information. This approach is described in section 3.

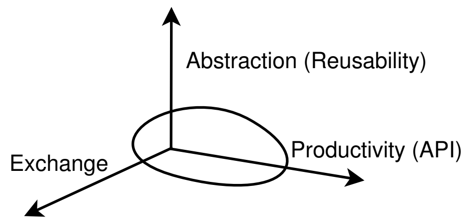

Figure 2: Metamodelling viewpoints

3 Methodology

In the following sections, the production chain for the API framework is detailed. Then main Petri net types metamodels are introduced and their designing process explained. In the following the ISO/IEC 15909 part 2 Standard will be refered to as *the Standard*.

3.1 Overview

The API framework has been designed following a three-level structuring approach:

1. First of all, a *Petri Net Core Metamodel* (PNCM) has been created. It is based on *PNML Core Model* defined by this Standard. This metamodel represents the modelling process starting point. It means that from the PNCM, a hierarchical classification allows for structuring the main Petri Net types models defined by the Standard and through its future versions. Each main Petri Net type model represents a significant Petri net family (e.g., P/T Systems, High-Level Petri Nets, etc.) which could contain many variants (e.g., P/T Systems with capacities, Timed Petri nets with priorities, etc.), thus meeting the extensibility goal. From the Standard point of view, the classification approach roughly corresponds to the relationship between the Petri Net type (meta)models packages, symbolically represented by the UML *merge* concept.

2. From a given Petri Net type metamodel (see Figure 3), *a tailored API is automatically generated* to manipulate models structured by the given metamodel specification. Another programming interface is generated to import/export the models from/into PNML documents, according to the specified Petri Net type metamodel and its variants.

3. To start model manipulation (i.e., creation, modification, navigation to fetch information) and having them eventually transformed into PNML or into a tool internal Petri net representation, a client program is needed. Tools developers write this client to perform the mapping between their own tool internal representation and the information retrieved or injected into the models. At this stage, the only requirement from tools developers is to know the specified Petri Net type metamodel elements and their structuring relationships, in order to use the corresponding API to create models and fetch information.

Figure 3 is a diagram illustrating the three-level approach described above.

In the following, the main Petri net types metamodels defined in this Standard will be introduced, along with their designing process.

3.2 The Core metamodel

The Standard Core Model [8] specifies the general concepts composing a Petri net and their relationships. This point of view is of course very conceptual and at a high level of abstraction. When implementing this model by means of Eclipse Modeling Framework's Ecore language, we aim at a more specific productivity-oriented use. Therefore the Standard Core Model is refined.

Consequently, some general concepts are directly implemented and some are not. Yet, the hierarchical approach is preserved for all main Petri net types.

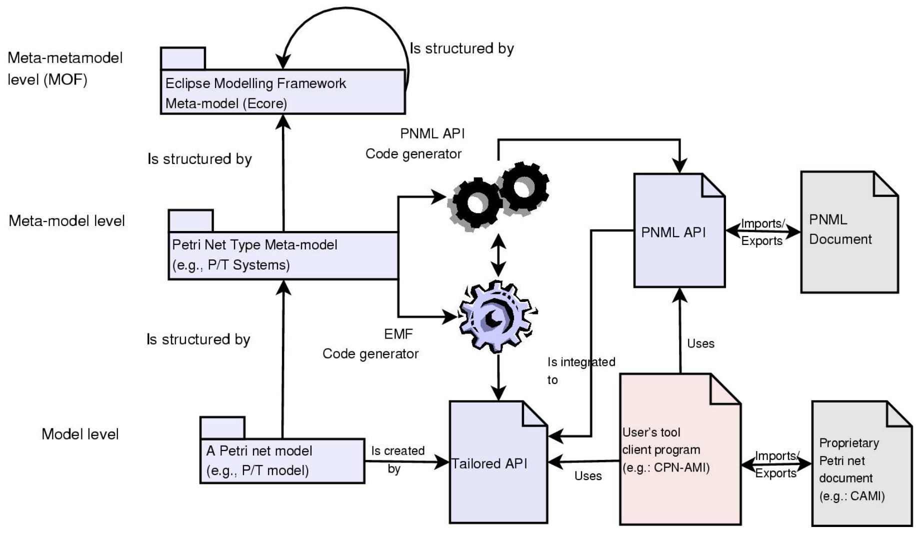
Meta-metamodel level (MOF)
Eclipse Modelling Framework Meta-model (Ecore)
Is structured by
PNML API Code generator
Is structured by
Meta-model level
Petri Net Type Meta-model (e.g., P/T Systems)
PNML API
Imports/ Exports
PNML Document
EMF Code generator
Is structured by
Is integrated to
Uses
Model level
A Petri net model (e.g., P/T model)
Is created by
Tailored API
Uses
User's tool client program (e.g.: CPN-AMI)
Imports/ Exports
Proprietary Petri net document (e.g.: CAMI)

Figure 3: Overview of the production chain

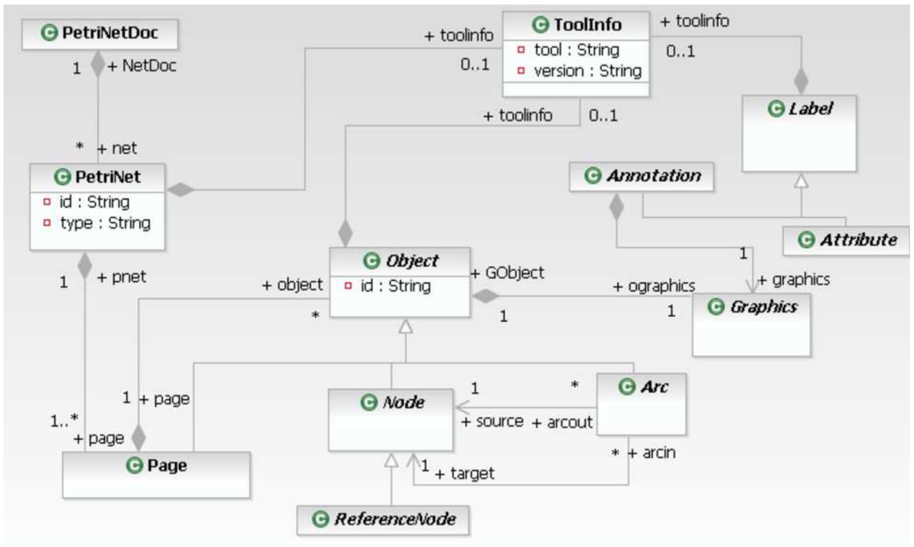

Figure 4: Overview of the implemented Core metamodel.

3.2.1 Providing a starting package with abstraction viewpoint

The basic Core Model package is designed with abstract classes for most parts, so as to provide a strong basis for future extensions. As a matter of fact, a Petri net model cannot be of any particular Petri net type (e.g., P/T Systems) by only using this Core Model. Instead, it would be instantiated from well identified Petri net types packages, such as P/T Systems or Structured Class Nets, all of which implement the UML *merge* concept [13] (used in the Standard) by means of the EMF's *import* capability associated with inheritance mechanism, since EMF does not natively provide *merge*. Therefore, the inheritance relationship between these concrete packages and the Core Model is possible. In this case, *import* and *inheritance* replace *merge*, which is used in the Standard.

In Figure 4, which is a UML diagram, abstract classes are in *italic* letters. This convention will be kept throughout this document for all other figures.

3.2.2 Keeping minimum transversal information

In the implemented Core Model, only the minimum subset of transversal information in all Petri net types is kept. Consequently, there is no Place or Transition concept anymore. The Node concept suffices. If Place or Transition was kept in the PNCM, the inheritance relationship from it to a further Petri net type package would be confusing, regarding the Node concept. For instance, consider that in P/T net type package there are *PTPlace*, *PTTransition* and *PTNode* specializing Place, Transition and Node, respectively. Then it is natural to say that *PTPlace* and *PTTransition* are *PTNode*, and *PTNode* is a Node. But what about saying that PTPlace is also a Place and PTTransition is a Transition ? Conceptually speaking, multiple inheritance is allowed, but in this case, it is confusing for refinement and implementation purposes. Since Place and Transition in the Core Model are Nodes and do not contain any relevant information at that stage, they are simply removed. But the Arc concept is kept in the Core Model since *source* and *target* associations are relevant for the Node concept and would not be enriched by further Petri net types, whatever the Petri Net type.

For lack of space in Figure 4, the entire Graphics metamodel is not shown. The *Graphics* metaclass is a representative of the Graphics metamodel. From the Standard Core Model to this refinement, no important requirement was lost.

3.2.3 Avoiding basic inconsistencies

As an immediate consequence, ReferencePlace and ReferenceTransition concepts are also removed from the Core Model. They are replaced by the ReferenceNode concept. Since the UML ”Redefines” concept is not implemented by EMF, it is redundant keeping *Label* information in PetriNet and Object metaclasses in the Core Model whereas it is purposely redefined in the concrete packages. From this point of view, it will be then impossible for a Transition to have an initial marking, since this label is only attached to the Place metaclass in a concrete package (e.g., P/T net type). The same consideration is applicable for a Place. Basic inconsistencies are therefore avoided.

3.2.4 How to enforce the constraints ?

Constraints are expressed in the Standard by means of the OCL language [14], but in the implementation they are enforced by programming and set into a well identified package, easy to maintain and to extend. This is because there is no mature OCL implementation technology integrated to metamodels verification process. Some interesting work by Kent University [18] was considered but it is not yet mature and not fully documented. Another work has been introduced at OCM'2005 by Université de Lille but the first release is not available yet. We have also stressed in 2.2 that we follow a productivity viewpoint in developing this framework. Consequently, no OCL assertions and checking are embbeded in its development at the current stage. A future work will be to seek for mature constraints checking technologies at the metamodel level, either OCL or derived work from OMG.

Constraints on datatypes particular ranges of values, such as Positive integer or Natural, are also enforced by programming. However, constraints checking is not mandatory. Users can simply ignore it when using the framework to save or load models.

3.3 P/T System type metamodel

In this section, the P/T Systems type metamodel is specified. In the following, P/T Systems will be refered to as **PT Net type** or simply **PT Net**. Since it is a refinement and an implementation of the corresponding formal type metamodel introduced in this Standard in clause 5.3, its designing process is explained. Especially, when speaking of involved technologies such as Eclipse Modeling Framework, model productivity is considered and consequently restrictions are set on the formal type metamodel, without any lost of requirements.

3.3.1 PT Net metaclasses

The PT Net package is the first concrete package from which a Petri net could be instantiated. It imports the Core Model and hence could inherit from its concepts. EMF does not provide any explicit implementation of the UML *merge* concept [13], as specified in this Standard. Yet, the *import* one associated with the inheritance mechanism is powerful enough to help managing the semantics of this framework main objectives, as stated in the introductory section. Because of the PT Net package concreteness, abstract metaclasses are seldom, as it can be seen on Figure 5. Putting aside the Graphics package and not considering *Object* and *Annotation* metaclasses that are kept here only for structuring purposes, the **PTNode** metaclass is the very new abstract metaclass that holds the concrete concept of **Name** which *redefines* Label. The UML *redefines* concept is also used in the Standard.

In this metamodel, the following concrete metaclasses are defined (See the UML diagram in Figure 5):

- **PetriNetDoc** is the entry point. It represents the PNML document and may contain many P/T net type models.
- **PTNet** is a P/T net type model and it has a **Name** which is an Annotation that *redefines* Label (See Core Model). It also contains at least one Page.
- **Page** is an *Object* and contains Pages, PTArcs, PTPlaces, PTTransitions, PTRefPlaces and PTRefTransitions. *Page* is a concrete concept imported ”as is” from the Core Model and consequently does not need to be redefined, unless it brings new specific features to this Petri net type, which is not the case up to now. A Page does not have a *Name*.
- **PTPlace** is a *PTNode* and thus has a *Name*. More specifically, it may have an initial marking, represented by *PTMarking*, which is an Annotation, redefining Label.
- A **PTTransition** is a *PTNode* and thus has a *Name*. It has no additional feature.
- **PTArc** is an Arc and still connects two Nodes, as stated in the Core Model. It may have an inscription[1] which is an Annotation, redefining Label.
- Since **PTNode** is a Node, a PTArc can connect two PTNodes.
- **PTRefPlace** and **PTRefTransition** are *ReferenceNodes* and reference a PTPlace and a PTTransition respectively.
- **PTArcAnnotation** is a PTArc inscription. It is an Annotation, therefore it is also a Label.

3.4 Structured Class Net type metamodel

In this section, the Structured Class Net [7] (SC Net) type metamodel is specified. The designing approach is the same as for P/T Systems. Relationships between these two metamodels are implemented via the import and inheritance mechanisms. Some consideration about MathML [17] will be introduced.

Figure 6 is a UML diagram which shows the metamodel specification for SC Nets. This metamodel defines main metaclasses for a SC Net. These metaclasses mostly inherit from PT Net type metamodel metaclasses. It also introduces some new metaclasses imported from the math specification metamodel (Cf. 3.5), necessary for its completeness.

3.4.1 The SC Net metaclasses

Though no more complex than PT Net type metamodel, the SC Net type is however more specialised and imports the PT Net package. It means that a tool that can read or write SC Net models is also able to read and write PT Net models. The SC Net main metaclasses and their relationships are structured as follow:

[1]Actually, an inscription of value 1 is implicitly assumed if not specified.

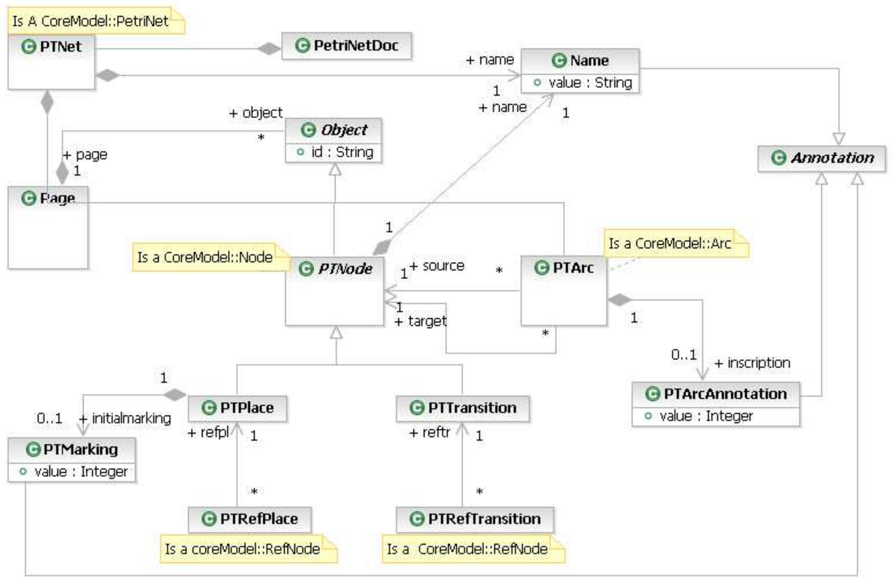

Figure 5: Overview of the implemented PT Net type metamodel

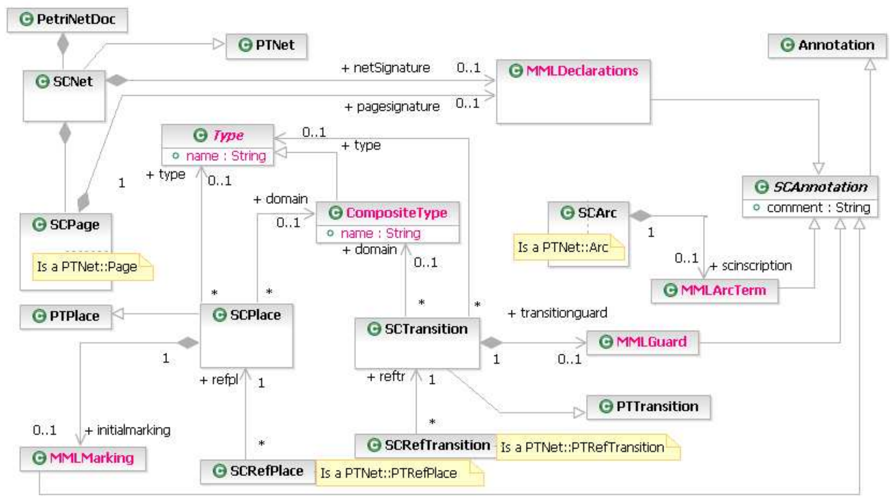

Figure 6: Overview of the implemented SC Net metamodel

- The **SCNet** metaclass specialises PT Net's *PTNet* class. Therefore it inherits all the features from the PTNet metaclass. It adds its own feature that is essentially its *signature*. This signature is a **SCAnnotation** which is an *Annotation*. This feature is a *MMLDeclarations* metaclass which consists of a set of elements from the math specification package (See 3.5), which shall be serialised in MathML.

- **SCPage** inherits all the features from *PTPage*. we could also directly specialise the Core Model Page class, but it is better for compatibility and extension issues to preserve this bottom-top inheritance mechanism. A *SCPage* has a signature.

- **SCPlace** obviously inherits from *PTPlace* so that it can have a name. It adds its own features that are an *initial marking* (**MMLMarking**), a *type* (**Type**) and a *domain* (**CompositeType**), the specifications of which come from the math package.

- **SCTransition** is a *PTTransition* and has its own features that are a *guard* (**MMLGuard**), specified in the math package, and a *type or domain* specified in the same package.
- **SCArc** is a *PTArc* with its own specific annotation which is defined as a **MMLArcTerm** in the math package.
- **SCRefPlace** and **SCRefTransition** specialise respectively *PTRefPlace* and *PTRefTransition* from PT Net Package. It is important to notice then that a SCRefPlace could reference a PTPlace and so can a SCRefTransition.

From this specification, it could be deduced that the specificity of SC Net type comes from the math specification package, which defines the metamodel of the annotations required for SC Net type specification completeness. This metamodel, unlike the fundamental ones (i.e., Core Model, PT Net, SC Net) will be serialised in MathML [17], instead of PNML. Thus, its structure must reflect MathML way of specifying the contents of math expressions as Abstract Syntax Trees.

3.5 MathML annotations for SC Net type

The algebra for SC Net will be implemented in MathML, which is a W3C standard [17]. As stated by W3C,

> ≪MathML is a low-level specification for describing mathematics as a basis for machine to machine communication.≫

MathML is composed of presentation and content markups. The presentation markup is for rendering purposes while the content one captures mathematical semantics. They can both be combined in the same document for the same expression. The content markup is mainly considered here. As could be expected, contents for MathML use Abstract Syntax Trees, Prefix Notation, for structuring math expressions.

3.5.1 Why MathML ?

MathML is used to specify the annotations for higher level Petri nets types other than PT Systems, since it is well suited to this purpose. Those higher level Petri nets types have annotations more complex than PT Systems'. These annotations cannot be simply expressed as for PT Systems. They thus require some sophisticated means to specify their algebra and MathML provides such annotations definition mechanisms.

MathML has all the features required to annotate a SC Net and provides a powerful way to define non-MathML standard functions (e.g., the broadcast function, cf. SCNet Standard specification, [7]). Moreover it is a XML technology and a standard, and could thus be easily integrated into PNML. PNML does not provide any implementation mechanisms for higer level Petri nets types annotations yet. However, exchanging Petri nets remains possible. For instance, Since the SC Net package possesses all the features defined in the PT Net package, a SCNet could be serialised as a PTNet. The MathML annotation information would be lost in such a case. Also, if a Petri net tool does not take into account MathML annotations for a SC Net, it can simply ignore them and fetch the underlying PTNet instead. Using MathML is then both an interesting approach and an open issue that the Petri net community wishes to investigate.

3.5.2 Main specification.

The main classes specified in this package and their relationships are defined as follow (See Figure 7):

- Every SC Net Signature is of **MMLDeclarations** type. A *MMLDeclaration* consists of one or more **Types** (e.g., *Class C* in AMI-NET syntax [1]), **Variables** (e.g., *Var y* in AMI-NET syntax), and **Composite Types** (e.g., *Domain D* in AMI-NET syntax).
- A **Type** is either an **Interval** (e.g., *Class C is 1..4* in AMI-NET syntax), or an **Enumeration** (e.g., *Class C is [a,c,e,t]* in AMI-NET syntax). Thus, *Type* is an abstract class.

- A **MMLMarking** and a **MMLArcTerm** are of **Expression** type, which is composed of a **Function** and **Parameters**. A *Parameter* is itself an *Expression*, thus allowing a recursive definition. Such a definition aims at modelling Abstract Syntax Trees for *Expressions*.
- A **MMLGuard** of a SCTransition is of **Boolean Expression** type, which is composed of a **Boolean Function** and **Boolean Parameters**. A *Boolean Parameter* is both a *Boolean Expression* and an *Expression*, since both boolean and non-boolean functions can be used in a Boolean Parameter.

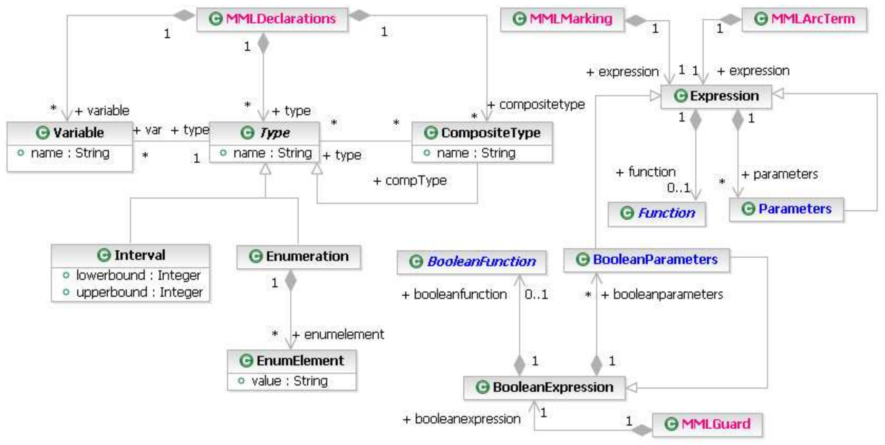

Figure 7: MathML annotations metamodel for SC Net

Figure 7 is an UML diagram illustrating the math annotations metamodel for SC Net.

For lack of space in this paper, detailed descriptions for *Expression, Boolean Expressions, Parameters* and *Boolean Parameters* are not included. Mappings from SC Net built-in functions to MathML have also been defined but not included in this paper.

Synthesis

An effort was put on keeping close to the Standard metamodels: Core Model, P/T Systems and SC Nets. The main purpose is to provide a strong basis for future extensions, so that new concepts or features will be added by the import and inheritance mechanisms between packages. For this purpose, the Core Model mainly expresses an abstraction viewpoint for Petri nets. It keeps minimum information about the core concepts and their relationships. From a refinement and implementation point of view with EMF, a balance has to be found so as not to provide redundant and irrelevant information in the Core Model that would not be used in any other concrete package, while keeping conformance to the Standard specification.

A modular design of the major semantic fields such as: Core Model, Graphics metamodel, PT Net type, SC Net type and math specification, makes it easier to use, extend and maintain the packages. The metamodel for HLPNGs could as well specialise SC Net type as this latter specialises PT Net type. In this particular case, the modular design is of interest: the main extensions would be added to the math specification package.

The metamodel design corresponds to MDA's *Platform Independent Model* (PIM) approach [11], since, through this section, no platform preocupation was outlined. Some restrictions were put on the Standard metamodels, considering EMF's PIM design approach which is based on Ecore [5].

These restrictions constitute a refinement process on the Standard metamodels. Ecore is the metamodelling language for EMF. It is based on *Essential MOF 2.0*, a OMG supporting Standard [12] to help MDA taking shape. Once this stage completed, a *Platform Specific Model* (PSM) is generated, and code generation follows. The PSM takes into consideration a given platform specificities, allowing for parameterisation in order to facilitate code generation. EMF provides such a PSM for each Ecore Model.

In the following, the code generation process will be introduced.

4 Code generation

Code generation is generally one of the last steps in the model transformation process. For each metamodel, a tailored application programming interface (API) and its implementation are generated in Java programming language.

4.1 Code generation process

Figure 8 is a Dia diagram illustrating the code generation process. As could be expected, its starts from a Petri net Type metamodel specification at a high level of abstraction as defined in the Standard [8] (discussed in section 3.2). Then, thanks to EMF's Ecore language, a first model transformation is performed through refinements which led to the metamodels shown in section 3. According to MDA requirements, these metamodels correspond to Platform Independant Models (PIM) [11], as stated in section 3. From each PIM, a Platform Specific Model (PSM) known in EMF as a *Genmodel* is then generated. This is a second model transformation. This PSM targets EMF's Java-based platform.

As its MDA name suggests, platform specific concerns can be expressed on a PSM (aka Genmodel). Consequently, specific code generation parameters can be put on a Genmodel. We can therefore show to EMF code generation engine where to find our user-specific templates. Then with this EMF built-in code generator, based on Java Emitter Templates (JET) [15], a tailored API is generated.

With this tailored API, models can be created and manipulated. The tailored API also offers repository-oriented features such as fetching all transitions, all places and all arcs from a model. The native tailored API generated by EMF does not offer such features. These models are structured by the given Petri net Type metamodel. With JET, export and import API to/from PNML are also generated.

The framework APIs are structured into four virtual categories.

1. **Create**: the framework-based PNML models are created by the user when performing model transformation from its tool internal representation into PNML. They are also created by the import API (cf. Load).
2. **Save**: the framework-based PNML models are written into a PNML file.
3. **Load**: the user asks the framework to parse a PNML file and load the corresponding models. The models are then returned to the user in the framework-based representation.
4. **Fetch**: after having loaded a model, the user can fetch its elements. This API is also used by the export API (cf. Save).

4.2 Importing and exporting models.

Typically, a tool developer would provide a PNML file name to the import API and get in return the model(s) entry point object class, namely, *PetriNetDoc*. From this entry point, the developer could navigate into the model(s) to fetch information where needed, or directly use its repository-oriented features. During the import process, a log is written to report the encountered problems.

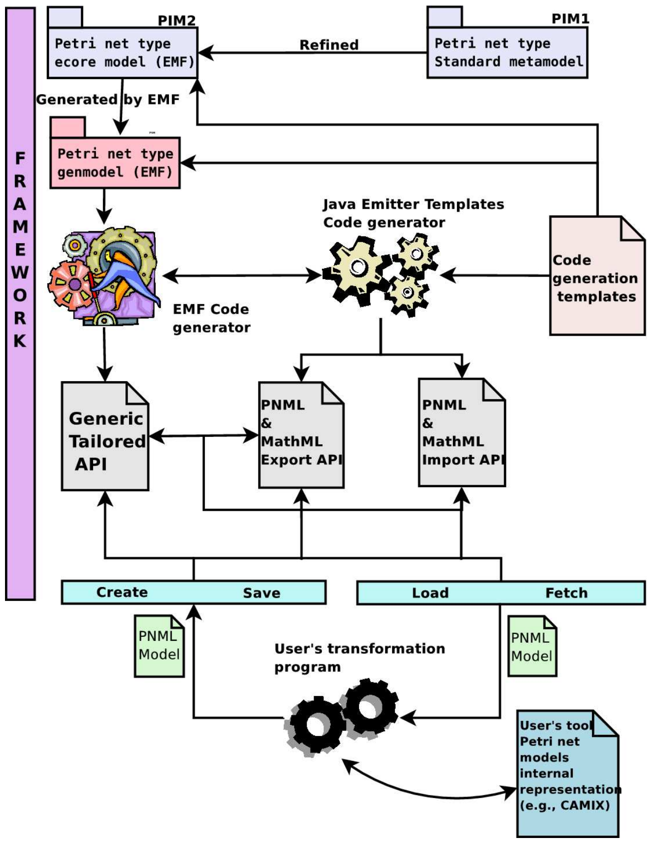
PIM2
Petri net type ecore model (EMF)
Refined
PIM1
Petri net type Standard metamodel
Generated by EMF
FRAMEWORK
Petri net type genmodel (EMF)
Java Emitter Templates Code generator
Code generation templates
EMF Code generator
Generic Tailored API
PNML & MathML Export API
PNML & MathML Import API
Create
Save
Load
Fetch
PNML Model
User's transformation program
PNML Model
User's tool Petri net models internal representation (e.g., CAMIX)

Figure 8: The code generation process.

The export process is the counterpart of the import one, with *PetriNetDoc* class as the input and a PNML file as the output. An application example with CPN-AMI is detailed in section 5.

5 Application to CPN-AMI

As mentionned in the introductory section of this paper (section 1), a tool developer point of view is adopted in order to evaluate the low-cost integration of the Standard accompanying framework to an existing tool. CPN-AMI [1] and his internal proprietary CAMI(X) format are taken as an example. In this section, a proprietary tool adaptation layer is written to integrate the Standard accompanying framework. A model transformation example is discussed using CPN-AMI [1], a LIP6 modelling CASE environment.

5.1 CPN-AMI: a Petri net design and verification environment

CPN-AMI specifies a set of modelling formalisms for Petri nets. The team working on Distributed and Cooperative Systems issues at LIP6 uses these formalisms for the design, verification, simulation and validation of complex systems.

CPN-AMI is integrated to a widely used and robust prototyping platform named *Framekit*, along with numerous tools to help handling efficiently the complex case studies. It has an intuitive graphical user interface named *Macao* [9], which is a powerful generic graph drawing software. With Macao, Petri net models are drawn and exchanged with the Framekit platform in a representation of the modelling formalisms that is called CAMI. CAMI is thus a formalism-based language to manipulate Petri net nodes, their annotations and their graphical aspect between Framekit, its integrated tools and Macao. When considering PT nets, we speak of *CAMI*. When considering SC nets, we speak of *CAMIX*, which is an extension of CAMI to represent abstract syntax trees for mathematical annotations on SC nets and higher level nets.

5.2 Adapting CPN-AMI to this framework : from Cami(x) to Pnml

To perform model transformation from CAMI(x) to PNML, we needed to develop the following components on the tool designer side:

- A CAMI(X) parser. It performs the parsing of CAMI(X) instructions.
- A CAMI(X) repository. It stores the parsed models.
- A component to populate the framework-based PNML repository instances with each Petri net model.
- A principal program which drives the above components and the nets builders provided by the framework.

Figure 9 depicts the model transformation process. First, a Petri net model contained in a CAMI(X) file is loaded and passed to the parser. The CAMI(X) repository is populated with the elements parsed from the model. Then a PNML repository instance is populated with the same model elements converted into the framework representation. Afterwards, the repository is passed to a net builder which returns the model built according to its metamodel, for example SC Net Type. Eventually, the model is attached to a *PetriNetDoc* object representing a PNML document and the store method is called on that object to write the model(s) in a PNML document.

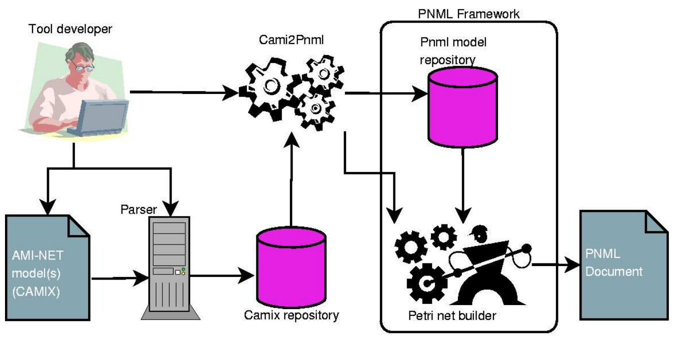

Figure 9: CAMI(x) to PNML model transformation process.

5.3 Adapting CPN-AMI to this framework : from Pnml to Cami(x)

To perform model transformation from PNML to CAMI(X), we had to develop the following programs:

- A program which fetches PNML models elements from PNML repository instances into CAMI(X) repository instances. Note that it is also possible to fetch the models elements into CAMI(X) repository instances by navigating directly within the returned models parsed from the PNML document.
- A program which builds models in CAMI(X) representation from the CAMI(X) repository.
- A principal program which drives the above ones and calls the PNML reader which parses PNML documents.

Figure 10 illustrates the model transformation process from PNML to CAMI(X). It follows the reverse path of the transformation from CAMI(X) to PNML.

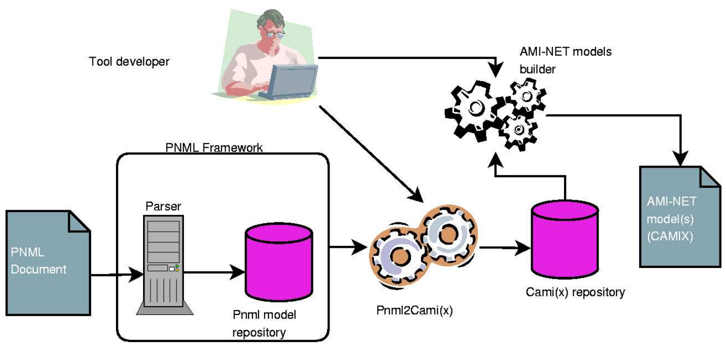

Figure 10: PNML to CAMI(x) model transformation process.

5.4 Time and effort required to develop the framework and the application

Writing CPN-AMI adaptation layer to integrate the framework was neither a complex task nor a trivial one. The adaptation layer architecture had to be carefully considered. Some technical designs and algorithms in programming tasks had also to be carefully considered, especially the ASTs for SC nets. Model-driven development approach was used to develop the adaptation layer. For CAMI(X) commands modeling it was very efficient so this approach is highly recommended.

Basically, a fortnight sufficed to develop this adaptation layer. Depending on how complex the tool designer internal representation is, the integration process to the framework will cost at most a month.

Generally speaking, the entire software development (including CPN-AMI adaptation) through its detailed stages from specifications to the first final release lasted about three months.

Synthesis

One of the primary goals by proposing this accompanying framework is to enable tools developers with low-cost integration of the Standard to their tools, in order to ease the Standard applicability. The application example described in this chapter shows that this objective is realistic, affordable and can be achieved at a reasonable cost.

The next release of CPN-AMI (version 3.0) will partially integrate a feature for the PNML exporting and importing capabilities that were developed in this section.

6 Guidelines to extending Petri Net Types metamodels

There are many variants of Petri nets, when considering each main Petri Net Type. The Standard has defined some of them, namely, P/T Systems, SC Nets [7] and HLPNGs [6]. However, tools developers may need to use these variants (e.g., P/T Systems with capacity, etc.) even if they are not yet formally defined. Therefore, it would be interesting to have means to extend the metamodels provided by the proposed framework in order to adapt the corresponding APIs while keeping some level of conformance to the Standard. To meet this expectation somehow, guidelines are provided to help tools developers to enhance the given metamodels. Following these guidelines ensures that the generated API will adapt to the extended metamodels.

In the following sections, we are going to use extensively metamodeling and EMF's Ecore vocabulary. However, this vocabulary is simple to understand, since some examples illustrate the discussion.

6.1 Extending a Petri net type metamodel by adding new metaclasses

The extension of a metamodel in this framework is guided by the following key hints:

1. An extension must be made mainly on *labels*. This makes sense since enrichment of known Petri net types is done so far on labels.

2. The new metaclasses must therefore inherit from **Annotation** or **Label** or **Attribute** metaclasses. By doing so, compliance is kept with the International Standard [8]. Figure 5 provides a good understanding example for P/T nets.

3. The new labels, if they are designed as new metaclasses, must preferably be added as *EReference containment* attributes to the net object they belong to, or at least as simple *EReference* attributes. This kind of relationship roughly corresponds to UML *composition* relationship [13]. **Example:** *initialMarking*, which is an Annotation (thus a label) is referenced from *Place* metaclass as an EReference containment attribute.

4. Since an extension mechanism may involve many metaclasses but not all are to be serialised, a naming convention called **Iso** must be applied on the new EReference attributes names. It consists of naming the attributes with the pattern **anyNameIsoPnmlElementName**, where 1) **anyName** is generally the package name, e.g., *pt*. It could also be any relevant name for the developer. 2) **Iso** is a pattern needed by code generation rules to determine PNML elements names corresponding to the attributes, 3)**PnmlElementName** is the *exact* spelling of the PNML element name for a given attribute.

5. Since metaclasses names have to reflect the corresponding PNML elements, the *String* on the right of *Iso* must be the *exact*[2] name of the corresponding PNML element.

6. New features of primitive types such as *String, Integer* and *Enumeration* will be processed. If they are non-EReference attributes, then they will become attributes of the PNML element corresponding to the metaclass they belong to.

Example: To add a capacity with lower and upper bounds to the *Place* metaclass, the following steps are required:

1. A new metaclass named **PTCapacity** is created and inherits from Annotation.

2. An attribute named *value* of Integer type is added to PTCapacity. The *value* name means that it must be serialised as *<text>value string representation</text>*, following the PNTD and Convention document rules as specified in the Standard [8].a

3. the *lower_bound* and *upper_bound* attributes of Integer type are added too. They will be serialised as attributes of the corresponding PNML element to PTCapacity.

4. A new EReference containment attribute is added to *Place* metaclass. Its type is PTCapacity and its name is *ptIsocapacity*. Consequently, **capacity** is the mapping for the corresponding PNML element name.

In practice, this extension example will have the following PNML representation:

```
<place id="plId3">
 .................
 <capacity lower-bound="1" upper-bound="4">
  <text>3</text>
 </capacity>
</place>
```

It should be noticed that the underscore in *lower_bound* and *upper_bound* has become a hyphen in their PNML representation. This must also be considered as a naming rule for compound names.

[2]Despite Java naming conventions. For example, in ptIsoinitialMarking, there is no capital I for *initial*.

6.2 Extending a Petri net type metamodel by a new metamodel

A very good example is to consider how SC Net extends PT Net. The main issue is to implement UML *merge* concept [13] as discussed in sections 3.3.1 and 3.4 via the *inheritance* and *import* mechanisms. In addition, all the rules stated in 6.1 must be applied.

Implementation must be provided by tools designers for new features that do not have one of the primitive types discussed in 6.1. For example, if new user-defined MathML functions are added to the math annotations metamodel, then the code generation templates must be updated.

7 Conclusion

Thanks to the future ISO/IEC 15909-2 International Standard, the lack of a common interchange format among Petri net tools is being adressed. This universal transfer syntax, PNML, is designed at a high level of abstraction with interoperability and extensibility as primary goals. To ease its applicability and low-cost integration to Petri net tools at a lower level and thus its wide use, an API framework is proposed to accompany the Standard.

The methodology used to design the framework is based on model-driven techniques, based on OMG's MDA initiative. Metamodels for the main Petri net types were designed, with a structure that enables future extensions, according to the ISO/IEC 15909-2 Standard philosophy. Model transformations were then implemented, powered by a templates-driven code generation engine.

The model transformations chain within this framework is completely achieved when tools developers plug their tool adaptation layer into it. An application example to CPN-AMI was introduced in 5.1. Actually, the proposed framework does not intend to substitute tools internal representations with its own. It serves as a mediator between tools and compliance to PNML Standard. Its main advantage if widely used will be to make a quick convergence of Petri net tools users and systems designers from academic and industrial worlds to the Standard compliance and facilitate fast re-engineering on the metamodels and code re-generation.
It provides a very comprehensive and intuitive API to use. It relies on mature and multi-platform technologies, namely Eclipse Java-based platform and Eclipse Modeling Framework, which are acknowledged by the industry. Eventually, its maintenance is facilitated by model-driven development approach.

This work will be input as a contribution to the annexes of the future ISO/IEC 15909-2 Standard.

Perspectives

So far, the API framework has been integrated into a Petri net CASE environment, namely CPN-AMI [1]. The proposed approach should now be investigated and extended with applications to other Petri net tool such as Great SPN [4] and CPN Tools [10]. Through many application examples, specific needs for flexibility and new functionalities will arise, thus improving the framework. The applications will also help to validate this working plan.

Constraints checking on the models is not yet mandatory. A dedicated API is provided with the proposed framework. It can be turned on/off by the user. Nevertheless, it does not mean that a malformed model, w.r.t. its metamodel, is accepted. The constraints we are speaking of are concerned with fundamental semantics in Petri nets, e.g., connecting two places. The current constraints checking approach is specified by programming. Future work should seek for OCL [14] technologies to implement these constraints at the metamodels level. Current OCL technologies are not yet mature enough to integrate them into the framework at this early stage. However, implementing these constraints at the metamodel level implies a debate in the Petri net community to agree on, set and acknowledge them.

The framework has been developed with Java programming language. However, with the templates-driven code generator, APIs could be generated in any other language. In this particular case, the entire generated tailored APIs must be substituted with other interfaces and implementation packages in the targeted language (e.g., C++ or ADA).

Acknowledgement

We would like to thank ISO/IEC 15909 editors, Jonathan Billington and Ekkart Kindler, for insightful discussions at the PNML workshop in Helsinki that have helped enhancing this work.

References

[1] *The CPN-AMI Home page.* `http://www.lip6.fr/cpn-ami`.

[2] J. Billington, S. Christensen, K. van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber. The Petri Net Markup Language: Concepts, technology and tools. In *Proc. 24th Int. Conf. Application and Theory of Petri Nets (ICATPN'2003), Eindhoven, The Netherlands, June 2003*, volume 2679 of *Lecture Notes in Computer Science*, pages 483–505. Springer, 2003.

[3] J. Clark. *RELAX NG Home Page*. `http://www.relaxng.org/`, 2003.

[4] Universita di Torino. *Great SPN*. `http://www.di.unito.it/~greatspn/index.html/`.

[5] Eclipse Foundation. *Eclipse Modeling Framework*. `http://www.eclipse.org/emf/`.

[6] ISO/IEC. Software and Systems Engineering - High-level Petri Nets, Part 1: Concepts, Definitions and Graphical Notation, International Standard ISO/IEC 15909, December 2004.

[7] ISO/IEC. Software and Systems Engineering - High-level Petri Nets, Part 1: Concepts, Definitions and Graphical Notation. Proposed Draft Addendum to International Standard ISO/IEC 15909-1, June 2005.

[8] E. Kindler. Software and Systems Engineering - High-level Petri Nets. Part2: Transfert Format. Working Draft for the International Standard ISO/IEC 15909 Part 2 - Version 0.6.3, June 2005.

[9] *The Macao Home page.* `http://www.lip6.fr/macao/`.

[10] University of Aarhus. *Computer Tool for Coloured Petri Nets - CPNTool*. `http://wiki.daimi.au.dk/cpntools/cpntools.wiki`.

[11] OMG. MDA Guide Version 1.0.1, document no: omg/2003-06-01, 2003.

[12] OMG. *MetaObjectFacility 2.0 Core Specification, document no:omg/2003-10-04*. OMG, 2003.

[13] OMG. *UML 2.0 Infrastructure Specification, document no:omg/03-09-15*. OMG, 2003.

[14] OMG. *UML 2.0 OCL Specification, document no:omg/03-10-14*. OMG, 2003.

[15] R. Popma. *JET tutorials part 1 (Introduction to JET) and part 2 (Write code that writes code)*. Azzurri Ltd., `http://eclipse.org/emf/docs.php`, 2005.

[16] W3C. *The EXtensible Markup Language*. `http://www.w3.org/XML/`.

[17] W3C. *MathML 2.0, W3C Math Home*. W3C, `http://www.w3.org/Math/`.

[18] M. Wahler. *Using OCL to interrogate your EMF model.* IBM Zurich, `http://www.zurich.ibm.com/~wah/doc/emf-ocl/index.html`, August 2004.

[19] M. WEBER. *PNML*. `http://www.informatik.hu-berlin.de/top/pnml/pnml.html`.