INTERNATIONAL STANDARD

ISO/IEC 15909-2

First edition
2011-02-15

Systems and software engineering — High-level Petri nets —

Part 2: Transfer format

Ingénierie des systèmes et du logiciel — Réseaux de Petri de haut niveau —

Partie 2: Format de transfert

Reference number
ISO/IEC 15909-2:2011(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Contents

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75% of the national bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC 15909-2 was prepared by Joint Technical Committe ISO/IEC JTC 1, *Information technology*, Subcommittee SC 7, *Software and systems engineering*.

ISO/IEC 15909 consists of the following parts, under the general title *Systems and software engineering – High-level Petri nets*:

— *Part 1: Concepts, definitions and graphical notation*

— *Part 2: Transfer format*

“Extensions” will form the subject of a future Part 3.

Introduction

ISO/IEC 15909 is concerned with defining a modelling language and its transfer format, known as *High-level Petri Nets*. ISO/IEC 15909-1 provides the mathematical definition of *High-level Petri Nets*, called the semantic model, the graphical form of the technique, known as *High-level Petri Net Graphs* (HLPNGs), and its mapping to the semantic model. It also introduces some common notational conventions for HLPNGs.

This part of ISO/IEC 15909 defines a transfer format for *High-level Petri Nets* in order to support the exchange of *High-level Petri Nets* among different tools. This format is called the *Petri Net Markup Language* (*PNML*). Since there are many different versions of *Petri nets* in addition to *High-level Petri Nets*, this part of ISO/IEC 15909 defines the *core concepts* of *Petri nets* along with an XML syntax, which can be used for exchanging any kind of *Petri net*. Based on this *PNML Core Model*, this part of ISO/IEC 15909 also defines the transfer syntax for the three versions of *Petri nets* that are defined in ISO/IEC 15909-1: *Place/Transition Nets*, *Symmetric Nets*[1], and *High-level Petri Nets*, where *Place/Transition Nets* and *Symmetric Nets* can be considered to be restricted versions of *High-level Petri Nets*. For *Place/Transition Nets*, this part of ISO/IEC 15909 introduces two different transfer formats: one is a format specifically tuned to *Place/Transition Nets*, the other is a format that represents *Place/Transition Nets* as a restricted version of *High-level Petri Nets* as defined in ISO/IEC 15909-1.

The basic level of conformance to this part of ISO/IEC 15909 is to the *PNML Core Model*. The other levels are according to the particular type of the *Petri net*; for *High-level Petri Nets* there are two levels of conformance: *textual conformance* ignores the exact syntax and structure of the *labels*; *structural conformance* requires that *labels* are given in the exact syntax as defined here. Since *Symmetric Nets* are designed for analysability, *textual conformance* does not make any sense for *Symmetric Nets*; therefore, there is only *structural conformance* for *Symmetric Nets*.

Note that this part of ISO/IEC 15909 introduces some concepts that are not defined in ISO/IEC 15909-1. These concepts are not related to the mathematical concepts of *Petri nets* and their semantics. They concern the graphical representation of nets and the structuring of large *Petri net* models. These concepts need to be defined, along with a transfer format for *Petri nets*, in order to ensure that the graphical appearance of a *Petri net* in different tools is similar.

This part of ISO/IEC 15909 is structured as follows: Clause 1 describes the scope, the areas of application and the intended audience of this part of ISO/IEC 15909. Clause 2 defines conformance. Clause 3 gives references that are essential for the correct interpretation of this International Standard. Clause 4 defines all terms relevant to this International Standard and includes a list of abbreviations. Clause 5 introduces the concepts of PNML using UML meta models. Clause 5.2 defines the *PNML Core Model*, which is the structure common to all versions of *Petri nets*. Clause 5.3 defines the particular concepts of the different *Petri net types*. Clause 6 provides the mapping of the syntactical concepts defined in this part of ISO/IEC 15909 to the concepts defined in ISO/IEC 15909-1. Clause 7 defines how the concepts of PNML as defined in Clause 5 are mapped to XML syntax.

Annex A defines the exact XML syntax for the *PNML Core Model* in terms of a RELAX NG grammar. Annex B defines the exact XML syntax for the different types of *Petri nets*. Annex C provides a small example for the syntax of a symmetric net. Annex D discusses a framework for implementing this International Standard and an API for accessing *Petri nets*, which is based on the UML models for the PNML meta models.

[1] Symmetric nets were first introduced as well-formed nets and are currently standardized as ISO/IEC 15909-1:2004/Amd. 1:2010.

Systems and software engineering – High-level Petri nets – Part 2: Transfer format

1 Scope

This part of ISO/IEC 15909 defines an XML-based transfer format for *Petri nets*, which are defined conceptually and mathematically in ISO/IEC 15909-1. This transfer format enables the exchange of *Petri nets* among different *Petri net* tools and among different parties. Moreover, this part of ISO/IEC 15909 defines some concepts and XML-based syntax for defining the detailed graphical appearance of *Petri nets*.

The focus of this part of ISO/IEC 15909 is on the transfer format for *Place/Transition Nets*, *High-level Petri Nets* and *Symmetric Nets*. The presentation, however, is structured in such a way that it is open for future extensions, so that other versions of *Petri nets* can be added later. The exact definition of this extension mechanism, called *Petri net type definition*, is not defined in this part of ISO/IEC 15909; it will be defined in ISO/IEC 15909-3.

The transfer format will be used to transfer specifications of systems developed in *High-level Petri Nets* between tools to facilitate the development of systems in teams.

This part of ISO/IEC 15909 is written as a reference for developers of *Petri net* tools. Moreover, it will be useful for researchers who define new versions and variants of *Petri nets*.

2 Conformance

There are different levels of conformance to this part of ISO/IEC 15909. All conformance levels impose additional conditions on valid XML documents.

2.1 PNML Documents

An XML document is conformant to the *PNML Core Model* if it meets the definitions of Clause 5.2 (concepts) and Clause 7.1 (their mapping to XML syntax) – such a document is called a *PNML Document* or a *Petri Net Document*. A Petri net tool is conformant to the *PNML Core Model* if it can import all *PNML Documents* and if it can export all *Petri nets* to a *PNML Document*.

The other levels of conformance concern the different *Petri Net Types*.

2.2 PNML Place/Transition Net Documents

A *PNML Document* is a conformant *Place/Transition Net* if it meets the additional restrictions of Clause 5.3.1 (concepts of P/T-nets) and Clause 7.2 (their mapping to XML syntax) – such a document is called a *PNML Place/Transition Net Document*. A Petri net tool is conformant to the *PNML Place/Transition Net* definition if it can import all *PNML Place/Transition Net Documents*, and if it can export all *Place/Transition Nets* to *PNML Place/Transition Net Documents*. Note that this transfer format is tuned to *Place/Transition Nets*. There is another format, which considers *Place/Transition Nets* as a restricted form of *High-level Petri Nets* (see Clause 2.5).

2.3 Textually conformant PNML High-level Petri Net Documents

For *High-level Petri Nets*, there are two different levels of conformance. The first level requires the existence of textual labels as defined in Clause 5.3.2 and 5.3.11 (concepts) and Clause 7.3 (mapping to XML syntax). But,

it does not require the existence of the structural parts of the *annotations*; it only requires that the textual parts of the *annotations* exist, but the text is not required to be in a specific syntax and, therefore, the meaning of it cannot be transferred to other tools. Such a *PNML Document* is called a *textually conformant PNML High-level Petri Net Document*. A Petri net tool is conformant to the *textual PNML High-level Petri Net* definition if it can import all *textually conformant PNML High-level Petri Net Documents*, and if it can export all *High-level Petri Nets* to a *textually conformant PNML High-level Petri Net Document*.

2.4 Structurally conformant PNML High-level Petri Net Documents

The second level of *High-level Petri Net* conformance requires that all *annotations* obey the rules defined in Clause 5.3.2 and 5.3.11 (concepts) and Clause 7.3 (mapping to XML syntax). Such a *PNML Document* is called a *structurally conformant PNML High-level Petri Net Document*. A Petri net tool is conformant to the *structural PNML High-level Petri Net* definition if it can import all *structurally conformant PNML High-level Petri Net Documents*, and if it can export all *High-level Petri Nets* to a *structurally conformant PNML High-level Petri Net Document*.

2.5 Place/Transition Net Document in High-level Notation

A *structurally conformant PNML High-level Petri Net Document* that uses only the single sort *dot* and only the *arc annotations* of *Place/Transition Nets* is a *conformant Place/Transition Net Document in High-level Notation*.

2.6 Symmetric Net Documents

Finally, there is conformance to *Symmetric Nets*, which is a restricted version of *High-level Petri Nets*. A *Symmetric Net Document* is a *structurally conformant PNML High-level Petri Net Document* if it contains only the concepts defined in Clause 5.3.10. A Petri net tool is conformant to the *Symmetric Net* definition if it can import all *Symmetric Net Documents*, and if it can export all *Symmetric Nets* to *Symmetric Net Documents*.

3 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 15444 (all parts), *Information technology – JPEG 2000 image coding system*

ISO/IEC 15909-1, *Systems and software engineering – High-level Petri nets – Part 1: Concepts, definitions and graphical notation*

ISO/IEC 15948, *Information technology – Computer graphics and image processing – Portable Network Graphics (PNG): Functional specification*

ISO/IEC 19757-2:2008, *Information technology – Document Schema Definition Language (DSDL) – Part 2: Regular-grammar-based validation – RELAX NG*

CSS, *Cascading Style Sheets, level 2 revision 1, CSS 2.1 Specification ; W3C Candidate Recommendation, 25 February 2004*

OCL 2.0, *Object Constraint Language, OMG Available Specification, Version 2.0. OMG formal/06-05-01, May 2006*

UML 2.1, *OMG Unified Modeling Language (OMG UML): Superstructure, V2.1.2 OMG Available Specification, November 2007*

XML 1.1, *Extensible Markup Language (XML) 1.1 (Second Edition); W3C Recommendation, 29 September 2006*

XML Schema Datatypes: *XML Schema Part 2: Datatypes (Second Edition); W3C Recommendation, 28 October 2004*

4 Terms, definitions and abbreviations

4.1 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 15909-1 and the following apply.

4.1.1
annotation
label (4.1.5) represented as text near to the **object** (4.1.7) it is associated with

4.1.2
attribute
label (4.1.5) that governs the form or shape of the **object** (4.1.7) it is associated with, which, in contrast to an **annotation** (4.1.1), is typically not shown as text

4.1.3
graphical information
information defining the graphical appearance of **objects** (4.1.7) and **labels** (4.1.5) of a **net graph**, which can be the position, size, line colour, fill colour, font or line width

4.1.4
global label
label (4.1.5) associated with the **net graph** itself, rather than with an **object** (4.1.7) of a **net graph**

4.1.5
label
information associated with the **net graph** or one of its **object** (4.1.7)

4.1.6
meta model
model defining the concepts and their relations for some modelling notation

NOTE The meta models in this part of ISO/IEC 15909 are mainly UML class diagrams.

4.1.7
object (of a net graph)
arc, **node**, **reference node** (4.1.14), or **page** (4.1.8) of a **net graph**

4.1.8
page
structuring mechanism used to split a large **net graph** into smaller parts, which are also the units of the net to be printed

4.1.9
PNML Core Model
meta model (4.1.6) defining the basic concepts and structure of **net graph** models that are common to all versions of **Petri nets**

4.1.10
PNML Document (Petri Net Document)
XML document that contains one or more **net graphs**

4.1.11
PNML High-level Net Document
PNML Document (4.1.10) that contains one or more **net graphs**, where all **net graphs** conform to **High-level Petri Nets**

4.1.12
PNML Place/Transition Net Document
PNML Document (4.1.10) that contains one or more **net graphs**, where all **net graphs** conform to **Place/Transition Nets**

4.1.13
PNML Symmetric Net Document
PNML Document (4.1.10) that contains one or more **net graphs**, where all **net graphs** conform to **Symmetric Nets**

4.1.14
reference node
node of a **Petri net** that is a representative of another **node**, possibly defined on another **page** (4.1.8) of the **net graph**

4.1.15
reference place
reference node (4.1.14) that represents a **place** and refers to either another **reference place** (4.1.15) or to a **place**

4.1.16
reference transition
reference node (4.1.14) that represents a **transition** and refers to either another **reference transition** (4.1.16) or to a **transition**

4.1.17
source node
node associated with the start of an **arc**

4.1.18
target node
node associated with the end of an **arc**

4.1.19
tool specific information
information associated with an **object** (4.1.7) of a **net graph** or with the **net graph** itself that is specific to a particular tool and is not meant to be used by other tools

4.2 Abbreviations

4.2.1
CSS
Cascading Stylesheets

4.2.2
CSS2
Cascading Stylesheets level 2

4.2.3
OCL
Object Constraint Language

4.2.4
P/T Net
Place/Transition Net

4.2.5
PNML
Petri Net Markup Language

4.2.6
RELAX NG
Regular Language description for XML/ New Generation

4.2.7
SN
Symmetric Net

4.2.8
UML
Unified Modeling Language

4.2.9
XML
eXtensible Markup Language

5 Concepts

A *Petri net* can be considered to be a labelled graph, where the particular *labels* depend on the particular type of *Petri net*. The concepts for *High-level Petri Net Graphs* are defined in part 1 of ISO/IEC 15909 and are represented in terms of a meta model in UML notation in this clause. The mapping of these concepts to XML syntax is defined in Clause 7.

Some concepts, however, have not been defined in part 1 because the main concern of part 1 was on the semantics of *High-level Petri Nets*. Part 2 of ISO/IEC 15909 defines concepts for defining the graphical appearance of *Petri nets* in order to enable similar graphical representation in different tools. Moreover, part 2 defines the concept of *pages* in order to be able to split the graphical representation of a *Petri net* into smaller units for printing and viewing them.

Note that the concepts and definitions presented in part 1 of ISO/IEC 15909 are semantic in nature, whereas part 2 is syntactic in nature. For example, part 1 refers to *types*, which are sets, but does not mandate any concrete syntax for transferring such sets. Likewise part 1 uses *functions*, but does not mandate a syntax for transferring them. In order to transfer *High-level Petri Nets*, this part of ISO/IEC 15909 needs to introduce a syntax for *types* and *functions*. To this end, it uses the concepts of *sorts* and *operators*. These are syntactic symbols for denoting *types* and *functions*, and are assigned a fixed interpretation, which is defined along with the *sorts* and *operators*. Therefore, this part of ISO/IEC 15909 often refers to *sorts* and *operators*, where part 1 refers to their semantical counterpart, *types* and *functions*. By the fixed interpretation of *sorts* and *operators*, the respective *types* and *functions* are also defined.

5.1 General Principles

This part of ISO/IEC 15909 defines a transfer format for *Place/Transition Nets*, *High-level Petri Net Graphs*, and *Symmetric Nets* as defined in part 1 of ISO/IEC 15909. This transfer format has been designed to be extensible and to be open for future variants of *Petri nets* and possibly for other use, such as the transfer of results associated with the analysis of Petri nets, e. g., reachability graphs. In order to obtain this flexibility, the transfer format considers a *Petri net* as a labelled directed graph, where all type specific information of the net is represented in *labels*. A *label* may be associated with a *node*, an *arc*, a *page* or the *net* itself. This basic structure is captured in the *PNML Core Model*, which is defined in Clause 5.2.

The *PNML Core Model* is presented using UML class diagrams. Note that these UML diagrams do not define the concrete XML syntax for *PNML Documents*; rather, they define the abstract syntax only. The mapping of the *PNML Core Model* elements (i. e., the abstract syntax) to the concrete XML syntax will be defined in Clause 7; Annex A defines the exact XML syntax in terms of a RELAX NG grammar.

The *PNML Core Model* imposes no restrictions on *labels*. Therefore, the *PNML Core Model* can represent any kind of *Petri net*. Due to this generality of the *PNML Core Model*, there can be *PNML Documents* that do not correspond to a *Petri net* at all. For example, there could be *labels* from two different and even incompatible versions of *Petri nets* within the same *PNML Document*. For a concrete version of *Petri nets*, the legal *labels* are defined by extending the *PNML Core Model* with another meta model that exactly defines the legal *labels* of this type.

Technically, the *PNML Core Model* is a UML package, and there are additional UML packages for the different *Petri net types* that extend the *PNML Core Model* package. This part of ISO/IEC 15909 defines a package for *Place/Transition Nets*, a package for *Symmetric Nets*, and a package for *High-level Petri Net Graphs*, where the package for *High-level Petri Net Graphs* extends the package for *Symmetric Nets*. Therefore, every *Symmetric Net* is also a *High-level Petri Net Graph*. The representation of these concepts in XML syntax is defined in Clause 7.2; Annex B defines the exact XML syntax of the concepts for the different types in terms of RELAX NG grammars.

Figure 1 shows the different packages and how they are related. The package *PNML Core Model* defines the basic structure of *Petri nets* or *net graphs*; this structure is extended by the package for each type. The *PNML Core Model* is defined in Clause 5.2, the package PT-Net is defined in Clause 5.3.1, the package SymmetricNet is defined in Clause 5.3.10 and the package HLPNG is defined in Clause 5.3.11. Clause 7 shows how the concepts defined in these packages are represented in concrete XML syntax.

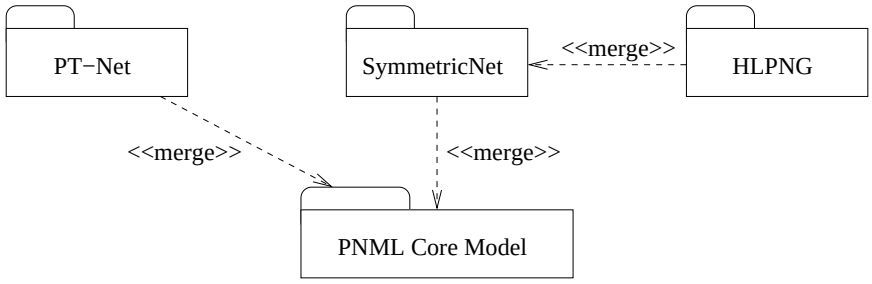

Figure 1: Overview of the UML packages of PNML

5.2 PNML Core Model

Figures 2 and 3 show the *PNML Core Model* as a UML class diagram. The diagram of Fig. 2 focuses on the conceptual parts, whereas the diagram of Fig. 3 focuses on the parts concerning the graphical representation. Note that the data type *String* is imported from a separate package *XMLSchemaDataTypes*, which is discussed in Clause 5.2.6.

The concepts defined in the *PNML Core Model* are discussed in the following clauses.

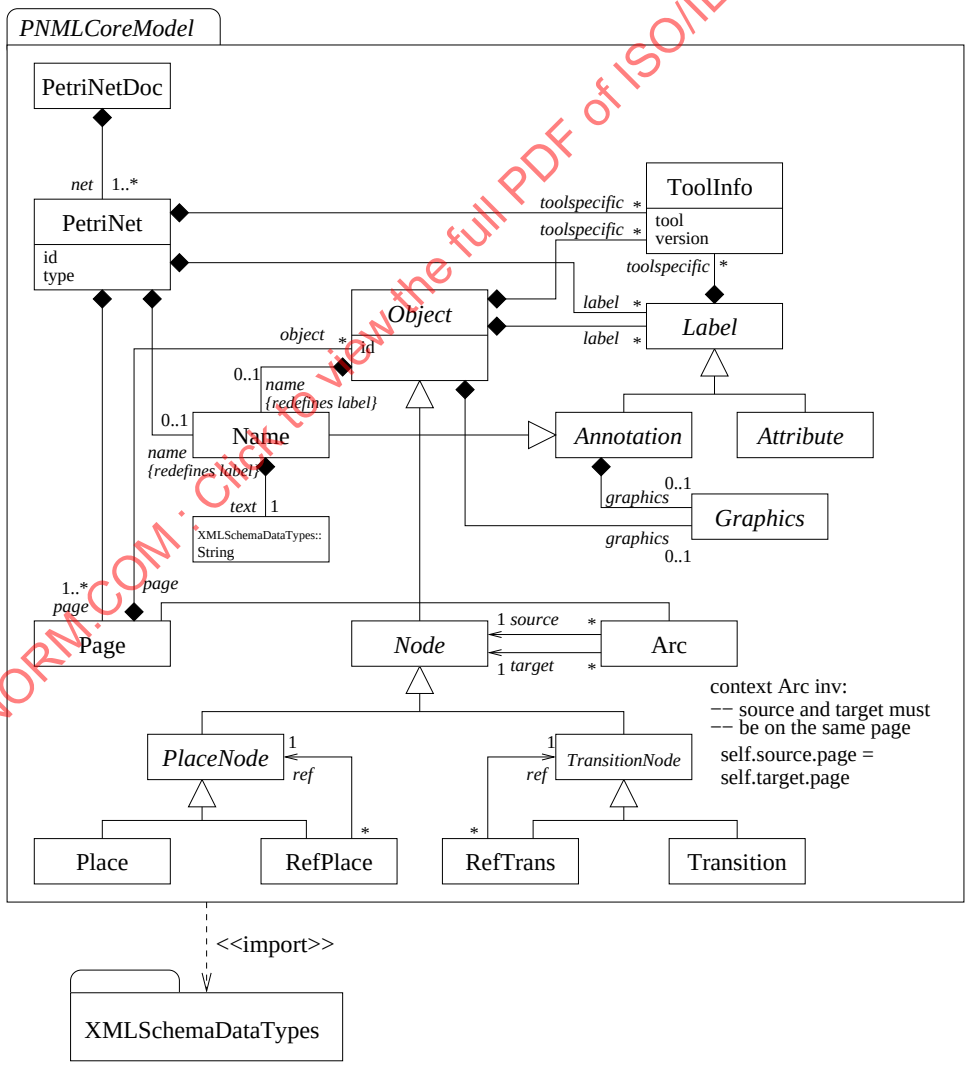

Figure 2: The *PNML Core Model* package: concepts

5.2.1 Petri Net Documents, Petri Nets, and Objects

A document that meets the requirements of the *PNML* Core Model is called a *Petri Net Document* (PetriNetDoc) or a *PNML Document*. It contains one or more *Petri Nets* (PetriNet). Each *Petri Net* has a unique identifier and a *type*. The *type* is a name referring to the package with its definition; an example for such a package name is `http://www.pnml.org/version-2009/grammar/ptnet` for the package defining *P/T Nets*. The URL for the definition of PNML itself is `http://www.pnml.org/version-2009/grammar/pnml`.

A *Petri net* consists of one or more top-level *pages* that in turn consist of several *objects*. These *objects*, basically, represent the graph structure of the *Petri net*. Each *object* within a *Petri net document* has a unique *identifier*, which can be used for referring to this *object*. Moreover, each *object* may be equipped with graphical information defining its position, size, colour, shape and other attributes on its graphical appearance. The precise graphical information that can be provided for an *object* depends on the particular type of *object* (see Clause 5.2.4 and Fig. 3).

The main *objects* of a *Petri net* are *places*, *transitions* and *arcs*. For convenience, a *place* or a *transition* is generalized to a *node*. For reasons explained in Clause 5.2.2 below, this generalization is via *place nodes* and via *transition nodes*. *Nodes* of a *Petri net* can be connected by *arcs*.

Note that it is legal to have an *arc* from a *place* to a *place* or from a *transition* to a *transition* according to the *PNML Core Model*. The reason is that there are versions of *Petri nets* that support such *arcs*. If a *Petri net type* does not support such *arcs*, this restriction will be defined in the particular package defining this type.

5.2.2 Pages and Reference Nodes

Three other kinds of *objects* are used for structuring a *Petri net*: *pages*, *reference places*, and *reference transitions*. As mentioned above, a *page* may contain other *objects*; since a *page* is an *object* itself, a *page* may even contain other *pages*, which defines a hierarchy of subpages.

This part of ISO/IEC 15909 requires that an *arc* must connect *nodes* on the same *page* only. The reason for this requirement is that *arcs* connecting *nodes* on different *pages* cannot be drawn graphically on a single *page*. In the *PNML Core Model* of Fig. 2, this requirement is captured by the OCL expression (OCL 2.0, OMG) next to the class for *arcs*.

In order to connect *nodes* on different *pages* by an *arc*, a representative of one of the two *nodes* is drawn on the same *page* as the other *node*. Then, this representative may be connected with the other *node* by an *arc*. This representative is called a *reference node*, because it has a reference to the *node* it represents. Note that a *reference place* must refer to a *place* or a *reference place*, and a *reference transition* must refer to a *reference transition* or a *transition*. Moreover, cyclic references among *reference nodes* are not allowed. Though this requirement cannot be expressed in UML or OCL notation, this requirement is mandatory.

The concepts of *pages* and *reference nodes* are not defined in part 1 of ISO/IEC 15909. The reason for introducing them in part 2 is that these concepts are needed for graphical and structuring purposes. Semantically, they do not carry any meaning. The meaning of these concepts is defined by merging each *reference node* with the *node* it ultimately refers to and by simply ignoring the *pages*. This procedure is called *flattening* of the page structure.

5.2.3 Labels

In order to assign further meaning to an *object*, each *object* may have *labels*. Typically, a *label* represents the name of a *node*, the initial marking of a *place*, the *transition condition*, or some *arc annotation*. In addition, the *Petri net* itself or its *pages* may have some *labels*, which are called *global labels*. For example, the package HLPNG defines *declarations* as *global labels* of a *High-level Petri Net*, which are used for defining *variables*, and user-defined *sorts* and *operators*.

This part of ISO/IEC 15909 distinguishes two kinds of *labels*: *annotations* and *attributes*. An *annotation* comprises information that is typically displayed as text next to the corresponding *object*. Examples of *annotations* are names, initial markings, arc annotations, transition conditions, and timing or stochastic information.

In contrast, an *attribute* is, typically, not displayed as text next to the corresponding *object*. Rather, an *attribute* has an effect on the shape or colour of the corresponding *object*. For example, an *attribute* such as arc type could have domain { `normal`, `read`[2], `inhibitor`, `reset` }. This part of ISO/IEC 15909, however, does not mandate the effect that an *attribute* has on the graphical appearance of a net *object*.

Note that the classes for *label*, *annotation* and *attribute* are abstract in the *PNML Core Model*, which means that the *PNML Core Model* does not define concrete *labels*, *annotations*, and *attributes*. The only concrete *label* defined in the *PNML Core Model* is the *name*, which is a *label* that can be used for any *object* within any *Petri net type*. Note that, this way, any *object* – including *nodes*, *pages*, even the *net* itself as well as *arcs* – can have a *name*. The value of a *name* is a *String*, which is imported from the separate package *XMLSchemaDataTypes* (see Clause 5.2.6 for more information). The other concrete *labels* will be defined in the packages for the concrete *Petri net types* (see Clause 5.3).

In order to support the exchange of information among tools that have different textual representation for the same concepts (i. e. if they have different concrete syntax), there are two ways for representing the information within an annotation: *textually* in some concrete syntax and *structurally* as an abstract syntax tree (see Clauses 5.3.11 and 7.1.2 for details).

Note that *reference nodes* may have *labels*, but these *labels* do not have any effect on the semantics of the net and can be completely ignored semantically. Since the labels of a *reference node* do not have an effect on the semantics, it is easy to obtain a semantically equivalent *Petri net* without *pages* by merging every *reference node* to the *node* it directly or indirectly refers to. This is called *flattening* of the *Petri net*. But, the labels of a *reference node* can have an effect on the graphical appearance or can give some additional information to the user.

5.2.4 Graphical Information

In addition to the *Petri net* concepts, information concerning the graphical appearance can be associated with each *object* and each *annotation*. For a *node*, this information includes its position; for an *arc*, it includes a list of positions that define intermediate points of the *arc*; for an *object's annotation*, it includes its relative position with respect to the corresponding *object*; and for an *annotation* of a *page*, the position is absolute. There can be further information concerning the size, colour and shape of nodes or arcs, or concerning the colour, font and font size of labels. Note that this information can be used for automatically transforming a *Petri Net* into *Scalable Vector Graphics* (SVG) by XSLT transformations (see [8] for more details). This transformation, however, is not defined in this part of ISO/IEC 15909.

Figure 3 shows the different graphical information that can be attached to the different types of *objects* and the different attributes. Note that this still belongs to the *PNML Core Model*; it is shown in a different figure only for better understandability. Table 1 gives an overview of the meaning and the domain of the attributes of the different graphical features. The exact XML syntax is defined in Clause 7.1.3.

A *position* element defines the absolute position for *nodes* and *pages*. For an *annotation* an *offset* element defines its relative position to the *object* it is attached to – if it is a *global annotation*, the *offset* also defines the absolute position on that *page*. Each absolute or relative position consists of a pair of Cartesian coordinates (x, y), where the units are points (pt). As for many graphical tools, the x-axis runs from left to right and the y-axis from top to bottom. And the reference point for the position of an *object* is its centre.

For an *arc*, the (possibly empty) sequence of *positions* defines its intermediate points (bend points). Note that the positions of the *start point* and the *end point* of an *arc* are not given explicitly for the arc. These positions are determined from the position of the source and the target *node* of the *arc* and the direction of the respective segment of the *arc*. They are defined by the line starting or ending at the reference point of the respective node in the direction of the arc and the intersection with the border of that node. Altogether, the *arc* is displayed as a path from the *start point* on the border of the *source node* to the *end point* on the border of the *target node* via the intermediate points. Depending on the value of the attribute *shape* of element *line*, the path is displayed as a broken *line* (polyline) or as a quadratic Bezier *curve*. In the case of a Bezier curve the intermediate positions alternately are the line connectors or Bezier control points. The reference point of an arc is the middle of the arc, which is the middle of the middle segment of the arc, if there are an odd number of segments; it is the middle point, if there are an even number of segments.

[2]Sometimes, a read arc is called a test arc.

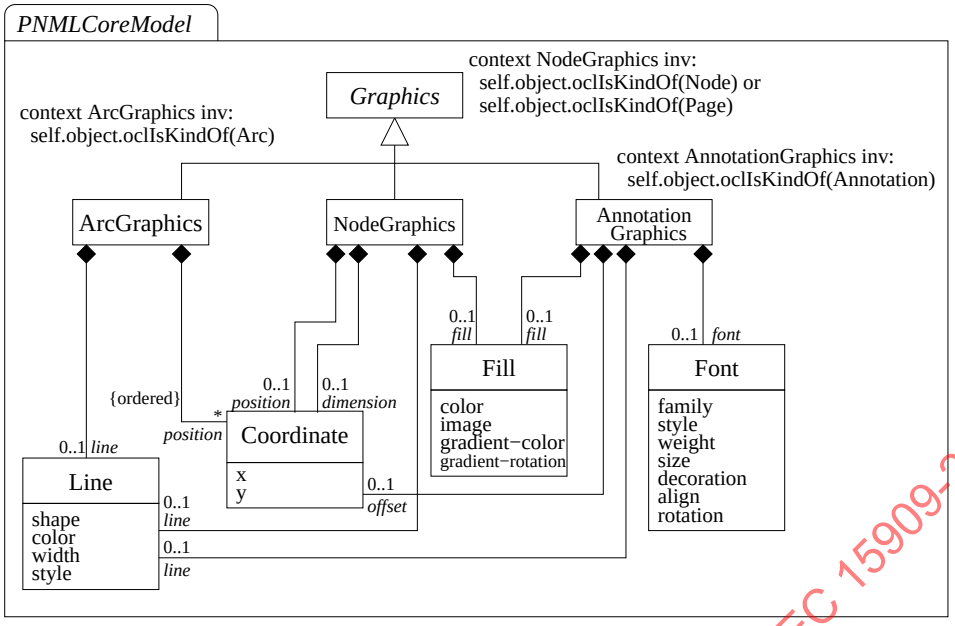

Figure 3: The *PNML Core Model*: graphical information

Table 1: PNML attributes of graphical information

Class	Attribute	Domain
Coordinate	x	decimal
	y	decimal
Fill	color	CSS2-color
	image	anyURI
	gradient-color	CSS2-color
	gradient-rotation	{vertical, horizontal, diagonal}
Line	shape	{line, curve}
	color	CSS2-color
	width	nonNegativeDecimal
	style	{solid, dash, dot}
Font	family	CSS2-font-family
	style	CSS2-font-style
	weight	CSS2-font-weight
	size	CSS2-font-size
	decoration	{underline, overline, line-through}
	align	{left, center, right}
	rotation	decimal

Although there is no way to define the *start point* and the *end point* of an arc explicitly, the above concepts allow a user to make sure that an arc starts and ends exactly where it should. If the first, (or last) intermediate point of an arc lies exactly on the border of the respective node (or very close to it), this will be the *start point* (or the *end point* respectively) of that arc.

For an *arc*, there are some *line* attributes, which define the *style*, the *colour*, and the *width* in which it is displayed.

The *dimension* of a *node* or *page* gives its size, again as a pair referring to its width (x) and height (y). Depending on the ratio of height and width, a *place* is displayed as an ellipse rather than a circle. A *transition* is displayed as a rectangle of the corresponding size. If the dimension of an element is missing, each tool is free to use its own default value for the dimensions.

The two elements *fill* and *line* define the interior and outline colours of the corresponding element. The value assigned to a *color* attribute must be a RGB value or a predefined colour as defined by CSS2. When the attribute

gradient-color is defined, the fill colour continuously varies from color to gradient-color. The additional attribute *gradient-rotation* defines the orientation of the gradient. When the attribute *image* is defined, the node is displayed as the image which is provided at the specified URI, which must be a graphics file in JPEG (ISO/IEC 15444) or PNG (ISO/IEC 15948) format. In this case, all other attributes of *fill* and *line* are ignored.

For an *annotation*, the *font* element defines the font used to display the text of the *label*. The attributes *family*, *style*, *weight*, and *size* are CSS2 attributes for defining the appearance of the text. The detailed description of the possible values of these attributes and their effect can be found in the CSS2 specification (CSS 2.1 Specification). The *decoration* attribute defines additional properties of the appearance of the text such underlining, overlining, or striking-through the text. Additionally, the *align* attribute defines the alignment of the text to the *left*, *right* or *center*. The *rotation* attribute defines a clockwise rotation of the text.

The reference point of an annotation is always its centre.

5.2.5 Tool Specific Information

For some tools, it might be necessary to store *tool specific information* (ToolInfo), which is not meant to be used by other tools. In order to store this information, *tool specific information* may be associated with each *object* and each *label*. The internal structure of the *tool specific information* depends on the tool and is not specified by PNML. PNML provides a mechanism for clearly marking *tool specific information along* with the name and the version of the tool adding this information. Therefore, other tools can easily ignore it, and adding *tool specific information* will never compromise a *Petri Net Document*.

The same *object* may be tagged with *tool specific information* from different tools. This way, the same document can be used and changed by different tools at the same time. The intention is that a tool should never change or delete the information added by another tool as long as the corresponding *object* is not deleted. Moreover, *tool specific information* should be self-contained and not refer to other *objects* of the net because the deletion of other *objects* by a different tool might make this reference invalid and leave the *tool specific information* inconsistent. This use of the *tool specific information* is strongly recommended; however, it is not normative!

5.2.6 Data Types

This Clause defines six data types, which can be used when defining labels for some *Petri net type* with numerical and textual information. They correspond to different data types in XMLSchema. Figure 4 gives an overview on these data types and their relation.

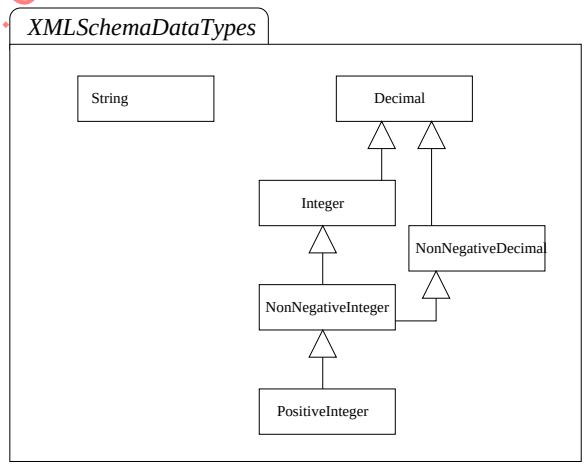

Figure 4: Standard data types

String refers to any printable sequence of characters, which will be mapped to XML PCDATA. *Decimals*, *NonNegativeDecimals*, *Integer*, *NonNegativeInteger*, and *PositiveInteger* refer to any character sequence which denotes a number of the corresponding kind.

5.3 Petri Net Type Meta Models and their Built-in Sorts

This Clause defines meta models for three versions of *Petri nets*: *Place/Transition Net*, *Symmetric Nets*, and *High-Level Petri Net Graphs* (HLPNGs) as defined in part 1 of ISO/IEC 15909, where *Symmetric Nets* are a restricted version of *High-Level Petri Net Graphs* currently defined as an Amendment to part 1. This general structure has already been shown in Fig. 1. These meta models define the labels of the respective *Petri net type*.

Annex B.1 of part 1 of ISO/IEC 15909 defines a mapping from the concepts of *Place/Transition Nets* to the concepts of *High-Level Petri Net Graphs* and shows how the usual mathematical representation of *Place/Transition Nets* can be represented as a restricted version of *High-level Petri Nets*. This will be *Place/Transiton Nets in High-level Notation*, which will be defined in Clause 5.3.12. This part of ISO/IEC 15909 introduces an explicit transfer format for *Place/Transition Nets* in order not to force tools for *Place/Transition Nets* to use the syntax of *High-Level Petri Net Graphs*. This format reflects the usual mathematical definition of *Place/Transition Nets*.

5.3.1 Place/Transition Nets

This Clause defines the meta model for *Place/Transition Nets* in terms of a UML package *PT-Net*.

A *Place/Transition Net* is a *net graph*, where each *place* can be labelled with a natural number representing the *initial marking* and an *arc* can be labelled with a non-zero natural number representing the *arc annotation*, with the meaning as defined in part 1 of ISO/IEC 15909 (see Clause B.1): the *label* of a *place* p denotes the initial marking $M_0(p)$, the *label* of an *arc* f denotes the arc weight $W(f)$.

Figure 5 shows the package PT-Net. Note that the only classes defined here are *PTMarking* and *PTArcAnnotation*. The classes *Place*, *Arc*, and *Annotation* come from the package *PNML Core Model*. They are imported (actually, they are merged) here in order to define the possible *labels* for the particular *nodes* of *Place/Transition Nets*. Likewise, the classes prefixed with *XMLSchemaDataTypes* are imported from the standard data type package (see Clause 5.2.6).

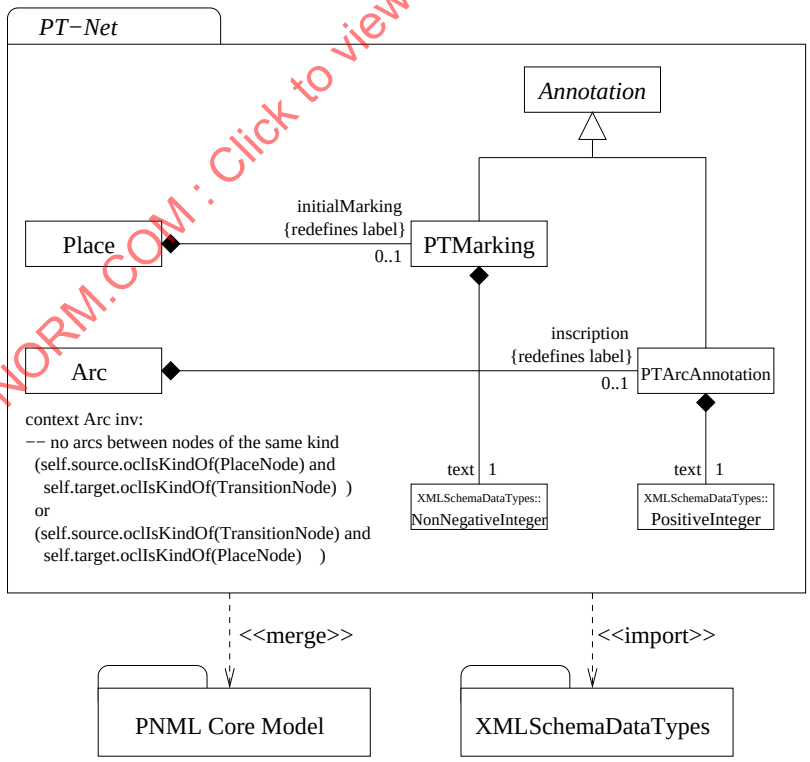

Figure 5: The package PT-Net

The *initial marking* of a *place* is represented by the *annotation PTMarking*, the contents of which must be a non-negative integer. Technically, the representation of the contents of this *label* is defined by referring to the data type *NonNegativeInteger*.

The *arc annotation* is represented by the *annotation PTArcAnnotation*, the contents of which must be a non-zero natural number, which is defined by referring to the data type *PositiveInteger*.

Note that, according to this definition, it is legal that a *place* does not have an *annotation* for the *initial marking*. In that case, the *initial marking* is assumed to be empty, i. e. 0. Likewise, there may be no *annotation* for an *arc*. In that case, the *arc annotation* is assumed to be 1.

In addition to the definition of the new *labels*, this package also defines the structural restriction that, in a *Place/-Transition Net*, an *arc* must not connect a *place* to a *place* or a *transition* to a *transition*. This is captured by the OCL expression below class *arc*.

Note that the UML meta model of this package does not define the XML syntax for the representation of these labels. The mapping to XML syntax is defined in Clauses 7.1.4 (data types) and 7.2 (labels for *P/T nets*).

Sometimes, one wants to store the position of the individual tokens within a place. In order not to mandate all tools to support this feature, this part of ISO/IEC 15909 suggests using tool specific information for this purpose, which would refer to the tool *org.pnml.tool*. Since no tool must support tool specific features, every tool is free to use and support this feature or not.

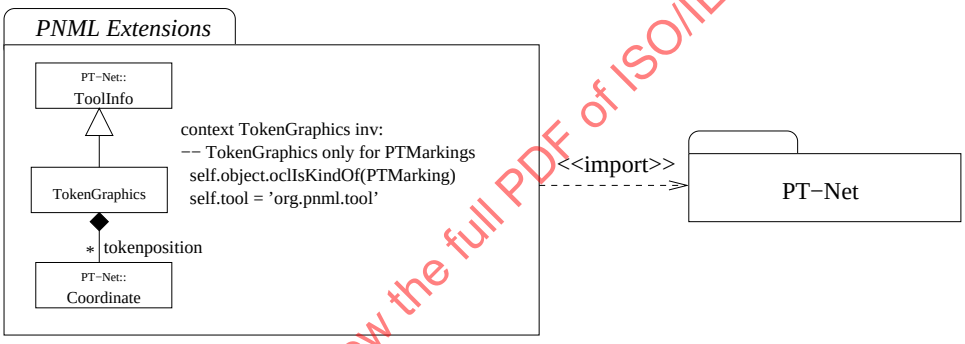

Figure 6: Tool specific extension for token positions

This extension is shown in Fig. 6. For a *PTMarking*, this label can be equipped with the *tool specific* information attached to *AnnotationGraphics*. This consists of information on a list of *tokenpositions*. Each of these elements represents the position of a token relative to the centre of the *place*; represented as a *Coordinate*, which is imported from package *PT-Net*, where it was obtained by a merge with the *PNML Core Model*. If this graphical information is present at all, the number of *tokenpositions* should be the same as indicated by the *PTMarking*.

5.3.2 High-Level Core Structure

Part 1 of ISO/IEC 15909 defines *High-level Petri Net Graphs*. *Symmetric Nets* and *Place/Transition Nets* are introduced as restricted versions of *High-level Petri Net Graphs*. The difference is in the *types* and *functions* that may be used in the restricted versions.

This Clause defines the general structure of *Symmetric Nets*, *High-Level Petri Net Graphs*, and *Place/Transition Nets in High-level Notation*. This is called the *High-Level Core Structure*. In Clauses 5.3.10, 5.3.11, and 5.3.12, this *High-Level Core Structure* will be used for defining the three Petri net types.

Syntactically, the basic features of a *High-Level Petri Net* are the *annotations* of *places*, *transitions*, and *arcs*. For each *place*, the *sort* defines the *type* of the tokens on this *place*. Note that, in contrast to part 1, the place is indirectly associated with a *type* via the *sort* and its fixed interpretation. The *term* associated with a *place* denotes the *initial marking* and must have the respective *sort*. The *term* associated with an *arc* to or from a *place*, defines which tokens are added or removed, when the corresponding *transition* fires. These *terms* also must be of the respective *sort*. Note that part 1 is a bit more general since it allows, any expression that always evaluate to elements of the correct type, which is a semantical condition. Here, we use *terms*, which enforce the correct typing syntactically.

For constructing such *terms*, one can use built-in *operators* and *sorts*, and user-defined *variables*, which are defined in a *variable declaration*. The *variable declarations* are *annotations* of the *net* or a *page*. Moreover, a *transition* can have a *condition*, which is a *term* of *sort* Bool and imposes additional conditions on the situations in which a transition can fire. These concepts and their semantics are precisely defined in part 1 of ISO/IEC 15909. The only difference is that, in part 2, there is a fixed relation between *sorts* and *types* and between *operators* and *functions*.

The UML diagram for the package *Terms*, which defines all these concepts, is shown in Fig. 7: It defines the concepts of *sorts*, *operators*, *declarations*, and *terms*, and how *terms* are constructed from *variables* and *operators*.

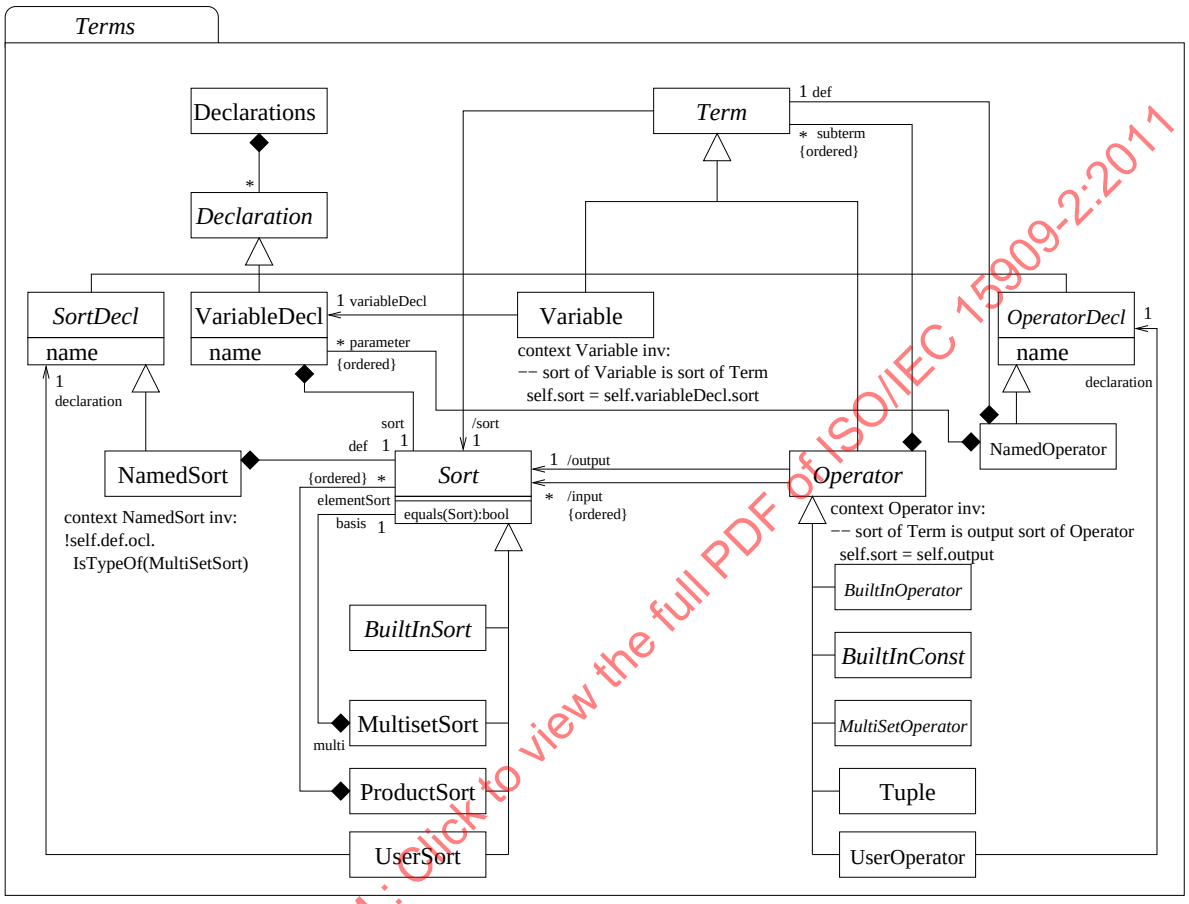

Figure 7: The meta model for *Terms*

For each *variable declaration* there is a corresponding *sort*. A *sort* can be a *built-in sort*, a *multiset sort* over some *basis sort*, a *product sort* over some *sorts*, or a *sort* which is given in a *user declaration*. In the core structure, the only possible *sort declaration* of a user is by constructing a new *sort* from existing ones and by giving them a new name. From these, the user can define new ones. Note that cyclic references in user defined *sorts* are not allowed. In addition to these user defined sorts, called *named sorts*, there are *arbitrary sorts*. Since these are not allowed in *Symmetric Nets* these are defined only in Clause 5.3.11.

An *operator* can be a *built-in constant* or *built-in operator*, a *multiset operator* which among others can construct a *multiset* from an enumeration of its elements, or a *tuple* operator. Each *operator* has a sequence of *sorts* as its *input sorts*, and exactly one *output sort*, which defines its signature. As for *sorts*, the user can define his own *operators*. Here, it is only possible to define an abbreviation, which is called a *named operator*: It can use a *term*, which is built from existing *operators* and variables, for defining a new *operator*. As for *sorts*, cycles (recursion) in these definitions are not allowed. As for sorts, there will be arbitrary operator declarations for *High-level Petri Net Graphs*, but not for *Symmetric Nets*. Therefore, these concepts will be defined in Clause 5.3.11.

From the built-in *operators* and the user-defined *operators* and *variables*, *terms* can be constructed in the usual way. The *sort* of a *term* is the *sort* of the *variable* or the *output sort* of the *operator*. Therefore, *sort* is indicated as a derived association; its definition is expressed by an OCL expression in the UML meta model. Note that the *input* and

output sort of the *operator* are also represented as derived associations because, in some situations, they need not be given explicitly since they can be derived from the type of the *operator*.

Figure 8 shows the package *High-Level Core Structure*, which defines all the *annotations* for both *Symmetric Nets* and *High-Level Petri Net Graphs*. Note that, since the classes for built-in *sorts* and *operators* are abstract, we do not have any built-in *sorts* and *operators* yet. These will be defined in Clauses 5.3.3 to 5.3.9.

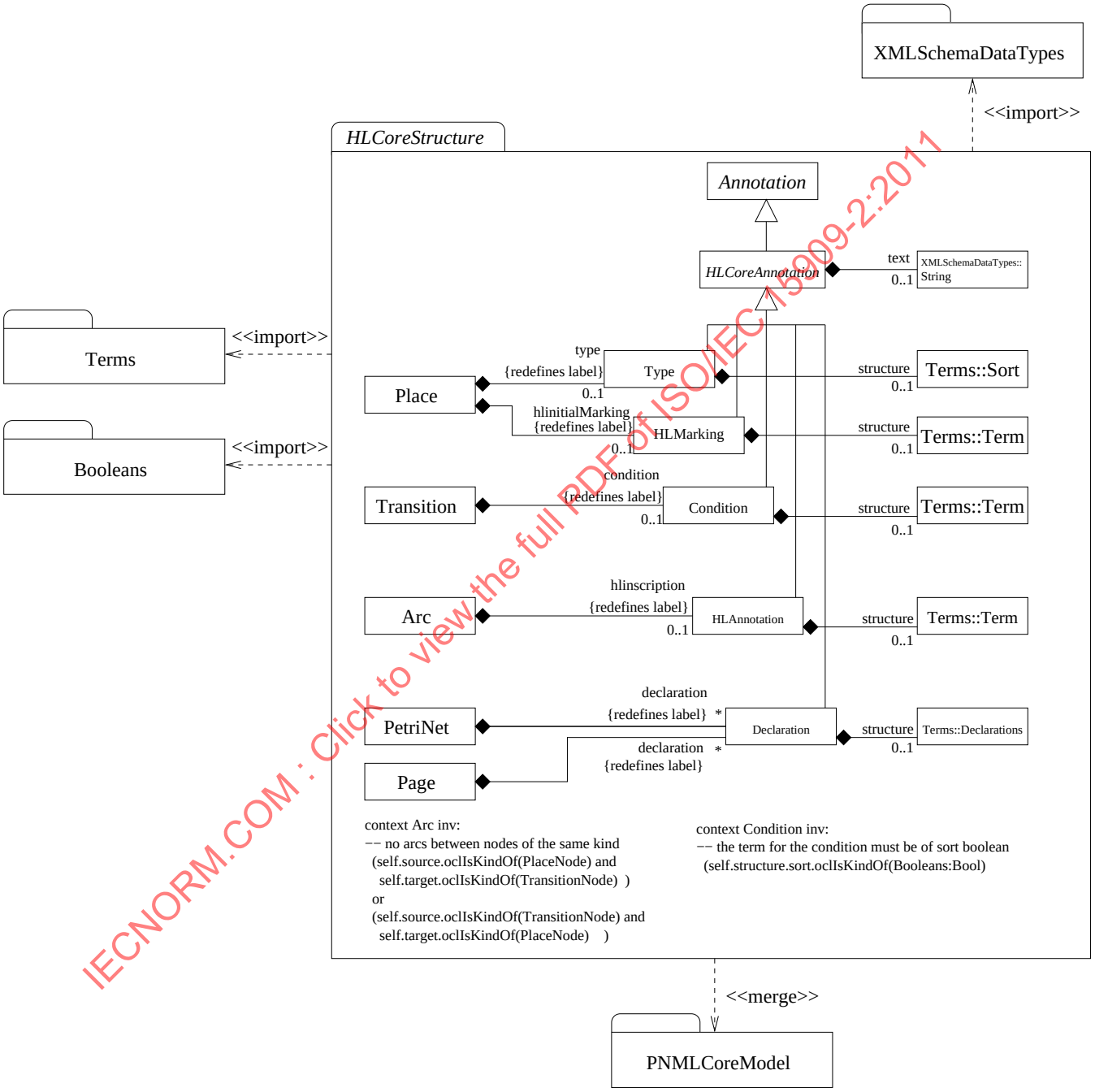

Figure 8: The package *High-Level Core Structure*

In addition to the *annotations* defined above, the package *High-Level Core Structure* requires that *arcs* must not connect two *places* and must not connect two *transitions*.

Note that this model defines an abstract syntax (*structure*) for all these concepts only. In order to allow tools to store some concrete text, all *annotations* of *High-level Nets* may also consist of text, which should be the same

expression in some concrete syntax of some tool. The concrete syntax, however, is not defined in this part of ISO/IEC 15909. This common structure of all *annotations* is captured by *HLCoreAnnotation* in the package *High-Level Core Structure*.

The concepts defined in Fig. 8 correspond to the concepts defined in part 1 of ISO/IEC 15909 in the following way. The *label Type* of a *place* defines the *type* by referring to some *sort*; by the fixed interpretation of built-in *sorts*, this *sort* defines the *type* of the *place*. The *label HLMarking* of a *place* is a *term* with some *multiset sort* denoting a collection of tokens on the corresponding *place*, which defines its initial marking. The *label Condition* is a *term* of *sort Bool*, which is the *transition condition*. The *label HLAnnotation* refers to a *term*, which is the expression defining the *arc annotation*.

Declaration labels correspond to the *declarations*, which can be *variable declarations* and declarations of *sorts* and *operators*. In the *High-level Core Structure* the user can define abbreviations only; later, in *High-Level Petri Net Graphs*, there will be arbitrary declarations.

As discussed above, this Clause defines the core structure of *High-level Petri Net Graphs*. The following Clauses 5.3.3 to 5.3.9 define different built-in *sorts* and *operators* along with their interpretations as *types* and *functions* respectively. For each built-in *sort*, there will be a UML package.

5.3.3 Dots

To enable the use of *Place/Transition Net* annotations as high-level annotations, the concept of Dots has been defined. *Place/Transition Nets* as High-level Net Graphs are defined in Clause 5.3.12.

Dots represent the *common understanding* of *non coloured annotation* in a High-level net. That is to say, dots are the high-level representation of the natural numbers labels used in Place/Transition Nets. The difference is that they are typed as a built-in sort to be compatible with the corresponding high-level annotation defined in part 1 of ISO/IEC 15909. Dots are depicted in Fig. 9.

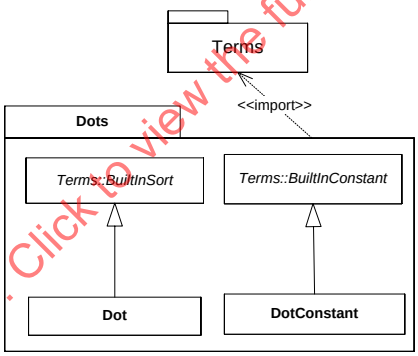

Figure 9: The package for Dots

They consist of a *sort* named Dot. The *type* associated with this *sort* is the set with a single element: $\{\bullet\}$. The constant $\mathrm{DotConstant}$ is associated with the 0-ary function that returns $\bullet$, i. e. the constant denotes that single element.

This *sort* can be used for representing *non coloured annotations* in *High-level Petri Nets*.

5.3.4 Multisets

Figure 10 shows the UML package for *multisets* and their *operators*. For every *sort*, the *multiset sort* over this *basis sort* is interpreted as the set of multisets over the type associated with the *basis sort*.

Below we define the *operators* on *multiset sorts* along with their interpretation:

- $\mathrm{Add} : \mathrm{Multiset} \times \mathrm{Multiset} \times \ldots \times \mathrm{Multiset} \rightarrow \mathrm{Multiset}$ is interpreted as the multiset addition. For example, $\mathrm{Add}([a, b], [b, c])$ evaluates to $[a, 2b, c]$.

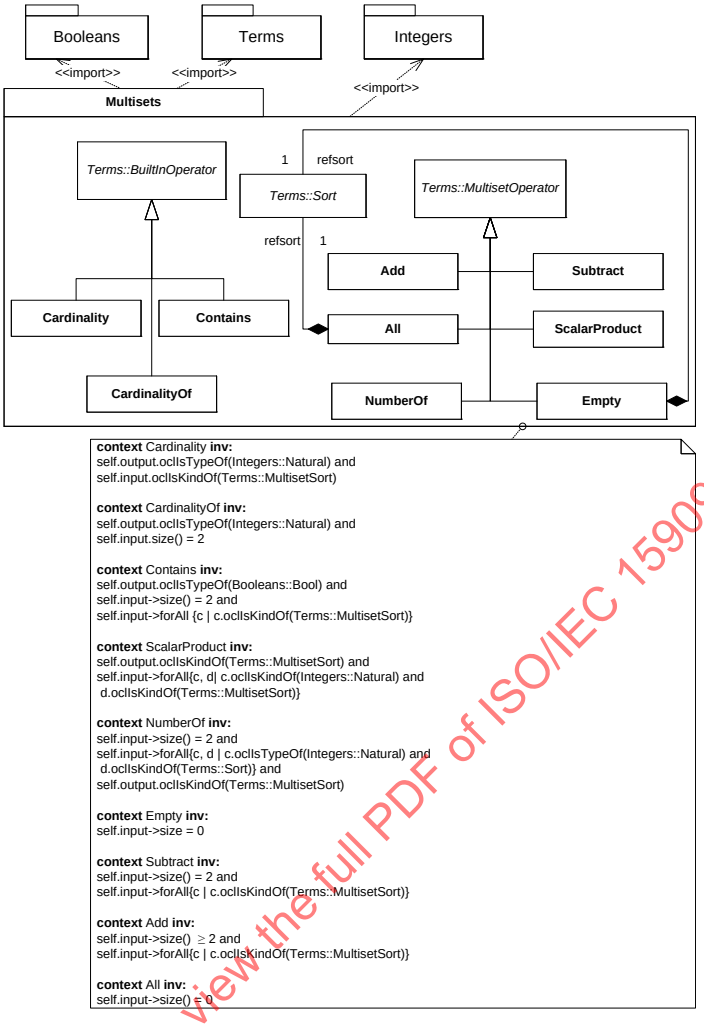

Figure 10: The package *multiset* and its built-in and *operators*

- $\mathrm{All} :\rightarrow \mathrm{Multiset}$ is a *constant* that, for a given finite *sort*, is interpreted as the multiset that contains every element of the type associated with that sort exactly once. For example, for a *sort* s with associated type $\{a, b, c\}$ the *constant* All_s denotes the multiset $[a, b, c]$. Note that the *sort* which is indicated by index s in this example is represented by the composition *refsort* in the meta model since the *output sort* cannot be derived from the *operator* or its *subterms*.
- $\mathrm{Empty} :\rightarrow \mathrm{Multiset}$ is a *constant* which constructs the empty multiset over the associated type of the *sort*, i. e. Empty_s denotes $[]$. Note that the *sort* which is indicated by the index s in this example is represented by the composition *refsort* in the meta model since the *output sort* cannot be derived from the *operator* or its *subterms*.
- $\mathrm{Subtract} : \mathrm{Multiset} \times \mathrm{Multiset} \rightarrow \mathrm{Multiset}$ is interpreted as the multiset subtraction. For example, the term $\mathrm{Subtract}([a, 2b, c], [b, c])$ evaluates to $[a, b]$.
- $\mathrm{ScalarProduct} : \mathbb{N} \times \mathrm{Multiset} \rightarrow \mathrm{Multiset}$ is interpreted as the product of a multiset with a scalar. For example, $\mathrm{ScalarProduct}(2, [a, c])$ evaluates to $[2a, 2c]$.
- $\mathrm{NumberOf} : \mathbb{N} \times \mathrm{Sort} \rightarrow \mathrm{Multiset}$ is interpreted as a function that produces a multiset in which the element occurs exactly in the given number and no other elements in it. For example, $\mathrm{NumberOf}(3, a)$ evaluates to $[3a]$.
- $\mathrm{Cardinality} : \mathrm{Multiset} \rightarrow \mathbb{N}$ is interpreted as the cardinality function, i. e. it returns the number of elements of a multiset. For example, $\mathrm{Cardinality}([a, 2b, c])$ evaluates to 4.

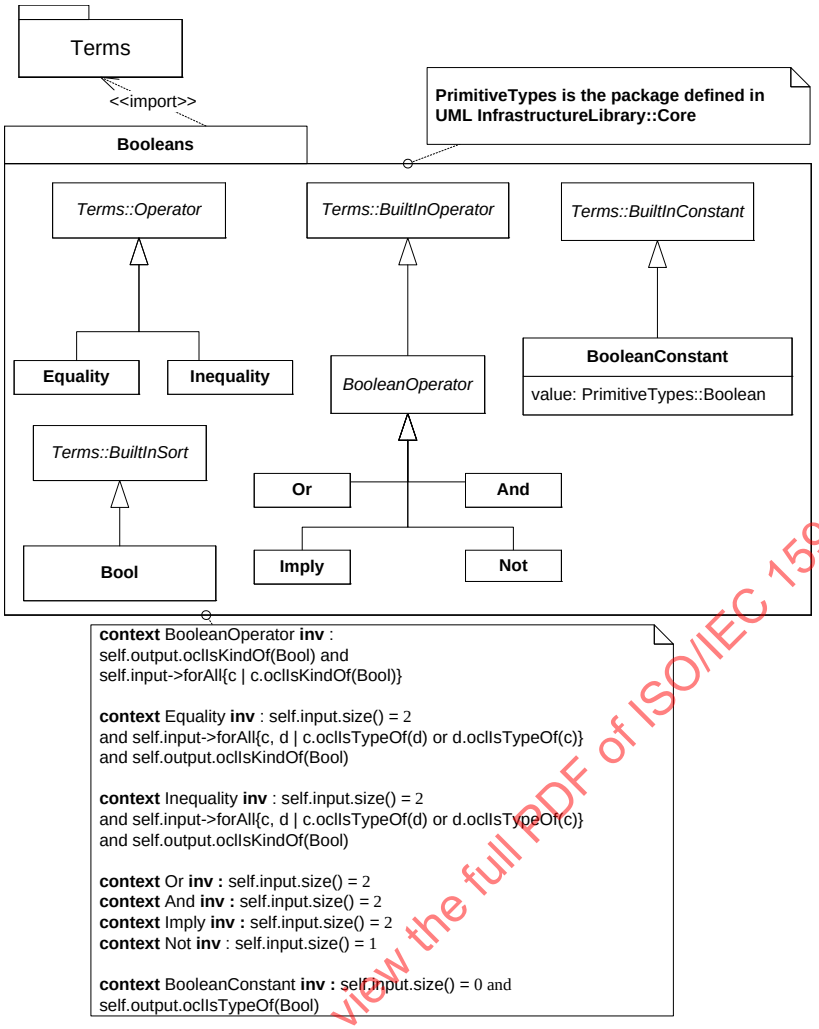

Figure 11: The package *Booleans*

- $\mathrm{CardinalityOf} : \mathrm{Multiset} \times \mathrm{Sort} \to \mathbb{N}$ is interpreted as the function that returns the number of times the given element occurs in the multiset. For example, the term $\mathrm{CardinalityOf}([a, 2b, c], a)$ evaluates to 1 and the term $\mathrm{CardinalityOf}([a, 2b, c], b)$ evaluates to 2.

- $\mathrm{Contains} : \mathrm{Multiset} \times \mathrm{Multiset} \to \mathrm{Bool}$ is interpreted as multiset-inclusion function. For example, let $m_1 = [a, b, 2c]$ and $m_2 = [2a, 2b, 2c]$ be two multisets over type $\{a, b, c\}$. Then, $\mathrm{Contains}(m_1, m_2)$ evaluates to false and $\mathrm{Contains}(m_2, m_1)$ evaluates to true.

Multiset literals on typical annotations such as an arc's *HLAnnotation* are defined using Add and $\mathrm{NumberOf}$.

5.3.5 Booleans

Figure 11 shows the UML package for *sort Bool* and its *operators*.

The boolean operators Not, And, Or, Imply operate on boolean arguments as stated by the OCL constraint on *BooleanOperator*. They are interpreted as the classical logical functions.

The $\mathrm{Equality}$ and $\mathrm{Inequality}$ operators take arguments of the same *sort* (not only booleans) and return a *Bool*. They are interpreted as the equality function and the inequality function. Any other built-in sort package may import and use these two operators.

The BooleanConstant is a built-in constant for representing the two values of its type.

5.3.6 Finite Enumerations

Figure 12 shows the UML package for *finite enumerations* and their *operators*.

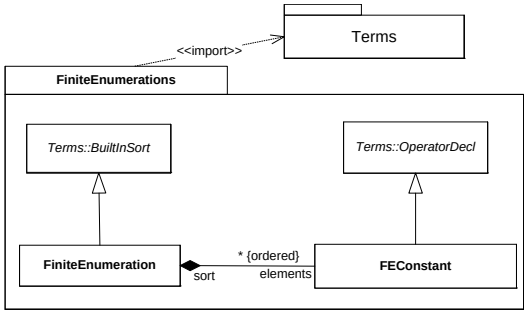

Figure 12: The package *Finite Enumerations* and its built-in *sorts* and *operations*

A *FiniteEnumeration* is defined by explicitly naming all its elements, which are called *FEConstant* in the meta model and the attribute *name* (inherited from *OperatorDecl*) represents the actual name of the respective element. On the one hand, these *FEConstants* are part of the *declaration* of the *FiniteEnumeration sort*. On the other hand, each of these *FEConstants* defines a 0-ary *operation*, i. e. is a *declaration* of a *constant*.

The *type* associated with the declaration of a *FiniteEnumeration* is the set of these constants. The operations on a *FiniteEnumeration* are the use of these *FEConstant* declarations. The *function* that is associated with this *FEConstant* is the 0-ary *function* that returns the respective element of the type.

5.3.7 Cyclic Enumerations

Figure 13 shows the UML package for *cyclic enumerations* and their *operators*.

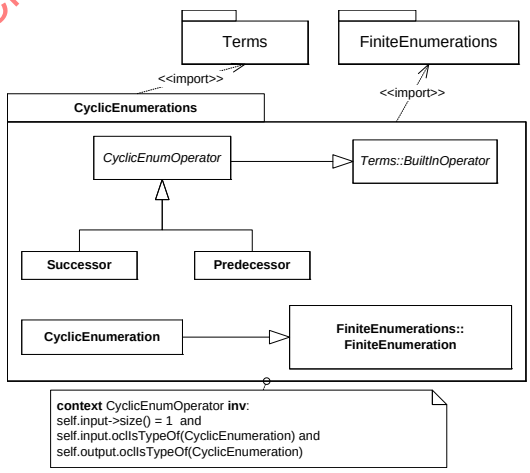

Figure 13: The package *Cyclic Enumerations* and its built-in *sorts* and *operators*

The declaration of a *CyclicEnumeration* has same structure as the *FiniteEnumeration*. This is why *CyclicEnumeration* is derived from *FiniteEnumeration* in the meta model. The only difference are two additional built-in *operations Successor* and *Predecessor*.

The order of the elements on the associated *type* is the order in which its elements are defined in the declaration. And the *functions* associated with the *operations Successor* and the *Predecessor* are the successor resp. predecessor function with respect to this order, where the order wraps at the first and last element.

5.3.8 Finite Integer Ranges

Figure 14 shows the UML package for *finite integer ranges* and their *operators*.

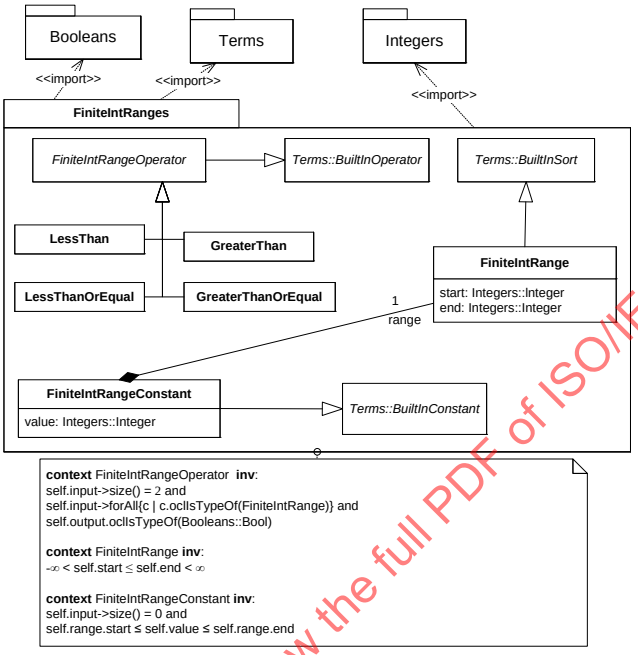

Figure 14: The package *Finite Integer Ranges* and its built-in *sorts* and *operators*

An integer range is defined by refering to *FiniteIntRange*, where the attributes *start* and *end* define the first and the last element of the range. The *type* associated with that *sort* is the set of all integers between (and including) the first and the last element.

The four built-in operations on that type are *LessThan*, *LessThanOrEqual*, *GreaterThan*, and *GreaterThanOrEqual*. The functions associated with these operations are the resp. comparisons with respect to integers.

In addition there are the *constants FiniteIntegerRangeConstant* which are interpreted as a 0-ary function refering to the respective value of the integer range, which is given by the attribute *value* and must refer to a value between the start and end of the respective range. In the package, this is expressed by an OCL constraint.

Note that there is no need to explicitly declare a $\mathrm{FiniteIntRangeConstant}$ in the declarative part of a Petri Net. It can be directly used in the body of the Petri Net and must therefore only refer to the declared range it belongs to.

5.3.9 Partitions

Figure 15 shows the UML package for *Partitions* and their *operators*.

The purpose of a *partition* is to partition the elements of a *type* into equivalence classes. Technically, a $\mathrm{Partition}$ is a *declaration* of a new *sort* refering to the partitioned *sort* and naming all its partitions, each of which is called $\mathrm{PartitionElement}$. Each $\mathrm{PartitionElement}$ is also an *operator declaration*. For each $\mathrm{PartitionElement}$, the declaration lists all the elements that belong to that partition, which are represented as a Term of the partitioned *sort*. The newly declared *partition sort* is then interpreted as the set of all its *partition elements*.

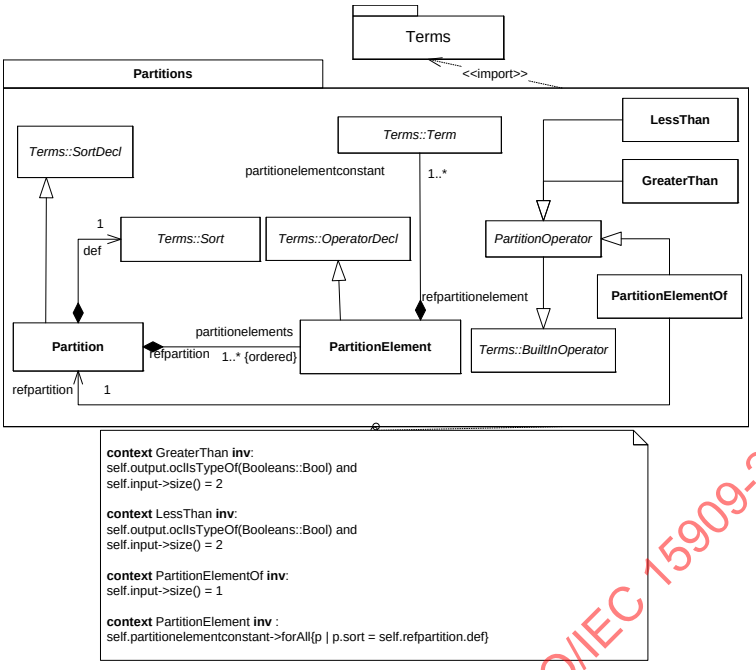

Figure 15: The package *Partitions* and its built-in *sorts* and *operators*

The operations that are defined on a *sort* Partition are the LessThan and GreaterThan. These are interpreted as comparison functions on the partition elements which is based on the order in which they occur in the declaration of the partition.

Moreover, there is an *operator* PartitionElementOf from the partitioned sort to the partition. This operator is interpreted as the function that maps each element of the type associated with the partitioned sort to the partition element it belongs to as defined in the declaration of the partition. Note that in some cases the *partition* to which the *operator* PartitionElementOf refers to cannot be derived from the subterms. Therefore, this partition is explicitly represented by a reference from this *operator* to the *partition*.

5.3.10 Symmetric Nets

Symmetric Nets as currently defined in an Amendment of part 1 of ISO/IEC 15909 (creation of Annex B.2), are *High-Level Petri Net Graphs* with some restrictions. Basically, the carrier sets of all basic *sorts* are finite, and only a fixed set of *operations*, as defined below, are allowed. Therefore, the *sort* of each *place* must not be a *multiset sort*. The available built-in *sorts* are defined below.

Moreover, for *Symmetric Nets*, it is required that every HLPNG *annotation* has the *structural* information.

The built-in *sorts* of *Symmetric Nets* are the following (as defined in the previous Clauses): *Bool*, *range of integers*, *finite enumerations*, *cyclic enumerations* and *dots*. Moreover, for every *sort*, there is the operator *all*, which evaluates to a multiset that contains exactly one occurrence of each of the elements of the type associated with the basis *sort*. This is often called the broadcast function.

Altogether, the *Symmetric Nets* can be defined by importing the packages as shown in Fig. 16.

5.3.11 High-Level Petri Net Graphs

This Clause defines the meta model for *High-Level Petri Net Graphs* (HLPNGs). This, basically, means adding more built-in *sorts* and *operators*; moreover, user *declarations* of arbitrary *sorts* and arbitrary *operators* are introduced.

This clause defines the built-in *sorts* and *operators* of HLPNGs.

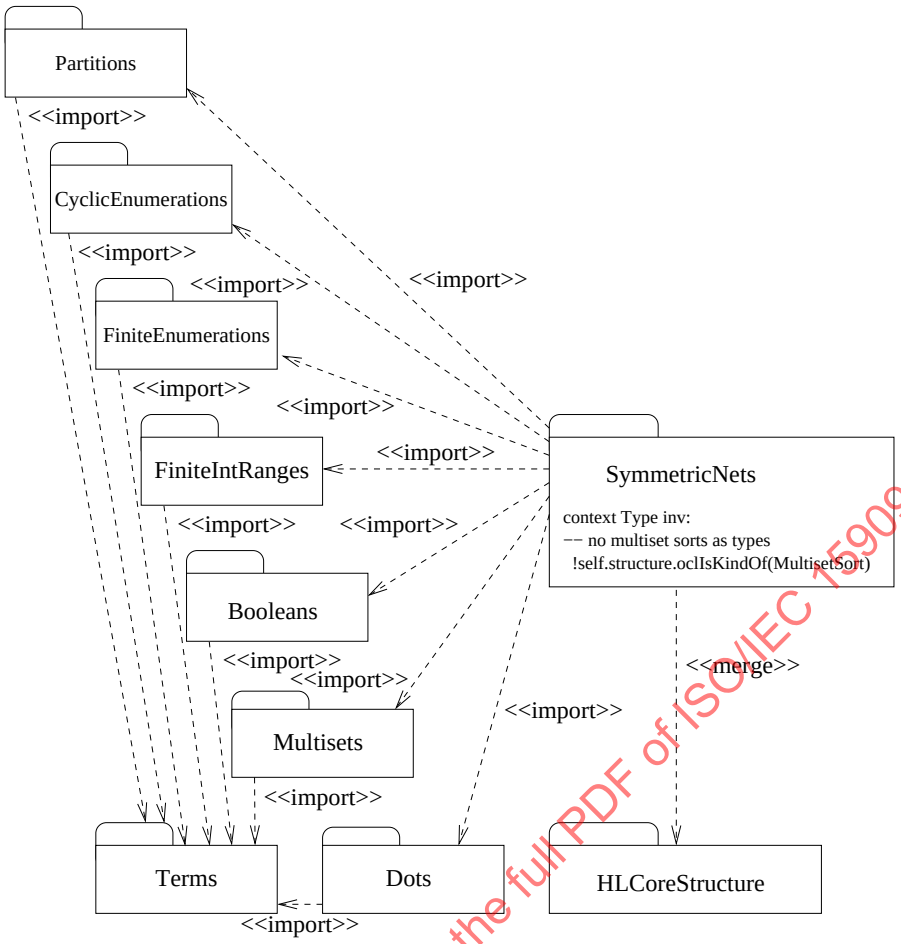

Figure 16: The package *Symmetric Nets* and its built-in *sorts* and *functions*

In addition to the built-in *sorts* and *operators* of *Symmetric Nets*, *High-Level Petri Net Graphs* have the following *sorts* and *operators*:

- Sort *integer* (corresponding to the set of all positive and negative numbers) with *addition*, *subtraction*, *multiplication*, integer *division*, and *modulo* operators as the functions as well as the comparison operators $\leq$, $<$, $>$, and $\geq$. Moreover, there is a constant for any integer number.

  This part of ISO/IEC 15909 does not mandate the exact implementation of integers; it could be finite or infinite.

- The sort of *strings* with the *concatenation* and comparison operators $\leq$, $<$, $>$, and $\geq$ as functions, where the order refers to the lexicographic order.

- For each *sort* there is a *sort* List over that *sort*. Moreover, there is a constant for the *empty list*, an operator for *appending* a single element of the type corresponding to the sort to the list, and an operator for *concatenating* two lists.

Figure 17 shows the UML package *Integers* with the defined *sorts* and *operators*.

Three sets of numbers are defined: Natural, Positive and Integer. Any number specified as a constant can be declared as an instance of NumberConstant. It refers to one of the sets and its value must be an element of the set it refers to. OCL constraints have been used to enforce this statement. Note that it is not necessary to declare these constants; number constants can be used wherever necessary without an explicit declaration.

Classical arithmetic operators are defined such as Addition, Subtraction, etc. and comparison operators such as LessThan, GreaterThan, etc.

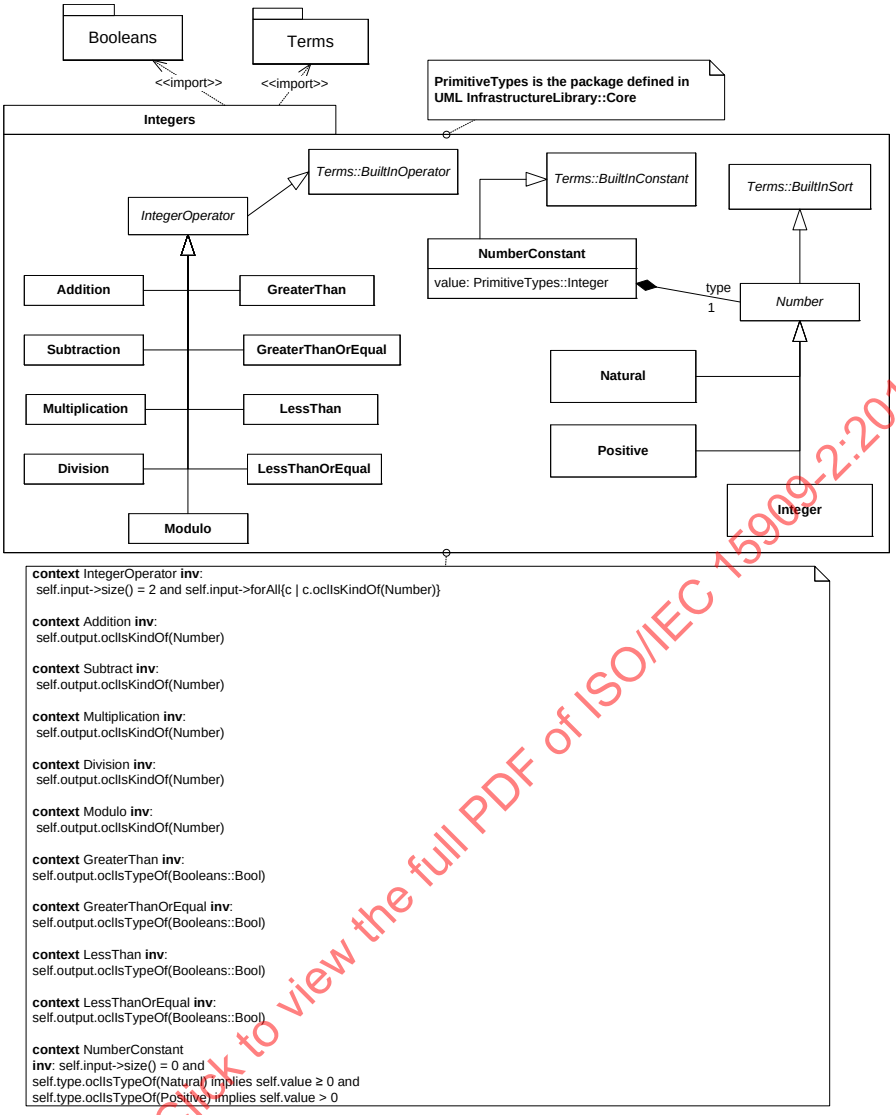

Figure 17: The package *Integers* and its built-in *sorts* and *operators*

Figure 18 shows the UML package *Strings* with the defined *sort* and its *operators*.

Any constant string is defined as an instance of StringConstant, and can be directly used in the body of a Petri Net.

Strings can be concatenated, resulting in a new string. The relational operators compare two strings according to the *lexicographic order*. Their output is thus a boolean. The operator Length of a string returns a natural. Substrings can be extracted from existing strings, using the Substring operator.

Figure 19 shows the UML package *Lists* with the defined *sort* and its *operators*.

A list is constructed over a pre-declared sort using the List class, or over one or more instances of declared sort elements, using MakeList, by enumerating all its elements. Note that in some cases the basis *sort* in the MakeList *operator* cannot be derived from its *subterms*; therefore, this is expliclity represented by a composition in the meta model.

The EmptyList is defined as a built-in constant. As a function, its arity is thus zero. Note that the basis *sort* of the EmptyList *operator* cannot be derived from its *subterms*; therefore, this is expliclity represented by a composition in the meta model.

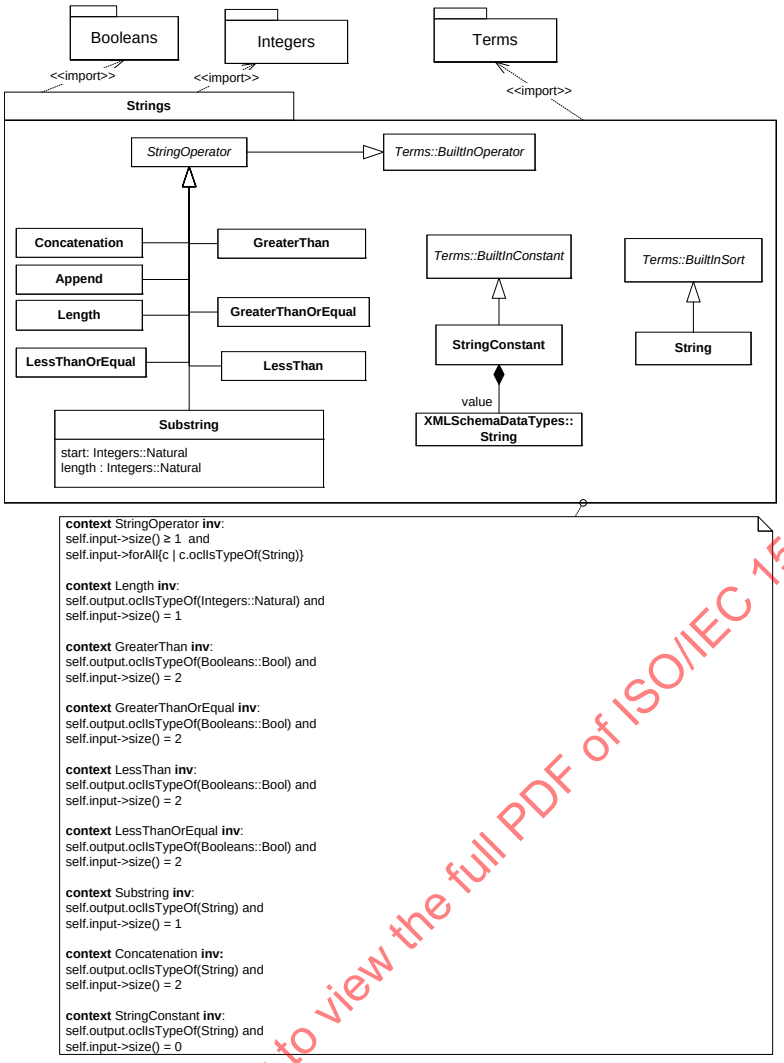

Figure 18: The package *String* and its built-in *sorts* and *operators*

Lists can be concatenated, and the length of a list can be defined. Sublists can be extracted from existing lists, using the Sublist definition. MemberAtIndex returns the element of a List at a given index.

Moreover, HLPNGs allow the user to define arbitrary *sorts* and *operators*. This package *ArbitraryDeclarations* is shown in Fig. 20. In contrast to *named sorts* and *named operators*, arbitrary *sorts* and *operators* do not come with a definition of the *sort* or *operation*; they just introduce a new symbol without giving a definition for it. So, these symbols do not have a meaning, but can be used for constructing *terms*.

The additional concept *Unparsed* in *terms* provides a means to include any text, which will not be parsed and interpreted by the tools. This might be helpful for exchanging the general structure of a term, but not all its details.

The complete definition for *High-Level Petri Net Graphs* is shown in Fig. 21. It extends *Symmetric Nets* by declarations for *sorts* and *functions* and the additional built-in *sorts* for *Integer*, *String*, and *List*.

5.3.12 Place/Transition Nets as High-level Net Graphs

Annex B of part 1 of ISO/IEC 15909 defines *Place/Transition Nets* as a restricted form of *High-level Net Graphs*. Fig. 22 shows the corresponding UML definition. This class allows for the use of the built-in *sorts Bool* and *Dot* only. It does not allow any user *declarations*, neither *variables*, nor *sorts*, nor *operators*.

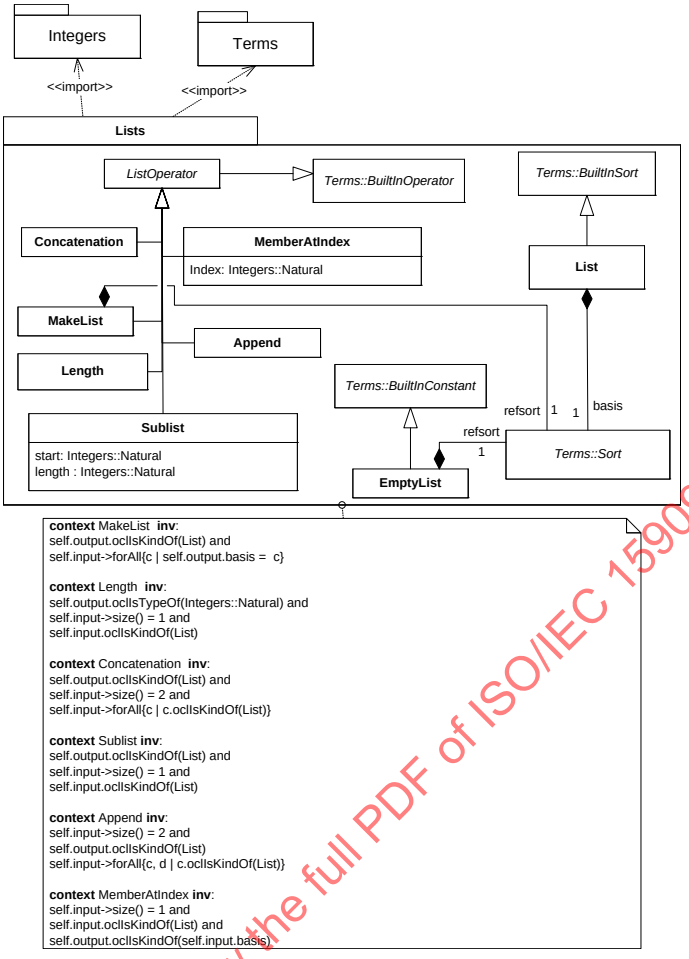

Figure 19: The package *List* and its built-in *sorts* and *operators*

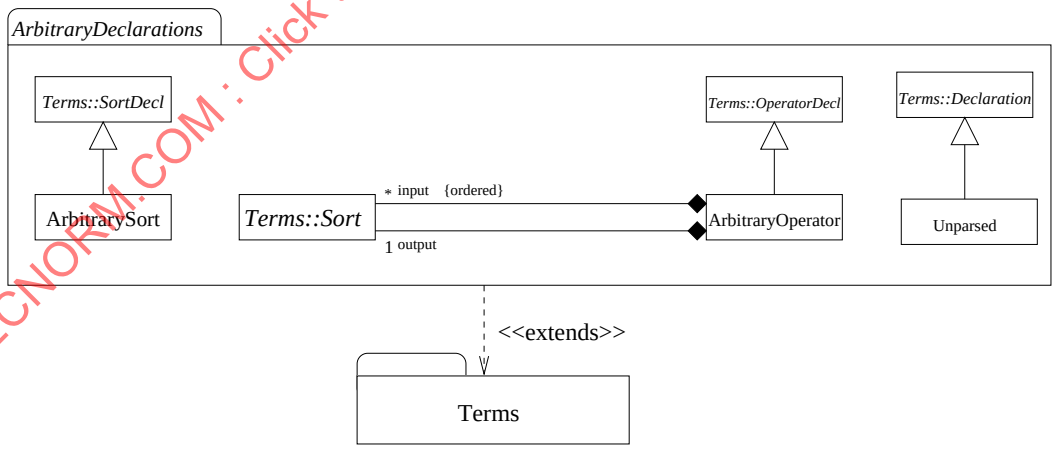

Figure 20: The package *ArbitraryDeclarations*

The *type* of each *place* must refer to *sort Dot*. All transition *conditions* need to be the constant *true*, if this label is present. And the *arc annotations* and the *initial markings* are *ground terms* of the *mulitset sort* over *Dot*.

Note that this *Place/Transition Nets in High-level Notation* is a restricted form of *High-level Net Graphs* and even of *Symmetric Nets*.

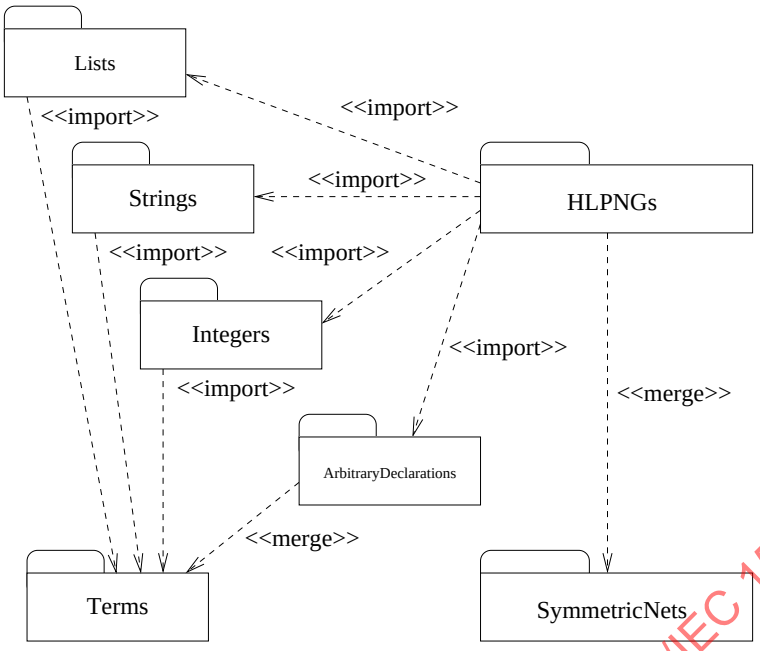

Figure 21: The package *HLPNGs*

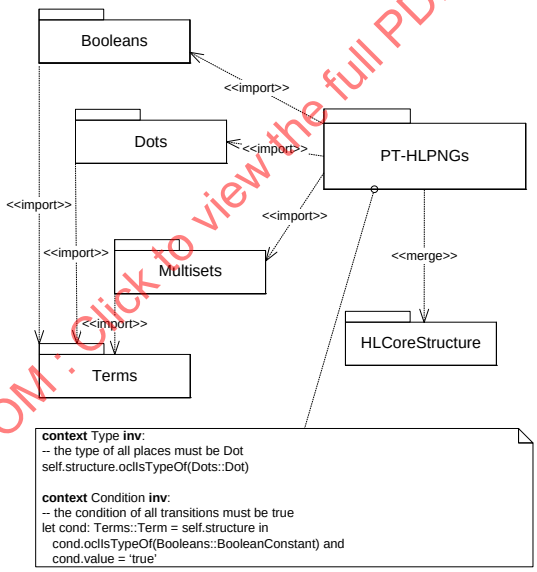

Figure 22: The package of *P/T Nets* defined as restricted *HLPNGs*

Annex B of part 1 of ISO/IEC 15909 defines a mapping from these *Place/Transition Nets in High-level Notation* to the usual mathematical definition of *Place/Transition Nets*. This way, there is a mapping from *Place/Transition Nets in High-level Notation* to the notation given in Clause 5.3.1.

6 Mapping between Part 1 and Part 2

This part of ISO/IEC 15909 defines a transfer format for *High-level Petri Net Graphs*. The concepts of *High-level Petri Net Graphs* have been defined in part 1 of ISO/IEC 15909. The focus of part 1, however, is on the semantical concepts. Therefore, some concepts are defined semantically only. For example, the *type* of a *place* can be any set. But, part 1 does not define any syntax for defining sets. Therefore, this part of ISO/IEC 15909 introduces some syntax for the concepts, which are defined semantically only in part 1. To this end, we use concepts which have been introduced in part 1 already, and which, in particular, have been used in the definition of *High-level Petri Net Schemas* (*HLPNS*) in the informative Annex C of part 1.

Moreover, this part of ISO/IEC 15909 defines some concepts that have not been defined in part 1 of ISO/IEC 15909. These are the concepts for defining the graphical appearance of *Petri Nets* (see Clause 5.2.4) and the concepts for structuring large *Petri Nets* (see Clause 5.2.2).

This Clause discusses the differences of part 1 of ISO/IEC 15909 and this part of ISO/IEC 15909 and maps the syntactical concepts of this part of ISO/IEC 15909 (actually the abstract syntax as defined in Clause 5) to the semantical concepts as defined in Clause 6.2 of part 1 of ISO/IEC 15909.

6.1 Graphics and Structuring

Part 1 of ISO/IEC 15909 defines general rules for the graphical appearance of *Petri Nets* and *High-level Petri Net Graphs* (see Clause 7 of part 1). These rules, basically, say that *places* are shown as ellipses, that *transitions* are shown as rectangles, and that *arcs* are shown as arrows. But there is no information on the size, position, and style of these shapes. Moreover, part 1 states that the other features of *High-level-Petri Net Graphs* are shown as annotations of the associated *object*.

This part of ISO/IEC 15909 allows for the provision of much more information on the graphical appearance of the different objects. This information is more detailed; e. g. concerning colour, line-width, line-style, and fonts, but the graphical result is compatible with the general rules of part 1.

In addition, this part of ISO/IEC 15909 introduces the concepts of *pages* and *reference nodes*. These are purely graphical and do not carry any meaning. By merging the *reference nodes* to the *node* they ultimately refer to, and by ignoring the *labels* of the *reference nodes* and the *page structure*, the additional concepts of this part of ISO/IEC 15909 map to the concepts as defined in part 1 of ISO/IEC 15909. This is called flattening of a *Petri Net* with *pages*.

6.2 Annotations of HLPNGs

In this Clause, it is shown how the concepts as defined in Clause 6.2 and Annex A of part 1 of ISO/IEC 15909 are represented by the concepts as defined in Clause 5.

The *net graph* $NG = (P, T; F)$ is defined by the *places*, *transitions*, and *arcs* of the *Petri Net Document*. As mentioned in Clause 6.1, *pages* and *reference nodes* can be eliminated by flattening.

The *signature* $SIG = (S, O)$ consists of two parts: all the used built-in *sorts* and *operators*; in addition, there are the *user defined sorts* and *operators*.

The S-indexed set of *variables* V is defined by all *declarations* of *variables*.

The many-sorted algebra H is not explicitly given, because there is no syntax for defining algebras. Instead, each built-in *sort* comes with a fixed interpretation, which is a set. These sets are the carrier sets of the algebra. The built-in *operators* have a fixed interpretation, which define the *functions* of the algebra.

Note that, for the *declaration* of *arbitrary sorts* and *operators*, there is no associated meaning. This corresponds to the concepts of *High-level Petri Net Schemas* that do not have an associated algebra (see Annex C of part 1).

The function $Type$ that assigns a *type* to each *place* is defined by the label *type* of the *place*. Since there is no syntax for denoting a type, this label refers to a *sort*. By the fixed interpretation of *sorts*, this implicitly refers to a set, which is the *type* of that *place*.

The *arc annotation* A is defined by the *HLAnnotation* of *arcs*, which refers to a *term*. The only difference to part 1 is that, in part 2 of ISO/IEC 15909, it is syntactically guaranteed that the *term* always evaluates to the correct *type*.

The *annotation* TC is defined by the annotation *Condition* of *transitions*, which needs to be a *term* of *sort Bool*.

The *initial marking function* M_0 is defined by the label *HLMarking* of the *places*. Note that, in this part of ISO/IEC 15909, this is a *ground term* of the corresponding multiset sort. The evaluation of this *term* defines the *initial marking* of the place. This is necessary in order to syntactically express the *initial marking*.

Table 2 provides a summary of the relationships between the concepts of *High-level Petri Net Graphs* as defined in part 1 and part 2 of ISO/IEC 15909.

Table 2: *Mapping between definitions of Part 1 and Part 2*

Part 1	**Part 2**
A *place s type* refers to a set	In package *HLCoreStructure*, the *place type* refers to a *sort*. For each *sort* a corresponding set is defined, except when it is an arbitrary sort. The *annotation* is *Type*.
place marking is a collection of elements where repetition is allowed.	*Term* (*HLCoreStructure::HLMarking*) denoting a *multiset*.
Arc annotation is an *expression* of function and variables.	*Term* (*HLCoreStructure::HLAnnotation*) denoting the expression. The *term* should have the *multiset sort* over the *sort* of the *place*.
Transition condition denoting a boolean expression.	*Term* (*HLCoreStructure::Condition*) of *sort Bool*.
Declarations.	*Declaration* concept (abstract) is attached to *HLCoreStructure::Net* or *HLCoreStructure::Page*.
User *declaration* of *place type*.	Appears as *Terms::SortDecl*.
Typing of *variable*.	Appears as relationship between *Terms::Variable*, *Terms::VariableDecl*, and *Terms::Sort*.
Operator declaration.	Appears as *HLCoreStructure::OperatorDecl*.
Annex A.1 defines $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{N}^+$ and $\mathbb{B}$.	Clauses 5.3.5 and 5.3.11, Fig. 11 and 17.
Annex A.2 defines *multiset*.	Fig. 10 and 7 define *Terms::MultisetSort* and *multiset operators*.
Annex A.2.2 defines *multiset* membership.	Provided *operator CardinalityOf* in *multisets* package in Fig. 10.
Annexes A.2.2, A.2.3, A.2.4, A.2.5 and A.2.6.	Provided *multisets* package depicted by Fig. 10.
Annex A.3.1 defines *signatures*.	Distinction is made between *built-in sorts*, *user-declared sorts* (as abbreviation of built-in sorts) and *user-defined sorts*. Also distinguished are *user-defined operators* from *built-in* and *user-declared operators*. Their arity are given by the *input* and *output sort* of *Terms::Operator*.
Annex A.3.2 defines *Boolean signature*.	Package Booleans which is built-in.
Annex A.3.3 defines *variables*.	*Terms::Variables* and *Terms::VariableDecl* associated to a *sort*.
Annex A.3.4 defines *terms*.	Captured in package *Terms* (Fig. 7).
Annex A.3.5 defines the semantics through algebra.	Part 2 is concerned with the syntax. Semantics for built-in sorts and operators is implicit. User-defined concepts are not a concern for Part 2. A framework for those constructs is provided in Fig. 20
Annex B1 defines *Place/Transition net graphs*.	Refers to Clauses 5.3.1 and 5.3.12.

Table 3 gives an overview of concepts of Part 2 that are not explicitly defined in Part 1 and gives their counterparts in Part 1.

Table 3: *Definitions of Part 2 not explicitly defined in Part 1*

Part 2	**Part 1**
Terms::ProductSort explicitly introduced due to its frequent use.	Implicit in Annex A.3.1 defining *signature* but not detailed. Mentioned as built-in in Clause 5.5
Pages, *reference nodes*.	Structure not handled in HLPNGs. *Pages* do not appear.
Fig. 16 defining *Symmetric Nets*.	Corresponds to the Amendment (currently PDAM 15909-1.2) to part 1 of ISO/IEC 15909.
Fig. 20 defining *user-defined sorts* and *operators*	Fits Annex A.3 on *signature* (and algebra) and Annex C (HLPNS).

7 PNML Syntax

Clause 5 defines the concepts of the *PNML Core Model* and the concepts of *Place/Transition Nets*, *High-level Petri Net Graphs*, and *Symmetric Nets* in terms of a UML meta model. Clause 5, however, does not define a concrete XML syntax for representing these concepts. The precise XML syntax of the *PNML Core Model* and the three *Petri net types* will be defined in this clause.

Clause 7.1 defines the general format of *PNML Documents*; i. e. the XML syntax for the *PNML Core Model*. Clause 7.2 defines the format for *Place/Transition Nets*. In fact, it gives general rules, how the concepts of a package defining some *Petri net type* are mapped to XML. For *High-level Petri Nets* and *Symmetric Nets* this part of ISO/IEC 15909 defines a dedicated mapping of their *labels* to XML in Clause 7.3.

Note that there is no need for a separate mapping for *Symmetric Nets* and *Place/Transition Nets in High-level Notation* since, as a restricted version of *High-level Petri Nets*, these mappings are fully covered by the mapping for *High-level Petri Nets*.

7.1 PNML Documents

The mapping of the *PNML Core Model* concepts to XML syntax is defined for each class of the *PNML Core Model diagram* separately.

7.1.1 PNML Elements.

Each concrete class[3] of the *PNML Core Model* is mapped to an XML element. The translation of these classes along with the attributes and their data types is given in Table 4. These XML elements are the *keywords* of PNML and are called *PNML elements* for short. For each *PNML element*, the compositions of the *PNML Core Model* define in which elements it may occur as a child element.

The data type ID in Table 4 refers to a set of unique identifiers within the *PNML Document*. The data type IDRef refers to the set of all identifiers occurring in the document, i. e. they are meant as references to identifiers. A reference at some particular position, however, is restricted to objects of a particular type – as defined in the *PNML Core Model*. For instance, the attribute `ref` of a *reference place* must refer to a *place* or a *reference place* of the same *net*. The set to which a reference is restricted, is indicated in the table (e. g. for a *reference place*, the attribute `ref` should refer to the `id` of a *Place* or a *RefPlace*). Note that these requirements are formalised in the UML meta model already; here, these requirement are repeated just for better readability.

7.1.2 Labels

Except for *names*, there are no explicit definitions of *PNML elements* for *labels* because the PNML Core Model does not define other *labels*. For concrete *Petri net types*, such as *Place/Transition Nets*, *Symmetric Nets*, and *High-level Petri Nets* the corresponding packages define these *labels*.

[3] A class in a UML diagram is concrete if its name is not displayed in italics.

Table 4: Translation of the PNML Core Model into PNML elements

Class	XML element	XML Attributes
PetriNetDoc	<pnml>	xmlns: anyURI (http://www.pnml.org/version-2009/grammar/pnml)
PetriNet	<net>	id: ID type: anyURI
Place	<place>	id: ID
Transition	<transition>	id: ID
Arc	<arc>	id: ID source: IDRef (Node) target: IDRef (Node)
Page	<page>	id: ID
RefPlace	<referencePlace>	id: ID ref: IDRef (Place or RefPlace)
RefTrans	<referenceTransition>	id: ID ref: IDRef (Transition or RefTrans)
ToolInfo	<toolspecific>	tool: string version: string
Graphics	<graphics>	
Name	<name>	

Table 5: Possible child elements of the <graphics> element

Parent element class	Sub-elements of <graphics>
Node, Page	<position> <dimension> <fill> <line>
Arc	<position> (zero or more) <line>
Annotation	<offset> <fill> <line> <font>

In general *PNML Documents*, any XML element that is not defined in the *PNML Core Model* (i. e. not occurring in Table 4) is considered as a *label* of the *PNML element* in which it occurs. For example, an <initialMarking> element could be a *label* of a *place*, which represents its initial marking. Likewise <name> represent the *name*[4] of an *object*, and <inscription> an arc annotation.

A legal element for a *label* must contain at least one of the two following elements, which represents the actual value of the *label*: a <text> element represents the value of the *label* as a simple string; the <structure> element can be used for representing the value as an abstract syntax tree in XML.

An optional PNML <graphics> element defines its graphical appearance; and optional PNML <toolspecific> elements may add tool specific information to the label. Note that this part of ISO/IEC 15909 does not mandate the inner structure of <toolspecific> elements; every tool is free to structure its information inside that element at its discretion.

7.1.3 Graphics

All *PNML elements* and all *labels* may include graphical information. The internal structure of the PNML <graphics> element, i. e. the legal XML children, depends on the element in which the graphics element occurs. Table 5 shows the elements which may occur within the <graphics> element (as defined by the UML model in Fig. 3).

[4]Note that this, actually, is the only *label* that is explicitly defined in the *PNML Core Model*.

The <position> element defines an absolute position for *nodes* and *pages*, whereas the <offset> element defines a relative position for *annotation*. The name of the element comes from the corresponding role in the UML diagram, and the possible attributes are derived from the attributes of the corresponding class in the UML diagram.

Table 6 explicitly list the attributes for each graphical element defined in Table 5 (cf. Fig. 3 and Table 1). The domain of the attributes refers to the data types of either XML Schema, or Cascading Stylesheets 2 (CSS2), or is given by an explicit enumeration of the legal values.

Table 6: PNML graphical elements

XML element	Attribute	Domain
<position>	x	decimal
	y	decimal
<offset>	x	decimal
	y	decimal
<dimension>	x	nonNegativeDecimal
	y	nonNegativeDecimal
<fill>	color	CSS2-color
	image	anyURI
	gradient-color	CSS2-color
	gradient-rotation	{vertical, horizontal, diagonal}
<line>	shape	{line, curve}
	color	CSS2-color
	width	nonNegativeDecimal
	style	{solid, dash, dot}
<font>	family	CSS2-font-family
	style	CSS2-font-style
	weight	CSS2-font-weight
	size	CSS2-font-size
	decoration	{underline, overline, line-through}
	align	{left, center, right}
	rotation	decimal

The meaning of these elements and attributes is defined in Clause 5.2.4.

7.1.4 Mapping of XMLSchemaDataTypes concepts

The concepts from the package *XMLSchemaDataTypes* are mapped to XML syntax in the following way. The *String* objects are mapped to XML PCDATA, i. e. there will be a PCDATA section within the element which contains the *String*. This, basically, corresponds to any printable text.

Likewise, *Integers*, *NonNegativeIntegers* and *PositiveIntegers* are mapped to the XMLSchema syntax constructs *integer*, *nonNegativeInteger*, and *positiveInteger*, respectively.

7.1.5 Example

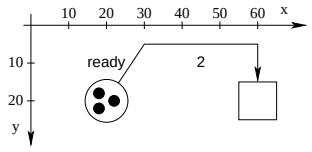

Figure 23: A simple *Place/Transition Net*

In order to illustrate the structure of a *PNML Document*, there is a simple example *PNML Document* representing the *Petri net* shown in Fig. 23, which actually is a *Place/Transition Net*. Listing 1 shows the corresponding *PNML*

Listing 1: PNML code of the example net in Fig. 23

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

Document in XML syntax. It is a straight-forward translation, where there are *labels* for the *names* of objects, for the initial markings, and for *arc annotations*.

Note that a tool does not display the initial marking as a textual *label*. Since the *initial marking* comes with a toolspecific information on the positions of the *tokens*, tokens are shown at the individual positions as given in the elements `<tokenposition>`.

Since there is no information on the dimensions in this example (in order to fit the listing to a single page), the tool has chosen its default dimensions for the place and the transition.

7.2 Mapping Petri Net Type Definitions to XML Syntax

Based on the *PNML Core Model* of Clause 5.2, Clauses 5.3.1 to 5.3.11 define the concepts of two particular *Petri net types*, which restrict *PNML Documents* to the particular *labels* defined in the corresponding packages.

These packages define the *labels* that are used in the particular *Petri net*. Here, it is shown how to map such a package to the corresponding XML syntax. This mapping is the same for all type definitions, unless the type definition comes with a separate mapping. This general mapping will be explained by the help of the example of *Place/Transition Nets* (see Fig. 5 and the *PNML Document* in Listing 1).

The PT-Net package defines two kinds of *labels* that can be used in a *Place/Transition Net*: PTMarkings and PTAnnotations. Each *place* can have one *annotation* PTMarking, and each *arc* can have one *annotation* PTAnnotation. This is indicated by the compositions in the UML diagram in Fig. 5.

The XML syntax for these *labels* is derived from the roles of these compositions. Every *annotation* PTMarking is mapped to an XML element `<initialMarking>`, and every *annotation* PTAnnotation is mapped to an element `<inscription>`. Listing 1 shows an example for the XML syntax of a *Place/Transition Net*.

Since all *labels* in this package are *annotations*, all graphical elements defined for *annotations* may occur as children in these elements.

In the PT-Net package each *label* is defined to have a `<text>` element, which defines the actual content of this *annotation*. For *Place/Transition Nets* there are no structured elements.

The corresponding classes from package DataTypes define the XML content of the `<text>` element. These definitions are given in Clause 7.1.4.

Note that for *Place/Transition Nets*, there is a predefined toolspecific extension for the label `<initialMarking>` (see Clause 5.3.1), which represents the positions of tokens within a place. The class *TokenGraphics* is mapped to the XML element `<tokengraphics>` with no attributes. The token positions contained by this element are represented by the elements `<tokenposition>` as shown in Table 7. Within the element `<tokengraphics>` there can be any number of `<tokenposition>` elements.

Table 7: Tool specific information for token positions

XML element	Attribute	Domain
`<tokengraphics>`		
`<tokenposition>`	x	decimal
	y	decimal

7.3 Mapping for *High-Level Nets*

7.3.1 Mapping High-Level Nets meta model elements to PNML syntax

In Table 8, the mappings between the meta models elements and their PNML constructs using RELAX NG grammar are defined. In these tables, only meta model elements that have corresponding PNML elements or attributes are displayed. Thus, most abstract elements are not shown in the tables, except if they have attributes. All attribute types are mapped to *XMLSchema-datatypes* library data types.

Table 8: *High-level meta model elements and their PNML constructs*

Model element	PNML element	PNML attributes
Booleans::Bool	bool	
Booleans::And	and	
Booleans::Or	or	
Booleans::Not	not	
Booleans::Imply	imply	
Booleans::Equality	equality	
Booleans::Inequality	inequality	
Booleans::BooleanConstant	booleanconstant	value: boolean
CyclicEnumerations::CyclicEnumeration	cyclicenumeration	
CyclicEnumerations::Successor	successor	
CyclicEnumerations::Predecessor	predecessor	
Dots::Dot	dot	
Dots::DotConstant	dotconstant	
FiniteEnumerations::FiniteEnumeration	finiteenumeration	
FiniteEnumerations::FEConstant	feconstant	name: string; id: ID
FiniteIntRanges::FiniteIntRange	finiteintrange	start, end: integer
FiniteIntRanges::LessThan	lessthan	
FiniteIntRanges::LessThanOrEqual	lessthanorequal	
FiniteIntRanges::GreaterThan	greaterthan	
FiniteIntRanges::GreaterThanOrEqual	greaterthanorequal	
FiniteIntRanges: FiniteIntRangeConstant	finiteintrangeconstant	value: integer; range: IDREF
Integers::Integer	integer	
Integers::Natural	natural	
Integers::Positive	positive	
Integers::NumberConstant	numberconstant	value: integer
Integers::GreaterThan	gt	
Integers::GreaterThanOrEqual	geq	
Integers::LessThan	lt	
Integers::LessThanOrEqual	leq	
Integers::Addition	add	
Integers::Subtraction	subtraction	
Integers::Multiplication	mult	
Integers::Division	div	
Integers::Modulo	mod	
HLCoreStructure::Declaration	declaration	
HLCoreStructure::Type	type	
HLCoreStructure::HLMarking	hlinitialmarking	
HLCoreStructure::Condition	condition	
HLCoreStructure::HLAnnotation	hlinscription	
Lists::List	list	
Lists::EmpyList	emptylist	
Lists::Length	listlength	
Lists::MakeList	makelist	
Lists::Concatenation	listconcatenation	
Lists::Sublist	sublist	start, length: nonNegativeInteger
Lists::Append	listappend	
Lists::MemberAtIndex	memberatindex	index: nonNegativeInteger

Table 8: *High-level meta model elements and their PNML constructs (cntd.)*

Model element	PNML element	PNML attributes
Multisets::Add	add	
Multisets::Subtract	subtract	
Multisets::All	all	
Multisest::Empty	empty	
Multisets::ScalarProduct	scalarproduct	
Multisets::NumberOf	numberof	
Multisets::Cardinality	cardinality	
Multisets::CardinalityOf	cardinalityof	
Multisets::Contains	contains	
Partitions::Partition	partition	
Partitions::PartitionElement	partitionelement	name: String
Partitions::LessThan	ltp	
Partitions::GreaterThan	gtp	
Partitions::PartitionElementOf	partitionelementof	
Strings::String	string	
Strings::Append	stringappend	
Strings::LessThan	lts	
Strings::LessThanOrEqual	leqs	
Strings::GreaterThan	gts	
Strings::GreaterThanOrEqual	geqs	
Strings::StringConstant	stringconstant	value: string
Strings::Concatenation	stringconcatenation	
Strings::Length	stringlength	
Strings::Substring	substring	start, length: nonNegativeInteger
Terms::Declarations	declarations	
Terms::VariableDeclaration	variabledecl	id: ID; name: string
Terms::OperatorDeclaration	No element (abstract class)	id: ID; name: string
Terms::SortDeclaration	No element (abstract class)	id: ID; name: string
Terms::Variable	variable	variabledecl: IDREF
Terms::NamedSort	namedsort	
Terms::NamedOperator	namedoperator	
Terms::Term	No element (abstract class)	
Terms::Sort	No element (abstract class)	
Terms::MultisetSort	multisetsort	
Terms::ProductSort	productsort	
Terms::UserSort	usersort	declaration: IDREF
Terms::Tuple	tuple	
Terms::Operator	No element (abstract class)	
Terms::UserOperator	useroperator	declaration: IDREF
ArbitraryDeclarations::ArbitrarySort	arbitrarysort	
ArbitraryDeclarations::Unparsed	unparsed	
ArbitraryDeclarations::ArbitraryOperator	arbitraryoperator	

(Blank page)

Annex A
(normative)

RELAX NG Grammar for the PNML Core Model

This Annex gives a complete definition of the *PNML Core Model* and its XML format in terms of a RELAX NG grammar (ISO/IEC 19757-2:2008). Unlike earlier versions of PNML grammar, especially *v-1.3.2*, this definition encompasses pages construct as specified in Clause 5.2.

Note that some syntactical restrictions of the *PNML Core Model* cannot be expressed in RELAX NG. Still, these restrictions as defined in Clause 5 are mandatory for valid *PNML Documents*. The RELAX NG grammar distills the more syntactical requirements on the XML transfer format.

In the following the general grammar defining the Core Model is provided in the file **pnmlcoremodel.rng**

```
<?xml version="1.0" encoding="UTF-8"?>

<grammar ns="http://www.pnml.org/version-2009/grammar/pnml"
        xmlns="http://relaxng.org/ns/structure/1.0"
        xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
        datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">

        <a:documentation>
                Petri Net Markup Language (PNML) schema.
                RELAX NG implementation of PNML Core Model.

                File name: pnmlcoremodel.rng
                Version: 2009
                (c) 2001-2009
                Michael Weber,
                Ekkart Kindler,
                Christian Stehno,
                Lom Hillah (AFNOR)
                Revision:
                July 2008 - L.H
        </a:documentation>

        <include href="http://www.pnml.org/version-2009/grammar/anyElement.rng"/>

        <start>
                <ref name="pnml.element"/>
        </start>

        <define name="pnml.element">
                <element name="pnml">
                        <a:documentation>
                                A PNML document consists of one or more Petri nets.
                                It has a version.
                        </a:documentation>
                        <oneOrMore>
                                <ref name="pnml.content"/>
                        </oneOrMore>
                </element>
        </define>

        <define name="pnml.content">
                <ref name="net.element"/>
        </define>

        <define name="net.element">
                <element name="net">
```

```
                <a:documentation>
                        A net has a unique identifier (id) and refers to
                        its Petri Net Type Definition (PNTD) (type).
                </a:documentation>
                <ref name="identifier.content"/>
                <ref name="nettype.uri"/>
                <a:documentation>
                        The sub-elements of a net may occur in any order.
                        A net consists of at least a top-level page which
                        may contain several objects. A net may have a name,
                        other labels (net.labels) and tool specific information in any order.
                </a:documentation>
                <interleave>
                        <optional>
                                <ref name="Name"/>
                        </optional>
                        <ref name="net.labels"/>
                        <oneOrMore>
                                <ref name="page.content"/>
                        </oneOrMore>
                        <zeroOrMore>
                                <ref name="toolspecific.element"/>
                        </zeroOrMore>
                </interleave>
        </element>
</define>

<define name="identifier.content">
        <a:documentation>
                Identifier (id) declaration shared by all objects in any PNML model.
        </a:documentation>
        <attribute name="id">
                <data type="ID"/>
        </attribute>
</define>

<define name="nettype.uri">
        <a:documentation>
                The net type (nettype.uri) of a net should be redefined in the grammar
                for a new Petri net Type.
                An example of such a definition is in ptnet.pntd, the grammar
                for P/T Nets. The following value is a default.
        </a:documentation>
        <attribute name="type">
                <value>http://www.pnml.org/version-2009/grammar/pnmlcoremodel</value>
        </attribute>
</define>

<define name="net.labels">
        <a:documentation>
                A net may have unspecified many labels. This pattern should be used
                within a PNTD to define the net labels.
        </a:documentation>
        <empty/>
</define>

<define name="basicobject.content">
        <a:documentation>
                Basic contents for any object of a PNML model.
        </a:documentation>
        <interleave>
```



```
                <optional>
                        <ref name="Name"/>
                </optional>
                <zeroOrMore>
                        <ref name="toolspecific.element"/>
                </zeroOrMore>
        </interleave>
</define>


<define name="page.content">
        <a:documentation>
                A page has an id. It may have a name and tool specific information.
                It may also have graphical information. It can also have many arbitrary labels.
                Note: according to this definition, a page may contain other pages.
                All these sub-elements may occur in any order.
        </a:documentation>
        <element name="page">
                <ref name="identifier.content"/>
                <interleave>
                        <ref name="basicobject.content"/>
                        <ref name="page.labels"/>
                        <zeroOrMore>
                                <ref name="netobject.content"/>
                        </zeroOrMore>
                        <optional>
                                <element name="graphics">
                                        <ref name="pagegraphics.content"/>
                                </element>
                        </optional>
                </interleave>
        </element>
</define>

<define name="netobject.content">
        <a:documentation>
                A net object is either a page, a node or an arc.
                A node is a place or a transition, a reference place of
                a reference transition.
        </a:documentation>
        <choice>
                <ref name="page.content"/>
                <ref name="place.content"/>
                <ref name="transition.content"/>
                <ref name="refplace.content"/>
                <ref name="reftrans.content"/>
                <ref name="arc.content"/>
        </choice>
</define>

<define name="page.labels">
        <a:documentation>
                A page may have unspecified many labels. This pattern should be used
                within a PNTD to define new labels for the page concept.
        </a:documentation>
        <empty/>
</define>

<define name="place.content">
        <a:documentation>
                A place may have several labels (place.labels) and the same content
```

```
                as a node.
        </a:documentation>
        <element name="place">
                <ref name="identifier.content"/>
                <interleave>
                        <ref name="basicobject.content"/>
                        <ref name="place.labels"/>
                        <ref name="node.content"/>
                </interleave>
        </element>
</define>

<define name="place.labels">
        <a:documentation>
                A place may have arbitrary many labels. This pattern should be used
                within a PNTD to define the place labels.
        </a:documentation>
        <empty/>
</define>

<define name="transition.content">
        <a:documentation>
                A transition may have several labels (transition.labels) and the same
                content as a node.
        </a:documentation>
        <element name="transition">
                <ref name="identifier.content"/>
                <interleave>
                        <ref name="basicobject.content"/>
                        <ref name="transition.labels"/>
                        <ref name="node.content"/>
                </interleave>
        </element>
</define>

<define name="transition.labels">
        <a:documentation>
                A transition may have arbitrary many labels. This pattern should be
                used within a PNTD to define the transition labels.
        </a:documentation>
        <empty/>
</define>

<define name="node.content">
        <a:documentation>
                A node may have graphical information.
        </a:documentation>
        <optional>
                <element name="graphics">
                        <ref name="nodegraphics.content"/>
                </element>
        </optional>
</define>

<define name="reference">
        <a:documentation>
                Here, we define the attribute ref including its data type.
                Modular PNML will extend this definition in order to change
                the behavior of references to export nodes of module instances.
        </a:documentation>
        <attribute name="ref">
```



```
                <data type="IDREF"/>
        </attribute>
</define>

<define name="refplace.content">
        <a:documentation>
                A reference place is a reference node.
        </a:documentation>
        <a:documentation>
                Validating instruction:
                - _ref_ MUST refer to _id_ of a reference place or of a place.
                - _ref_ MUST NOT refer to _id_ of its reference place element.
                - _ref_ MUST NOT refer to a cycle of reference places.
        </a:documentation>
        <element name="referencePlace">
                <ref name="refnode.content"/>
        </element>
</define>

<define name="reftrans.content">
        <a:documentation>
                A reference transition is a reference node.
        </a:documentation>
        <a:documentation>
                Validating instruction:
                - The reference (ref) MUST refer to a reference transition or to a
                transition.
                - The reference (ref) MUST NOT refer to the identifier (id) of its
                reference transition element.
                - The reference (ref) MUST NOT refer to a cycle of reference transitions.
        </a:documentation>
        <element name="referenceTransition">
                <ref name="refnode.content"/>
        </element>
</define>

<define name="refnode.content">
        <a:documentation>
                A reference node has the same content as a node.
                It adds a reference (ref) to a (reference) node.
        </a:documentation>
        <ref name="identifier.content"/>
        <ref name="reference"/>
        <ref name="basicobject.content"/>
        <ref name="node.content"/>
</define>

<define name="arc.content">
        <a:documentation>
                An arc has a unique identifier (id) and
                refers both to the node´s id of its source and
                the node´s id of its target.
                In general, if the source attribute refers to a place,
                then the target attribute refers to a transition and vice versa.
        </a:documentation>
        <element name="arc">
                <ref name="identifier.content"/>
                <attribute name="source">
                        <data type="IDREF"/>
                </attribute>
                <attribute name="target">
```

```
                        <data type="IDREF"/>
                    </attribute>
                    <a:documentation>
                        The sub-elements of an arc may occur in any order.
                        An arc may have a name, graphical and tool specific information.
                        It may also have several labels.
                    </a:documentation>
                    <interleave>
                        <optional>
                            <ref name="Name"/>
                        </optional>
                        <ref name="arc.labels"/>
                        <optional>
                            <element name="graphics">
                                <ref name="edgegraphics.content"/>
                            </element>
                        </optional>
                        <zeroOrMore>
                            <ref name="toolspecific.element"/>
                        </zeroOrMore>
                    </interleave>
                </element>
            </define>

            <define name="arc.labels">
                <a:documentation>
                    An arc may have arbitrary many labels. This pattern should be used
                    within a PNTD to define the arc labels.
                </a:documentation>
                <empty/>
            </define>

            <define name="pagegraphics.content">
                <a:documentation>
                    A page graphics is actually a node graphics
                </a:documentation>
                <ref name="nodegraphics.content"/>
            </define>

            <define name="nodegraphics.content">
                <a:documentation>
                    The sub-elements of a node´s graphical part occur in any order.
                    At least, there may be one position element.
                    Furthermore, there may be a dimension, a fill, and a line element.
                </a:documentation>
                <interleave>
                    <ref name="position.element"/>
                    <optional>
                        <ref name="dimension.element"/>
                    </optional>
                    <optional>
                        <ref name="fill.element"/>
                    </optional>
                    <optional>
                        <ref name="line.element"/>
                    </optional>
                </interleave>
            </define>

            <define name="edgegraphics.content">
                <a:documentation>
```

```
                The sub-elements of an arc´s graphical part occur in any order.
                There may be zero or more position elements.
                Furthermore, there may be a line element.
        </a:documentation>
        <interleave>
                <zeroOrMore>
                        <ref name="position.element"/>
                </zeroOrMore>
                <optional>
                        <ref name="line.element"/>
                </optional>
        </interleave>
</define>

<define name="simpletext.content">
        <a:documentation>
                This definition describes the contents of simple text labels
                without graphics.
        </a:documentation>
        <optional>
                <element name="text">
                        <a:documentation>
                                A text should have a value.
                                If not, then there must be a default.
                        </a:documentation>
                        <text/>
                </element>
        </optional>
</define>

<define name="annotationstandard.content">
        <a:documentation>
                The definition annotationstandard.content describes the
                standard contents of an annotation.
                Each annotation may have graphical or tool specific information.
        </a:documentation>
        <interleave>
                <optional>
                        <element name="graphics">
                                <ref name="annotationgraphics.content"/>
                        </element>
                </optional>
                <zeroOrMore>
                        <ref name="toolspecific.element"/>
                </zeroOrMore>
        </interleave>
</define>

<define name="simpletextlabel.content">
        <a:documentation>
                A simple text label is an annotation to a net object containing
                arbitrary text.
                Its sub-elements occur in any order.
                A simple text label behaves like an attribute to a net object.
                Furthermore, it contains the standard annotation contents which
                basically defines the graphics of the text.
        </a:documentation>
        <interleave>
                <ref name="simpletext.content"/>
                <ref name="annotationstandard.content"/>
        </interleave>
```

```
</define>

<define name="Name">
        <a:documentation>
                Label definition for a user given name of an
                element.
        </a:documentation>
        <element name="name">
                <ref name="simpletextlabel.content"/>
        </element>
</define>

<define name="annotationgraphics.content">
        <a:documentation>
                An annotation´s graphics part requires an offset element describing
                the offset the center point of the surrounding text box has to
                the reference point of the net object on which the annotation occurs.
                Furthermore, an annotation´s graphic element may have a fill, a line,
                and font element.
        </a:documentation>
        <ref name="offset.element"/>
        <interleave>
                <optional>
                        <ref name="fill.element"/>
                </optional>
                <optional>
                        <ref name="line.element"/>
                </optional>
                <optional>
                        <ref name="font.element"/>
                </optional>
        </interleave>
</define>

<define name="position.element">
        <a:documentation>
                A position element describes Cartesian coordinates.
        </a:documentation>
        <element name="position">
                <ref name="coordinate.attributes"/>
        </element>
</define>

<define name="offset.element">
        <a:documentation>
                An offset element describes Cartesian coordinates.
        </a:documentation>
        <element name="offset">
                <ref name="coordinate.attributes"/>
        </element>
</define>

<define name="coordinate.attributes">
        <a:documentation>
                The coordinates are decimal numbers and refer to an appropriate
                xy-system where the x-axis runs from left to right and the y-axis
                from top to bottom.
        </a:documentation>
        <attribute name="x">
                <data type="decimal"/>
        </attribute>
```

```
        <attribute name="y">
                <data type="decimal"/>
        </attribute>
</define>

<define name="dimension.element">
        <a:documentation>
                A dimension element describes the width (x coordinate) and height
                (y coordinate) of a node.
                The coordinates are actually positive decimals.
        </a:documentation>
        <element name="dimension">
                <attribute name="x">
                        <ref name="positiveDecimal.content"/>
                </attribute>
                <attribute name="y">
                        <ref name="positiveDecimal.content"/>
                </attribute>
        </element>
</define>

<define name="positiveDecimal.content" ns="http://www.w3.org/2001/XMLSchema-datatypes">
        <a:documentation>
                Definition of a restricted positive decimals domain with a total digits
                number of 4 and 1 fraction digit. Ranges from 0 to 999.9
        </a:documentation>
        <data type='decimal'>
                <param name='totalDigits'>4</param>
                <param name='fractionDigits'>1</param>
                <param name='minExclusive'>0</param>
        </data>
</define>

<define name="fill.element">
        <a:documentation>
                A fill element describes the interior colour, the gradient colour,
                and the gradient rotation between the colors of an object.  If an
                image is available the other attributes are ignored.
        </a:documentation>
        <element name="fill">
                <optional>
                        <attribute name="color">
                                <ref name="color.type"/>
                        </attribute>
                </optional>
                <optional>
                        <attribute name="gradient-color">
                                <ref name="color.type"/>
                        </attribute>
                </optional>
                <optional>
                        <attribute name="gradient-rotation">
                                <choice>
                                        <value>vertical</value>
                                        <value>horizontal</value>
                                        <value>diagonal</value>
                                </choice>
                        </attribute>
                </optional>
                <optional>
                        <attribute name="image">
```

```
                                        <data type="anyURI"/>
                                </attribute>
                        </optional>
                </element>
        </define>

        <define name="line.element">
                <a:documentation>
                        A line element describes the shape, the colour, the width, and the
                        style of an object.
                </a:documentation>
                <element name="line">
                        <optional>
                                <attribute name="shape">
                                        <choice>
                                                <value>line</value>
                                                <value>curve</value>
                                        </choice>
                                </attribute>
                        </optional>
                        <optional>
                                <attribute name="color">
                                        <ref name="color.type"/>
                                </attribute>
                        </optional>
                        <optional>
                                <attribute name="width">
                                        <ref name="positiveDecimal.content"/>
                                </attribute>
                        </optional>
                        <optional>
                                <attribute name="style">
                                        <choice>
                                                <value>solid</value>
                                                <value>dash</value>
                                                <value>dot</value>
                                        </choice>
                                </attribute>
                        </optional>
                </element>
        </define>

        <define name="color.type">
                <a:documentation>
                        This describes the type of a color attribute. Actually, this comes
                        from the CSS2 (and latest versions) data type system.
                </a:documentation>
                <text/>
        </define>

        <define name="font.element">
                <a:documentation>
                        A font element describes several font attributes, the decoration,
                        the alignment, and the rotation angle of an annotation´s text.
                        The font attributes (family, style, weight, size) should be conform
                        to the CSS2 and latest versions data type system.
                </a:documentation>
                <element name="font">
                        <optional>
                                <attribute name="family">
                                        <text/>  <!-- actually, CSS2 and latest versions font-family -->
```



```
                        </attribute>
                    </optional>
                    <optional>
                        <attribute name="style">
                            <text/>  <!-- actually, CSS2 and latest versions font-style -->
                        </attribute>
                    </optional>
                    <optional>
                        <attribute name="weight">
                            <text/>  <!-- actually, CSS2 and latest versions font-weight -->
                        </attribute>
                    </optional>
                    <optional>
                        <attribute name="size">
                            <text/>  <!-- actually, CSS2 and latest versions font-size -->
                        </attribute>
                    </optional>
                    <optional>
                        <attribute name="decoration">
                            <choice>
                                <value>underline</value>
                                <value>overline</value>
                                <value>line-through</value>
                            </choice>
                        </attribute>
                    </optional>
                    <optional>
                        <attribute name="align">
                            <choice>
                                <value>left</value>
                                <value>center</value>
                                <value>right</value>
                            </choice>
                        </attribute>
                    </optional>
                    <optional>
                        <attribute name="rotation">
                            <data type="decimal"/>
                        </attribute>
                    </optional>
                </element>
            </define>

            <define name="toolspecific.element">
                <a:documentation>
                    The tool specific information refers to a tool and its version.
                    The further substructure is up to the tool.
                </a:documentation>
                <element name="toolspecific">
                    <attribute name="tool">
                        <text/>
                    </attribute>
                    <attribute name="version">
                        <text/>
                    </attribute>
                    <zeroOrMore>
                        <ref name="anyElement"/>
                    </zeroOrMore>
                </element>
            </define>
```

```
</grammar>
```

In the following the grammar of the generic *anyElement* construct is described. It is provided in the file **anyElement.rng**.

```
<?xml version="1.0" encoding="UTF-8"?>

<grammar xmlns="http://relaxng.org/ns/structure/1.0"
        xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
        datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">

        <a:documentation>
                RELAX NG implementation the generic 'anyElement' construct.
                This construct may be useful in any grammar which does not
                directly depend on pnmlcoremodel.rng. That is why it is made
                independant of pnmlcoremodel.rng.

                File name: anyElement.rng
                Version: 2009
                (c) 2007-2009
                Lom Hillah (AFNOR)
                Revision:
                June 2008 - L.H
        </a:documentation>

        <define name="anyElement">
                <element>
                        <anyName/>
                        <zeroOrMore>
                                <choice>
                                        <attribute>
                                                <anyName/>
                                        </attribute>
                                        <text/>
                                        <ref name="anyElement"/>
                                </choice>
                        </zeroOrMore>
                </element>
        </define>

</grammar>
```


Annex B
(normative)

RELAX NG Grammars for special types

This Annex gives a complete definition of the XML syntax of the *Petri net types* defined in this part of ISO/IEC 15909. There is a separate RELAX NG grammar for each *Petri net type*.

Note that some syntactical restrictions in the meta models the *Petri net types* cannot be expressed in RELAX NG. Still, these restrictions as defined in Clause 5.3 are mandatory for valid *PNML Documents*.

B.1 Place/Transition Nets

B.1.1 The labels

First, there is a definition of the labels data for *Place/Transition Nets* in the file **conventions.rng**.

```
<?xml version="1.0" encoding="UTF-8"?>

<grammar xmlns="http://relaxng.org/ns/structure/1.0"
        xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0">

        <a:documentation>
                RELAX NG implementation of the Conventions Document.
                These conventions are short cuts for label definitions. They are
                used for simple data or if the label data is not really specified.

                File name: conventions.rng
                Version: 2009
                (c) 2007-2009
                Michael Weber,
                Lom Hillah (AFNOR)
                Revision:
                June 2008 - L.H
        </a:documentation>

        <define name="nonnegativeintegerlabel.content">
                <a:documentation>
                        A non negative integer label is an annotation with a
                        natural number as its value.
                        Its sub-elements occur in any order.
                        It contains the standard annotation content.
                </a:documentation>
                <interleave>
                        <element name="text">
                                <data type="nonNegativeInteger"
                                        datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes"/>
                        </element>
                        <ref name="annotationstandard.content"/>
                </interleave>
        </define>

        <define name="positiveintegerlabel.content">
                <a:documentation>
                        A positive integer label is an annotation with a natural
                        number as its value, zero excluded.
                        Its sub-elements occur in any order.
                        It contains the standard annotation content.
                </a:documentation>
```

```
                <interleave>
                        <element name="text">
                                <data type="positiveInteger"
                                        datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes"/>
                        </element>
                        <ref name="annotationstandard.content"/>
                </interleave>
        </define>

</grammar>
```

B.1.2 Token Graphics

Place/Transition Nets can use the token graphics grammar to include individual tokens positions as tool specific feature. This grammar is in the file **pnmlextensions.rng**.

```
<?xml version="1.0" encoding="UTF-8"?>

<grammar xmlns="http://relaxng.org/ns/structure/1.0"
        xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
        datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">

        <a:documentation>
                RELAX NG implementation of PNML Extensions package.
                This package is meant to define specific extensions to
                PT-Net type that would appear mostly as tool specific
                features. Those features are not semantically speaking essential
                to the exchange of PNML models.

                File name: pnmlextensions.rng
                Version: 2009
                (c) 2007-2009
                Lom Hillah (AFNOR)
                Revision:
                July 2008 - L.H
        </a:documentation>


        <define name="toolspecific.content" combine="interleave">
                <a:documentation>
                        Individual tokens positions as toolspecific elements.
                </a:documentation>
                <optional>
                        <ref name="TokenGraphics"/>
                </optional>
        </define>

        <define name="TokenGraphics">
                <a:documentation>
                        Individual tokens, may be have positions.
                        This element gathers them.
                        There should be as many token positions as the initial
                        marking states.
                </a:documentation>
                <element name="tokengraphics">
                        <zeroOrMore>
                                <ref name="TokenPosition"/>
                        </zeroOrMore>
                </element>
```



```
        </define>

        <define name="TokenPosition">
                <a:documentation>
                        The actual position of an individual token.
                        It is a coordinate.attributes.
                </a:documentation>
                <element name="tokenposition">
                        <ref name="coordinate.attributes"/>
                </element>
        </define>

</grammar>
```

B.1.3 The Grammar

The following RELAX NG grammar defines the XML syntax of *Place/Transition Nets* in the file **ptnet.pntd**

```
<?xml version="1.0" encoding="UTF-8"?>

<grammar ns="http://www.pnml.org/version-2009/grammar/pnml"
        xmlns="http://relaxng.org/ns/structure/1.0"
        xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0">

        <a:documentation>
                RELAX NG implementation of Petri Net Type Definition for Place/Transition nets.
                This PNTD re-defines the value of nettype.uri for P/T nets.

                File name: ptnet.pntd
                Version: 2009
                (c) 2007-2009
                Lom Hillah (AFNOR)
                Revision:
                July 2008 - L.H
        </a:documentation>

        <a:documentation>
                The PT Net type definition.
                This document also declares its namespace.
                All labels of this Petri net type come from the Conventions document.
                The use of token graphics as tool specific feature is possible.
        </a:documentation>

        <include href="http://www.pnml.org/version-2009/grammar/conventions.rng"/>

        <!--
        <include href="http://www.pnml.org/version-2009/grammar/pnmlextensions.rng"/>
                We do not need to include this, because the pnmlcoremodel.rng covers any
                toolspecific extension.
        -->

        <include href="http://www.pnml.org/version-2009/grammar/pnmlcoremodel.rng"/>

        <define name="nettype.uri" combine="choice">
                <a:documentation>
                        The URI value for the net type attribute,
                        declaring the type of P/T nets.
                </a:documentation>
                <attribute name="type">
```

```
                <value>http://www.pnml.org/version-2009/grammar/ptnet</value>
            </attribute>
        </define>

        <define name="PTMarking">
            <a:documentation>
                Label definition for initial marking in nets like P/T-nets.
                <contributed>Michael Weber</contributed>
                <date>2003-06-16</date>
                <reference>
                    W. Reisig: Place/transition systems. In: LNCS 254. 1987.
                </reference>
            </a:documentation>
            <element name="initialMarking">
                <ref name="nonnegativeintegerlabel.content"/>
            </element>
        </define>

        <define name="PTArcAnnotation">
            <a:documentation>
                Label definition for arc inscriptions in P/T-nets.
                <contributed>Michael Weber, AFNOR</contributed>
                <date>2003-06-16</date>
                <reference>
                    W. Reisig: Place/transition systems. In: LNCS 254. 1987.
                </reference>
            </a:documentation>
            <element name="inscription">
                <ref name="positiveintegerlabel.content"/>
            </element>
        </define>

        <define name="place.labels" combine="interleave">
            <a:documentation>
                A place of a P/T net may have an initial marking.
            </a:documentation>
            <optional><ref name="PTMarking"/></optional>
        </define>

        <define name="arc.labels" combine="interleave">
            <a:documentation>
                An arc of a P/T net may have an inscription.
            </a:documentation>
            <optional><ref name="PTArcAnnotation"/></optional>
        </define>

</grammar>
```


B.2 High-level Petri Nets

B.2.1 Core structure of HLPNGs

First, the grammar for Terms is defined in the file **terms.rng**.

```
<?xml version="1.0" encoding="UTF-8"?>

<grammar datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes"
   xmlns="http://relaxng.org/ns/structure/1.0"
```



```
    xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0">

        <a:documentation>
                RELAX NG implementation of High-level Petri Nets Terms grammar.
                This schema implements the core signature shared by High-level Petri Nets.

                File name: terms.rng
                Version: 2009
                (c) 2007-2009
                Lom Hillah (AFNOR)
                Revision:
                July 2008 - L.H
        </a:documentation>

        <!-- Definition of signature and variable -->
        <define name="Declarations">
                <a:documentation>
                        A core signature starts by zero or more declarations
                </a:documentation>
                <element name="declarations">
                        <zeroOrMore>
                                <ref name="Declaration"/>
                        </zeroOrMore>
                </element>
        </define>

        <define name="Declaration.content">
                <a:documentation>
                        A declaration may be a Sort, Variable or Operator declaration.
                        It has a name and an id.
                        It is extended by each definition of Sort, Variable and Operator.
                </a:documentation>
                <attribute name="id">
                        <data type="ID"/>
                </attribute>
                <attribute name="name">
                        <data type="string"/>
                </attribute>
        </define>

        <define name="Declaration">
                <a:documentation>
                 A declaration is part of a group of declarations.
                 It defines known concrete declarations
                </a:documentation>
                <choice>
                    <ref name="SortDeclaration"/>
                    <ref name="VariableDeclaration"/>
                    <ref name="OperatorDeclaration"/>
                </choice>
        </define>

        <define name="VariableDeclaration">
                <a:documentation>
                        A variable declaration is a user-declared variable.
                        It refers to a Sort.
                </a:documentation>
                <element name="variabledecl">
                        <ref name="Declaration.content"/>
                        <ref name="Sort"/>
                </element>
```

```
        </define>

        <define name="SortDeclaration.content">
                <a:documentation>
                        A Sort declaration content is derived from Declaration.content
                </a:documentation>
                <ref name="Declaration.content"/>
        </define>

        <define name="SortDeclaration">
                <a:documentation>
                        A Sort declaration is a user-declared sort, using built-in sorts.
                        It defines known concrete sort declarations.
                </a:documentation>
                <ref name="NamedSort"/>
        </define>

        <define name="OperatorDeclaration.content">
                <a:documentation>
                        The content of OperatorDeclaration is the one of Declaration
                        constructs.
                </a:documentation>
                <ref name="Declaration.content"/>
        </define>

        <define name="OperatorDeclaration">
                <a:documentation>
                        An Operator declaration is a user-declared operator using built-in.
                        constructs.
                </a:documentation>
                <ref name="NamedOperator"/>
        </define>

        <define name="Variable">
                <a:documentation>
                        A simple variable refers to a VariableDeclaration.
                        As a Term, a variable also refers to a Sort, which is derived.
                        See Term definition in the corresponding section of this grammar.
                </a:documentation>
                <element name="variable">
                        <attribute name="refvariable">
                                <data type="IDREF"/>
                        </attribute>
                </element>
        </define>

        <define name="NamedSort">
                <a:documentation>
                        A named sort is the concrete definition of a SortDeclaration.
                        It contains a Sort definition.
                </a:documentation>
                <element name="namedsort">
                        <ref name="SortDeclaration.content"/>
                        <ref name="Sort"/>
                </element>
        </define>

        <define name="NamedOperator">
                <a:documentation>
                        A named operator is the concrete definition of and OperatorDeclaration.
                        User-defined operators typically use this construct.
```

```
                VariableDeclaration is used as the construct for its parameters
                (ordered), and Term for its body definition.
        </a:documentation>
        <element name="namedoperator">
                <ref name="OperatorDeclaration.content"/>
                <element name="parameter">
                        <zeroOrMore>
                                <ref name="VariableDeclaration"/>
                        </zeroOrMore>
                </element>
                <element name="def">
                        <ref name="Term"/>
                </element>
        </element>
</define>

<define name="Term.content">
        <empty/>
</define>

<define name="Term">
        <a:documentation>
                A Term involves operators and variables. It refers to a Sort,
                which is usually derived
        </a:documentation>
        <!-- Derived Attribute sort is not externalized (made transient)
        in the PNML file -->
        <choice>
                <ref name="Variable"/>
                <ref name="Operator"/>
        </choice>
</define>

<define name="Sort.content">
        <empty/>
</define>

<!-- Generic definition for sorts and related classes -->
<define name="Sort">
        <a:documentation>
                A sort is not specified. Its is extended by common or high-level
                specific sorts.
                A sort may be a basis for a MultisetSort.
                Actually we don´t need to export the "multi" attribute, since it is most
                important to have the relation from the opposite direction, i.e., from
                MultiSet. This attribute can thus be made transient (not exported) in
                the PNML tool.
                Sorts don´t seem to need an ID since their definition is always included
                as sub-element of Declaration.
        </a:documentation>
        <choice>
                <ref name="BuiltInSort"/>
                <ref name="MultisetSort"/>
                <ref name="ProductSort"/>
                <ref name="UserSort"/>
        </choice>
</define>
<!-- NB: we could have defined BuiltInSort.content which would basically extend
Sort.content, but it is not necessary here since Sort.content is empty -->

<define name="BuiltInSort">
```

```
                <a:documentation>
                        Base definition of a built-in Sort.
                        Further definitions should refine this one using choice
                        as the value of combine attribute.
                </a:documentation>
                <empty/>
        </define>

        <define name="MultisetSort">
                <a:documentation>
                        A multiset sort is built upon a basis sort.
                        There is an issue regarding infinite recursion induced
                        by the fact that a MultisetSort is a Sort.
                        We then have the possibility to define a multiset sort of
                        multiset sorts.
                        In part 1, set of multiset is allowed, which seems akin to
                        this definition.
                        So beware of infinite recursion when using this construct.
                </a:documentation>
                <element name="multisetsort">
                        <ref name="Sort"/>
                </element>
        </define>

        <define name="ProductSort">
                <a:documentation>
                        A product sort is a sort. It refers to an ordered list
                        of members which are sorts.
                </a:documentation>
                <element name="productsort">
                        <zeroOrMore>
                                <ref name="Sort"/>
                        </zeroOrMore>
                </element>
        </define>

        <define name="UserSort">
                <a:documentation>
                        A user sort is used as an abbreviation of
                        existing users-declared sort. It thus refers to a SortDeclaration.
                        Recursion is forbidden.
                </a:documentation>
                <element name="usersort">
                        <attribute name="declaration">
                                <data type="IDREF"/>
                        </attribute>
                </element>
        </define>

        <define name="Operator.content">
                <a:documentation>
                        The sub-terms of an operator.
                </a:documentation>
                <zeroOrMore>
                        <element name="subterm">
                                <ref name="Term"/>
                        </element>
                </zeroOrMore>
        </define>

        <!-- Definition of operators and related classes -->
```

```
<define name="Operator" combine="choice">
        <a:documentation>
                An operator has an ordered list of inputs and an output,
                which are derived. They are thus not exported in the XML
                (transient for the tools).
                All refer to sorts of the arguments over which the operator is applied.
                It is applied on an ordered set of subterms which are either variables
                or application of operator on other subterms.
                The operator is either built-in, or user-declared from built-in.
        </a:documentation>
        <choice>
                <ref name="BuiltInOperator"/>
                <ref name="BuiltInConstant"/>
                <ref name="MultisetOperator"/>
                <ref name="Tuple"/>
                <ref name="UserOperator"/>
        </choice>
</define>

<define name="BuiltInOperator.content">
        <a:documentation>
                The content of BuiltInOperator.content is the one of Operator.content
        </a:documentation>
        <ref name="Operator.content"/>
</define>

<define name="BuiltInOperator">
        <a:documentation>
                Base definition of a built-in operator.
                Further definitions should refine this one using choice
                as the value of combine attribute.
        </a:documentation>
        <empty/>
</define>

<define name="BuiltInConstant.content">
        <a:documentation>
                The content of BuiltInConstant.content is the one of Operator.content
        </a:documentation>
        <ref name="Operator.content"/>
</define>

<define name="BuiltInConstant" combine="choice">
        <a:documentation>
                Base definitin of a built-in constant.
                Further definitions should refine this one using choice
                as the value of combine attribute.
        </a:documentation>
        <empty/>
</define>

<define name="MultisetOperator.content">
        <a:documentation>
                The content of MultisetOperator.content is the one of Operator.content
        </a:documentation>
        <ref name="Operator.content"/>
</define>

<define name="MultisetOperator" combine="choice">
        <a:documentation>
                The Multiset Operator base definition.
```

```
                        Further definitions should refine this one using choice
                        as the value of combine attribute.
                </a:documentation>
                <empty/>
        </define>

        <define name="Tuple">
                <a:documentation>
                        The 'Tuple' construct.
                </a:documentation>
                <element name="tuple">
                        <ref name="Operator.content"/>
                </element>
        </define>

        <define name="UserOperator">
                <a:documentation>
                        A user operator is used as an abbreviation of existing
                        user-declared operators.
                        It thus refers to an OperatorDeclaration.
                        Recursion is forbidden.
                </a:documentation>
                <element name="useroperator">
                        <attribute name="declaration">
                                <data type="IDREF"/>
                        </attribute>
                        <ref name="Operator.content"/>
                </element>
        </define>

</grammar>
```

Then, the grammar for core structure of high-level nets is defined in the file **hlcorestructure.rng**.

```
<?xml version="1.0" encoding="UTF-8"?>

<grammar xmlns="http://relaxng.org/ns/structure/1.0"
        xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
        datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">

        <a:documentation>
                RELAX NG implementation of High-level Petri nets Core Structure grammar.
                This schema implements the core structure shared by High-level Petri nets types.

                File name: hlcorestructure.rng
                Version: 2009
                (c) 2007-2009
                Lom Hillah (AFNOR)
                Revision:
                July 2008 - L.H
        </a:documentation>

        <!-- Definition of additional labels for high-level nets -->

        <define name="net.labels" combine="interleave">
                <a:documentation>
                        A high-level net may have a Declaration.
                </a:documentation>
                <zeroOrMore><ref name="HLDeclaration"/></zeroOrMore>
        </define>
```

```
<define name="page.labels" combine="interleave">
        <a:documentation>
                A page of a high-level net may also have a Declaration.
        </a:documentation>
        <zeroOrMore><ref name="HLDeclaration"/></zeroOrMore>
</define>

<define name="place.labels" combine="interleave">
        <a:documentation>
                A place of a high-level net may have a Type and an HLMarking.
        </a:documentation>
        <interleave>
                <optional><ref name="Type"/></optional>
                <optional><ref name="HLMarking"/></optional>
        </interleave>
</define>

<define name="transition.labels" combine="interleave">
        <a:documentation>
                A transition of a high-level net may have a Condition.
        </a:documentation>
        <optional><ref name="Condition"/></optional>
</define>

<define name="arc.labels" combine="interleave">
        <a:documentation>
                An arc of a high-level net may have a high-level inscription.
        </a:documentation>
        <optional><ref name="HLAnnotation"/></optional>
</define>

<!-- Complex labels definition for high-level nets. -->

<define name="HLDeclaration">
        <a:documentation>
                The 'Declaration' label definition for a net node or a page.
        </a:documentation>
        <element name="declaration">
          <interleave>
                <ref name="simpletextlabel.content"/>
                <optional>
                        <element name="structure">
                                <ref name="Declarations"/>
                        </element>
                </optional>
          </interleave>
        </element>
</define>

<define name="Type">
        <a:documentation>
                The 'Type' label definition for a place.
        </a:documentation>
        <element name="type">
          <interleave>
                <ref name="simpletextlabel.content"/>
                <optional>
                        <element name="structure">
                                <ref name="Sort"/>
                        </element>
                </optional>
```

```
                </interleave>
            </element>
        </define>

        <define name="HLMarking">
            <a:documentation>
                The 'high-level initial marking' label definition for a place.
                The same reasoning applies as for the definitions above.
            </a:documentation>
            <element name="hlinitialMarking">
              <interleave>
                <ref name="simpletextlabel.content"/>
                <optional>
                    <element name="structure">
                        <ref name="Term"/>
                    </element>
                </optional>
              </interleave>
            </element>
        </define>

        <define name="Condition">
            <a:documentation>
                The 'Condition' label definition expressing the guard of a transition.
                The same reasoning applies as for the above definitions.
            </a:documentation>
            <element name="condition">
              <interleave>
                <ref name="simpletextlabel.content"/>
                <optional>
                    <element name="structure">
                        <ref name="Term"/>
                    </element>
                </optional>
              </interleave>
            </element>
        </define>

        <define name="HLAnnotation">
            <a:documentation>
                The 'HLAnnotation' label definition for an arc.
            </a:documentation>
            <element name="hlinscription">
              <interleave>
                <ref name="simpletextlabel.content"/>
                <optional>
                    <element name="structure">
                        <ref name="Term"/>
                    </element>
                </optional>
              </interleave>
            </element>
        </define>

</grammar>
```


Further, built-in sorts definitions are provided in the following.

B.2.2 Dots

Definition of Dots in the file **dots.rng**.

```
<?xml version="1.0" encoding="UTF-8"?>

<grammar xmlns="http://relaxng.org/ns/structure/1.0"
        xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
        datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">

        <a:documentation>
                RELAX NG implementation of Dots grammar.
                Dots are part of high-level common sorts.
                They are used by PT-Nets defined as a restriction of HLPNGs.
                They define what is commonly understood as "non-colored annotation" in
                High-level nets.

                File name: dots.rng
                Version: 2009
                (c) 2007-2009
                Lom Hillah (AFNOR)
                Revision:
                July 2008 - L.H
        </a:documentation>

        <define name="BuiltInSort" combine="choice">
                <a:documentation>
                        Dot is a built-in sort.
                </a:documentation>
                <ref name="Dot"/>
        </define>

        <define name="BuiltInConstant" combine="choice">
                <a:documentation>
                        DotConstant is a built-in constant.
                </a:documentation>
                <ref name="DotConstant"/>
        </define>

        <define name="Dot">
                <a:documentation>
                        Dot is a built-in sort.
                </a:documentation>
                <element name="dot">
                        <empty/>
                </element>
        </define>

        <define name="DotConstant">
                <a:documentation>
                        It is a built-in constant for Dot.
                </a:documentation>
                <element name="dotconstant">
                        <empty/>
                </element>
        </define>

</grammar>
```

B.2.3 Multisets

Definition of Multisets in the file **multisets.rng**.

```
<?xml version="1.0" encoding="UTF-8"?>

<grammar xmlns="http://relaxng.org/ns/structure/1.0"
        xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
        datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">

        <a:documentation>
                RELAX NG implementation of Multisets grammar.
                This schema implements the multiset constructs package for High-level Petri Nets.

                File name: multisets.rng
                Version: 2009
                (c) 2007-2009
                Lom Hillah (AFNOR)
                Revision:
                July 2008 - L.H
        </a:documentation>

        <define name="BuiltInOperator" combine="choice">
                <a:documentation>
                        Cardinality, CardinalityOf, Contains are built-in operators
                </a:documentation>
                <choice>
                        <ref name="Cardinality"/>
                        <ref name="CardinalityOf"/>
                        <ref name="Contains"/>
                </choice>
        </define>

        <define name="MultisetOperator" combine="choice">
                <a:documentation>
                        The concrete Multiset Operators.
                </a:documentation>
                <choice>
                        <ref name="Add"/>
                        <ref name="All"/>
                        <ref name="NumberOf"/>
                        <ref name="Subtract"/>
                        <ref name="ScalarProduct"/>
                        <ref name="Empty"/>
                </choice>
        </define>

        <define name="Add">
                <a:documentation>
                        Defines the 'addition' of multisets.
                </a:documentation>
                <element name="add">
                        <ref name="MultisetOperator.content"/>
                </element>
        </define>

        <define name="Subtract">
                <a:documentation>
                        Defines the 'subtraction' of two multisets.
                </a:documentation>
                <element name="subtract">
```



```
                <ref name="MultisetOperator.content"/>
        </element>
</define>

<define name="All">
        <a:documentation>
                Defines the 'broadcast' operator over a multiset.
        </a:documentation>
        <element name="all">
                <ref name="MultisetOperator.content"/>
                <ref name="Sort"/>
        </element>
</define>

<define name="Empty">
        <a:documentation>
                Defines the 'empty' multiset.
        </a:documentation>
        <element name="empty">
                <ref name="MultisetOperator.content"/>
                <ref name="Sort"/>
        </element>
</define>

<define name="ScalarProduct">
        <a:documentation>
                Defines the 'scalar product' of multisets.
        </a:documentation>
        <element name="scalarproduct">
                <ref name="MultisetOperator.content"/>
        </element>
</define>

<define name="NumberOf">
        <a:documentation>
                Defines the construction of a multiset with a natural number
                and an element of a sort.
        </a:documentation>
        <element name="numberof">
                <ref name="MultisetOperator.content"/>
        </element>
</define>

<define name="Cardinality">
        <a:documentation>
                Defines the cardinality of a multiset.
        </a:documentation>
        <element name="cardinality">
                <ref name="BuiltInOperator.content"/>
        </element>
</define>

<define name="CardinalityOf">
        <a:documentation>
                Defines the cardinality of an element in a multiset.
        </a:documentation>
        <element name="cardinalityof">
                <ref name="BuiltInOperator.content"/>
        </element>
</define>
```

```
        <define name="Contains">
                <a:documentation>
                        Defines the containment relationship between two multisets.
                </a:documentation>
                <element name="contains">
                        <ref name="BuiltInOperator.content"/>
                </element>
        </define>

</grammar>
```

B.2.4 Booleans

Definition of Booleans in the file **booleans.rng**.

```
<?xml version="1.0" encoding="UTF-8"?>

<grammar xmlns="http://relaxng.org/ns/structure/1.0"
        xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
        datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">

        <a:documentation>
                RELAX NG implementation of Booleans grammar.
                Booleans are part of the high-level common sorts.
                They define the bool sort and related operators over
                elements of that sort.

                File name: booleans.rng
                Version: 2009
                (c) 2007-2009
                Lom Hillah (AFNOR)
                Revision:
                July 2008 - L.H
        </a:documentation>

        <define name="Operator" combine="choice">
                <a:documentation>
                        Equality and Inequality are operators.
                </a:documentation>
                <choice>
                        <ref name="Equality"/>
                        <ref name="Inequality"/>
                </choice>
        </define>

        <define name="BuiltInSort" combine="choice">
                <a:documentation>
                        Bool is a built-in sort.
                </a:documentation>
                <ref name="Bool"/>
        </define>

        <define name="BuiltInOperator" combine="choice">
                <a:documentation>
                        BooleanOperator is a built-in operator.
                </a:documentation>
                <ref name="BooleanOperator"/>
        </define>
```



```
<define name="BuiltInConstant" combine="choice">
        <a:documentation>
                BooleanConstant is a built-in constant.
        </a:documentation>
        <ref name="BooleanConstant"/>
</define>

<define name="Bool">
        <a:documentation>
                A Bool is a built-in sort.
        </a:documentation>
        <element name="bool">
                <empty/>
        </element>
</define>

<define name="BooleanOperator.content">
        <a:documentation>
                The content of BooleanOperator.content is the one of BuiltInOperator.content
        </a:documentation>
        <ref name="BuiltInOperator.content"/>
</define>

<define name="BooleanOperator">
        <a:documentation>
                A Boolean operator is a built-in operator.
                Its defines known concrete operators.
        </a:documentation>
        <choice>
                <ref name="And"/>
                <ref name="Or"/>
                <ref name="Imply"/>
                <ref name="Not"/>
        </choice>
</define>

<!-- Declaration of standard boolean operators -->

<define name="And">
        <a:documentation>
                Defines the boolean operator "and".
        </a:documentation>
        <element name="and">
                <ref name="BooleanOperator.content"/>
        </element>
</define>

<define name="Or">
        <a:documentation>
                Defines the boolean operator "or".
        </a:documentation>
        <element name="or">
                <ref name="BooleanOperator.content"/>
        </element>
</define>

<define name="Not">
        <a:documentation>
                Defines the boolean operator "not".
        </a:documentation>
        <element name="not">
```

```
                <ref name="BooleanOperator.content"/>
            </element>
        </define>

        <define name="Imply">
            <a:documentation>
                Defines the boolean operator "imply".
            </a:documentation>
            <element name="imply">
                <ref name="BooleanOperator.content"/>
            </element>
        </define>

        <define name="Equality">
            <a:documentation>
                Defines the boolean operator "equality" which may not necessary
                have booleans as input Sorts.
            </a:documentation>
            <element name="equality">
                <ref name="Operator.content"/>
            </element>
        </define>

        <define name="Inequality">
            <a:documentation>
                Defines the boolean function "inequality" which may not necessary
                have booleans as input Sorts.
            </a:documentation>
            <element name="inequality">
                <ref name="Operator.content"/>
            </element>
        </define>

        <define name="BooleanConstant">
            <a:documentation>
                Declaration of the "true" constant.
            </a:documentation>
            <element name="booleanconstant">
                <attribute name="value">
                    <data type="boolean"/>
                </attribute>
                <ref name="BuiltInConstant.content"/>
            </element>
        </define>

</grammar>
```


B.2.5 Finite Enumerations

Definition of Finite Enumerations in the file **finiteenumerations.rng**.

```
<?xml version="1.0" encoding="UTF-8"?>

<grammar xmlns="http://relaxng.org/ns/structure/1.0"
        xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
        datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">

        <a:documentation>
            RELAX NG implementation of Finite Enumerations grammar.
```



```
                Finite enumerations are part of high-level common sorts.
                They are used by both HLPNGs and Symmetric Nets.
                They define any finite enumeration sort and related operators
                over elements of that sort.

                File name: finiteenumerations.rng
                Version: 2009
                (c) 2007-2009
                Lom Hillah (AFNOR)
                Revision:
                July 2008 - L.H
        </a:documentation>

        <define name="OperatorDeclaration" combine="choice">
                <a:documentation>
                        An Operator declaration is a user-declared operator using built-in.
                        constructs.
                </a:documentation>
                <ref name="FEConstant"/>
        </define>

        <define name="BuiltInSort" combine="choice">
                <a:documentation>
                        FiniteEnumeration is a built-in sort.
                </a:documentation>
                <ref name="FiniteEnumeration"/>
        </define>

        <define name="FiniteEnumeration.content">
                <a:documentation>
                        The content of a Finite Enumeration is composed of constants.
                </a:documentation>
                <zeroOrMore>
                        <ref name="FEConstant"/>
                </zeroOrMore>
        </define>

        <define name="FiniteEnumeration">
                <a:documentation>
                        A Finite Enumeration is a built-in sort.
                </a:documentation>
                <element name="finiteenumeration">
                        <ref name="FiniteEnumeration.content"/>
                </element>
        </define>

        <define name="FEConstant">
                <a:documentation>
                        It is a built-in constant. It is a finite enumeration element.
                </a:documentation>
                <element name="feconstant">
                        <ref name="OperatorDeclaration.content"/>
                </element>
        </define>

</grammar>
```

B.2.6 Cyclic Enumerations

Definition of Cyclic Enumerations in the file **cyclicenumerations.rng**.

```
<?xml version="1.0" encoding="UTF-8"?>

<grammar xmlns="http://relaxng.org/ns/structure/1.0"
        xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
        datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">

        <a:documentation>
                RELAX NG implementation of Cyclic Enumerations grammar.
                Cyclic enumerations are part of high-level common sorts.
                They define any cyclic enumeration sort and related operators
                over elements of that sort.

                File name: cyclicenumerations.rng
                Version: 2009
                (c) 2007-2009
                Lom Hillah (AFNOR)
                Revision:
                July 2008 - L.H
        </a:documentation>

        <define name="BuiltInSort" combine="choice">
                <a:documentation>
                        CyclicEnumeration is a built-in sort.
                </a:documentation>
                <ref name="CyclicEnumeration"/>
        </define>

        <define name="BuiltInOperator" combine="choice">
                <a:documentation>
                        CyclicEnumOperator is a built-in operator.
                </a:documentation>
                <ref name="CyclicEnumOperator"/>
        </define>

        <define name="CyclicEnumeration">
                <a:documentation>
                        A Cyclic Enumeration is a Finite Enumeration.
                </a:documentation>
                <element name="cyclicenumeration">
                        <ref name="FiniteEnumeration.content"/>
                </element>
        </define>

        <define name="CyclicEnumOperator.content">
                <a:documentation>
                        A finite enumeration operator is a built-in operator.
                </a:documentation>
                <ref name="BuiltInOperator.content"/>
        </define>

        <define name="CyclicEnumOperator">
                <a:documentation>
                        A finite enumeration operator defines known concrete operators.
                </a:documentation>
                <choice>
                        <ref name="Successor"/>
                        <ref name="Predecessor"/>
```



```
                </choice>
        </define>

        <!-- Declaration of standard cyclic enumeration operators -->

        <define name="Successor">
                <a:documentation>
                        Defines the 'successor' operator.
                </a:documentation>
                <element name="successor">
                        <ref name="CyclicEnumOperator.content"/>
                </element>
        </define>

        <define name="Predecessor">
                <a:documentation>
                        Defines the 'predecessor' operator.
                </a:documentation>
                <element name="predecessor">
                        <ref name="CyclicEnumOperator.content"/>
                </element>
        </define>

</grammar>
```

B.2.7 Finite Integer Ranges

Definition of Finite Integer Ranges in the file **finiteintranges.rng**.

```
<?xml version="1.0" encoding="UTF-8"?>

<grammar xmlns="http://relaxng.org/ns/structure/1.0"
        xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
        datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">

        <a:documentation>
                RELAX NG implementation of Finite Integer Ranges grammar.
                Finite Integer Ranges are part of the high-level common sorts.
                They define any finite integer range and related operators
                over elements of that sort.

                File name: finiteintranges.rng
                Version: 2009
                (c) 2007-2009
                Lom Hillah (AFNOR)
                Revision:
                July 2008 - L.H
        </a:documentation>

        <define name="BuiltInSort" combine="choice">
                <a:documentation>
                        FiniteIntRange is a built-in sort.
                </a:documentation>
                <ref name="FiniteIntRange"/>
        </define>

        <define name="BuiltInOperator" combine="choice">
                <a:documentation>
                        FiniteIntRangeOperator is a built-in operator.
```

```
        </a:documentation>
        <ref name="FiniteIntRangeOperator"/>
</define>

<define name="BuiltInConstant" combine="choice">
        <a:documentation>
                FiniteIntRangeConstant is a built-in constant.
        </a:documentation>
        <ref name="FiniteIntRangeConstant"/>
</define>

<define name="FiniteIntRange">
        <a:documentation>
                A FiniteIntRange is a built-in sort.
        </a:documentation>
        <element name="finiteintrange">
                <attribute name="start">
                        <data type="integer"/>
                </attribute>
                <attribute name="end">
                        <data type="integer"/>
                </attribute>
        </element>
</define>

<define name="FiniteIntRangeOperator.content">
        <a:documentation>
                The content of FiniteIntRangeOperator is the one of BuiltInOperator.
        </a:documentation>
        <ref name="BuiltInOperator.content"/>
</define>

<define name="FiniteIntRangeOperator">
        <a:documentation>
                It is a built-in operator. It defines known concrete operators.
        </a:documentation>
        <choice>
                <ref name="FIRLessThan"/>
                <ref name="FIRLessThanOrEqual"/>
                <ref name="FIRGreaterThan"/>
                <ref name="FIRGreaterThanOrEqual"/>
        </choice>
</define>

<!-- Declaration of standard finite integer operators -->

<define name="FIRLessThan">
        <a:documentation>
                Defines the 'less than' operator.
        </a:documentation>
        <element name="lessthan">
                <ref name="FiniteIntRangeOperator.content"/>
        </element>
</define>

<define name="FIRLessThanOrEqual">
        <a:documentation>
                Defines the 'less than or equal' operator.
        </a:documentation>
        <element name="lessthanorequal">
                <ref name="FiniteIntRangeOperator.content"/>
```



```
            </element>
        </define>

        <define name="FIRGreaterThan">
            <a:documentation>
                Defines the 'greater than' operator.
            </a:documentation>
            <element name="greaterthan">
                <ref name="FiniteIntRangeOperator.content"/>
            </element>
        </define>

        <define name="FIRGreaterThanOrEqual">
            <a:documentation>
                Defines the 'greater than or equal' operator.
            </a:documentation>
            <element name="greaterthanorequal">
                <ref name="FiniteIntRangeOperator.content"/>
            </element>
        </define>

        <define name="FiniteIntRangeConstant">
            <a:documentation>
                Defines the constant of a declared Finite Integer Range sort.
                It refers to that declared sort.
            </a:documentation>
            <element name="finiteintrangeconstant">
                <attribute name="value">
                    <data type="integer"/>
                </attribute>
                <ref name="FiniteIntRange"/>
                <ref name="BuiltInConstant.content"/>
            </element>
        </define>

</grammar>
```

B.2.8 Partitions

Definition of Partitions in the file **partitions.rng**.

```
<?xml version="1.0" encoding="UTF-8"?>

<grammar xmlns="http://relaxng.org/ns/structure/1.0"
        xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
        datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">

        <a:documentation>
            RELAX NG implementation of Partitions grammar.
            Partitions are part of high-level common sorts.
            They are used by Symmetric Nets.

            File name: partitions.rng
            Version: 2009
            (c) 2007-2009
            Lom Hillah (AFNOR)
            Revision:
            July 2008 - L.H
        </a:documentation>
```

```
    <!-- Declarative part of a Partition, which is found in the signature of a Symmetric Net -->

        <define name="SortDeclaration" combine="choice">
                <a:documentation>
                        A Sort declaration is a user-declared sort, using built-in sorts.
                        It defines known concrete sort declarations.
                </a:documentation>
                <ref name="Partition"/>
        </define>

        <define name="OperatorDeclaration" combine="choice">
                <a:documentation>
                        An Operator declaration is a user-declared operator using built-in.
                        constructs.
                </a:documentation>
                <ref name="PartitionElement"/>
        </define>

        <define name="BuiltInOperator" combine="choice">
                <a:documentation>
                        PartitionOperator is a built-in operator.
                </a:documentation>
                <ref name="PartitionOperator"/>
        </define>

        <define name="Partition">
                <a:documentation>
                        A Partition is a SortDecl.
                        It is defined over a NamedSort which it refers to.
                </a:documentation>
                <element name="partition">
                        <ref name="SortDeclaration.content"/>
                          <interleave>
                                <ref name="Sort"/>
                                <group>
                                        <oneOrMore>
                                                <ref name="PartitionElement"/>
                                        </oneOrMore>
                                 </group>
                                </interleave>
                </element>
        </define>

        <define name="PartitionElement">
                <a:documentation>
                        Defines an element of a Partition.
                </a:documentation>
                <element name="partitionelement">
                        <ref name="OperatorDeclaration.content"/>
                        <oneOrMore>
                                <ref name="Term"/>
                        </oneOrMore>
                </element>
        </define>

    <!-- Operators -->
        <define name="PartitionOperator.content">
                <a:documentation>
                        Its content derives from the one of built-in operator.
                </a:documentation>
```

```
                <ref name="BuiltInOperator.content"/>
        </define>

        <define name="PartitionOperator">
                <a:documentation>
                        It is a built-in operator. It defines known concrete operators.
                </a:documentation>
                <choice>
                        <ref name="PartitionLessThan"/>
                        <ref name="PartitionGreaterThan"/>
                        <ref name="PartitionElementOf"/>
                </choice>
        </define>

        <define name="PartitionLessThan">
                <a:documentation>
                        Defines the 'less than' operator between two partitions.
                </a:documentation>
                <element name="ltp">
                        <ref name="PartitionOperator.content"/>
                </element>
        </define>

        <define name="PartitionGreaterThan">
                <a:documentation>
                        Defines the 'greater than' operator.
                </a:documentation>
                <element name="gtp">
                        <ref name="PartitionOperator.content"/>
                </element>
        </define>

        <define name="PartitionElementOf">
                <a:documentation>
                        Returns the PartitionElement of a finite sort constant.
                </a:documentation>
                <element name="partitionelementof">
                        <attribute name="refpartition">
                                <data type="IDREF"/>
                        </attribute>
                        <ref name="PartitionOperator.content"/>
                </element>
        </define>

</grammar>
```


B.2.9 Integers

Definition of Integers in the file **integers.rng**.

```
<?xml version="1.0" encoding="UTF-8"?>

<grammar xmlns="http://relaxng.org/ns/structure/1.0"
        xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
        datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">

        <a:documentation>
                RELAX NG implementation of Integers grammar.
                Integers are part of high-level specific sorts.
```

```
10              They define the general-purpose integer sort and
                related operators over elements of that sort.

                File name: integers.rng
                Version: 2009
15              (c) 2007-2009
                Lom Hillah (AFNOR)
                Revision:
                June 2008 - L.H
        </a:documentation>
20
        <define name="BuiltInSort" combine="choice">
                <a:documentation>
                        Number is a built-in sort.
                </a:documentation>
25              <ref name="Number"/>
        </define>

        <define name="BuiltInOperator" combine="choice">
                <a:documentation>
30                      IntegerOperator is a built-in operator.
                </a:documentation>
                <ref name="IntegerOperator"/>
        </define>

35      <define name="BuiltInConstant" combine="choice">
                <a:documentation>
                        NumberConstant is a built-in constant.
                </a:documentation>
                <ref name="NumberConstant"/>
40      </define>

        <define name="Number">
                <a:documentation>
                        A number is a built-in sort.
45              </a:documentation>
                <choice>
                        <ref name="Natural"/>
                        <ref name="Positive"/>
                        <ref name="Integer"/>
50              </choice>
        </define>

        <define name="Integer">
                <a:documentation>
55                      An integer type is a number.
                </a:documentation>
                <element name="integer">
                        <empty/>
                </element>
60      </define>

        <define name="Natural">
                <a:documentation>
                        A natural is a number.
65              </a:documentation>
                <element name="natural">
                        <empty/>
                </element>
        </define>
70
```

```
<define name="Positive">
        <a:documentation>
                A positive is a number.
        </a:documentation>
        <element name="positive">
                <empty/>
        </element>
</define>

<define name="IntegerOperator.content">
        <a:documentation>
                Its content derives from the one of built-in operator.
        </a:documentation>
        <ref name="BuiltInOperator.content"/>
</define>

<define name="IntegerOperator">
        <a:documentation>
                It is a built-in operator. It defines known concrete operators.
        </a:documentation>
        <choice>
                <ref name="Addition"/>
                <ref name="Subtraction"/>
                <ref name="Multiplication"/>
                <ref name="Division"/>
                <ref name="Modulo"/>
                <ref name="GreaterThan"/>
                <ref name="GreaterThanOrEqual"/>
                <ref name="LessThan"/>
                <ref name="LessThanOrEqual"/>
        </choice>

</define>

<!-- Declaration of standard finite integer operators -->

<define name="NumberConstant">
        <a:documentation>
                Defines a constant number.
                Must comply with the OCL constraint described in
                Integers package figure.
        </a:documentation>
        <element name="numberconstant">
                <attribute name="value">
                        <data type="integer"/>
                </attribute>
                <interleave>
                        <ref name="BuiltInConstant.content"/>
                        <ref name="Number"/>
                </interleave>
        </element>
</define>

<define name="LessThan">
        <a:documentation>
                Defines the 'less than' operator.
        </a:documentation>
        <element name="lt">
                <ref name="IntegerOperator"/>
        </element>
</define>
```

```
<define name="LessThanOrEqual">
        <a:documentation>
                Defines the 'less than or equal' operator.
        </a:documentation>
        <element name="leq">
                <ref name="IntegerOperator"/>
        </element>
</define>

<define name="GreaterThan">
        <a:documentation>
                Defines the 'greater than' operator.
        </a:documentation>
        <element name="gt">
                <ref name="IntegerOperator"/>
        </element>
</define>

<define name="GreaterThanOrEqual">
        <a:documentation>
                Defines the 'greater than or equal' operator.
        </a:documentation>
        <element name="geq">
                <ref name="IntegerOperator"/>
        </element>
</define>

<define name="Addition">
        <a:documentation>
                Defines the arithmetic 'addition' operator.
        </a:documentation>
        <element name="addition">
                <ref name="IntegerOperator"/>
        </element>
</define>

<define name="Subtraction">
        <a:documentation>
                Defines the arithmetic 'subtraction' operator.
        </a:documentation>
        <element name="subtraction">
                <ref name="IntegerOperator"/>
        </element>
</define>

<define name="Multiplication">
        <a:documentation>
                Defines the arithmetic 'multiplication' operator.
        </a:documentation>
        <element name="mult">
                <ref name="IntegerOperator"/>
        </element>
</define>

<define name="Division">
        <a:documentation>
                Defines the arithmetic 'division' operator.
        </a:documentation>
        <element name="div">
                <ref name="IntegerOperator"/>
```

```
                </element>
        </define>

        <define name="Modulo">
                <a:documentation>
                        Defines the arithmetic 'modulo' operator.
                </a:documentation>
                <element name="mod">
                        <ref name="IntegerOperator"/>
                </element>
        </define>

</grammar>
```

B.2.10 Strings

Definition of Strings in the file **strings.rng**.

```
<?xml version="1.0" encoding="UTF-8"?>

<grammar xmlns="http://relaxng.org/ns/structure/1.0"
        xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
        datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">

        <a:documentation>
                RELAX NG implementation of Strings grammar.
                Strings are part of high-level specific sorts.
                They define the general-purpose string type and
                related operators over elements of that type.

                File name: strings.rng
                Version: 2009
                (c) 2007-2009
                Lom Hillah (AFNOR)
                Revision:
                July 2008 - L.H
        </a:documentation>

        <define name="BuiltInSort" combine="choice">
                <a:documentation>
                        String is a built-in sort.
                </a:documentation>
                <ref name="String"/>
        </define>

        <define name="BuiltInOperator" combine="choice">
                <a:documentation>
                        StringOperator is a built-in operator.
                </a:documentation>
                <ref name="StringOperator"/>
        </define>

        <define name="BuiltInConstant" combine="choice">
                <a:documentation>
                        StringConstant is a built-in constant.
                </a:documentation>
                <ref name="StringConstant"/>
        </define>
```

```
<define name="String">
        <a:documentation>
                A String type is a built-in sort.
        </a:documentation>
        <element name="string">
                <empty/>
        </element>
</define>

<define name="StringConstant.content">
        <a:documentation>
                This definition describes the contents strings.
        </a:documentation>
        <element name="value">
                <text/>
        </element>
</define>

<define name="StringConstant">
        <a:documentation>
                Defines the constant string which may appear in any label
                 of a High-level Petri net object, except in the signature.
        </a:documentation>
        <element name="stringconstant">
                <interleave>
                        <ref name="StringConstant.content"/>
                        <ref name="BuiltInConstant.content"/>
                </interleave>
        </element>
</define>

<!-- Declaration of standard string operators -->

<define name="StringOperator.content">
        <a:documentation>
                Its content derives from the one of BuiltInOperator
        </a:documentation>
        <ref name="BuiltInOperator.content"/>
</define>

<define name="StringOperator">
        <a:documentation>
                It is a built-in operator.
        </a:documentation>
        <choice>
                <ref name="StringLessThan"/>
                <ref name="StringLessThanOrEqual"/>
                <ref name="StringGreaterThan"/>
                <ref name="StringGreaterThanOrEqual"/>
                <ref name="StringConcatenation"/>
                <ref name="StringAppend"/>
                <ref name="StringLength"/>
                <ref name="Substring"/>
        </choice>
</define>

<define name="StringLessThan">
        <a:documentation>
                Defines the 'less than' operator between two strings.
        </a:documentation>
        <element name="lts">
```

```
                <ref name="StringOperator.content"/>
        </element>
</define>

<define name="StringLessThanOrEqual">
        <a:documentation>
                Defines the 'less than or equal' operator.
        </a:documentation>
        <element name="leqs">
                <ref name="StringOperator.content"/>
        </element>
</define>

<define name="StringGreaterThan">
        <a:documentation>
                Defines the 'greater than' operator.
        </a:documentation>
        <element name="gts">
                <ref name="StringOperator.content"/>
        </element>
</define>

<define name="StringGreaterThanOrEqual">
        <a:documentation>
                Defines the 'greater than or equal' operator.
        </a:documentation>
        <element name="geqs">
                <ref name="StringOperator.content"/>
        </element>
</define>

<define name="StringConcatenation">
        <a:documentation>
                Defines the 'concatenation' of two strings.
        </a:documentation>
        <element name="stringconcatenation">
                <ref name="StringOperator.content"/>
        </element>
</define>

<define name="StringLength">
        <a:documentation>
                Defines the 'length' of a string.
        </a:documentation>
        <element name="stringlength">
                <ref name="StringOperator.content"/>
        </element>
</define>

<define name="StringAppend">
        <a:documentation>
                Defines the 'append' operator between an element and a string.
        </a:documentation>
        <element name="stringappend">
                <ref name="StringOperator.content"/>
        </element>
</define>

<define name="Substring">
        <a:documentation>
                With this operator, specified substrings
```

```
                        can be extracted from larger strings.
                </a:documentation>
                <element name="substring">
                        <attribute name="start">
                                <data type="nonNegativeInteger"/>
                        </attribute>
                        <attribute name="length">
                                <data type="nonNegativeInteger"/>
                        </attribute>
                        <ref name="StringOperator.content"/>
                </element>
        </define>

</grammar>
```

B.2.11 Lists

Definition of Lists in the file **lists.rng**.

```
<?xml version="1.0" encoding="UTF-8"?>

<grammar xmlns="http://relaxng.org/ns/structure/1.0"
        xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
        datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">

        <a:documentation>
                RELAX NG implementation of Lists grammar.
                The list signature is part of high-level specific sorts.
                It defines the general-purpose list sort and related operators
                over elements of that sort.

                File name: lists.rng
                Version: 2009
                (c) 2007-2009
                Lom Hillah (AFNOR)
                Revision:
                July 2008 - L.H
        </a:documentation>

        <define name="BuiltInSort" combine="choice">
                <a:documentation>
                        List is a built-in sort.
                </a:documentation>
                <ref name="List"/>
        </define>


        <define name="BuiltInOperator" combine="choice">
                <a:documentation>
                        ListOperator is a built-in operator.
                </a:documentation>
                <ref name="ListOperator"/>
        </define>

        <define name="List">
                <a:documentation>
                        A List is a built-in sort.
                        It is a set over a basis sort.
                </a:documentation>
```



```
        <element name="list">
                <ref name="Sort"/>
        </element>
</define>

<define name="BuiltInConstant" combine="choice">
        <a:documentation>
                EmptyList is a built-in constant.
        </a:documentation>
        <ref name="EmptyList"/>
</define>

<define name="ListOperator.content">
        <a:documentation>
                It is derived from BuiltInOperator.content
        </a:documentation>
        <ref name="BuiltInOperator.content"/>
</define>

<define name="ListOperator">
        <a:documentation>
                It is a built-in operator. It defines known concrete operators.
        </a:documentation>
        <choice>
                <ref name="ListAppend"/>
                <ref name="ListConcatenation"/>
                <ref name="MakeList"/>
                <ref name="ListLength"/>
                <ref name="MemberAtIndex"/>
                <ref name="Sublist"/>
        </choice>
</define>

<!-- Declaration of standard list operators -->

<define name="EmptyList">
        <a:documentation>
                This operator defines an empty list which is a built-in constant.
        </a:documentation>
        <element name="emptylist">
                <interleave>
                        <ref name="Sort"/>
                        <ref name="BuiltInConstant.content"/>
                </interleave>
        </element>
</define>

<define name="ListLength">
        <a:documentation>
                Defines the 'length' of a list.
        </a:documentation>
        <element name="listlength">
                <ref name="ListOperator.content"/>
        </element>
</define>

<define name="MakeList">
        <a:documentation>
                This operators creates a new list.
        </a:documentation>
        <element name="makelist">
```

```
                        <interleave>
                                <ref name="Sort"/>
                                <ref name="ListOperator.content"/>
                        </interleave>
                </element>
        </define>

        <define name="ListConcatenation">
                <a:documentation>
                        Defines the 'concatenation' of two lists.
                </a:documentation>
                <element name="listconcatenation">
                        <ref name="ListOperator.content"/>
                </element>
        </define>

        <define name="ListAppend">
                <a:documentation>
                        Defines the 'append' operation of an element to a list.
                </a:documentation>
                <element name="listappend">
                        <ref name="ListOperator.content"/>
                </element>
        </define>

        <define name="MemberAtIndex">
                <a:documentation>
                        At which index is an element in a List ?
                </a:documentation>
                <element name="memberatindex">
                        <attribute name="index">
                                <data type="nonNegativeInteger"/>
                        </attribute>
                        <ref name="ListOperator.content"/>
                </element>
        </define>

        <define name="Sublist">
                <a:documentation>
                        With this operator, specified sublists
                        can be extracted from larger lists.
                </a:documentation>
                <element name="sublist">
                        <attribute name="start">
                                <data type="nonNegativeInteger"/>
                        </attribute>
                        <attribute name="length">
                                <data type="nonNegativeInteger"/>
                        </attribute>
                        <ref name="ListOperator.content"/>
                </element>
        </define>

</grammar>
```

B.2.12 Arbitrary Declarations

Definition of Arbitrary Declarations in the file **arbitrarydeclarations.rng**.

```
<?xml version="1.0" encoding="UTF-8"?>
```


