Interoperable Petri Net Models via Ontology

Dragan V. Gašević and Vladan B. Devedžić

GOOD OLD AI Research Group
FON – School of Business Administration
University of Belgrade
POB 52, Jove Ilića 154, 11000 Belgrade, Serbia and Montenegro
Email: dgasevic@acm.org
Email: devedzic@fon.bg.ac.yu

Abstract. The paper presents a Petri net infrastructure that should provide sharing Petri nets on the Semantic Web. Previous solutions only provide model interchange mechanisms between Petri net tools. The Petri net ontology is a central part of our solution. The ontology is closely related to the Petri Net Markup Language (PNML) – an ongoing Petri net community sharing effort. We developed the Petri net ontology using both UML and Protégé tool, whereas we use RDF and OWL to represent the ontology. We implemented a Petri net software tool – P3 – that tool can be used to convert the Petri net ontology compliant models to the formats of current Petri net tools (e.g. DaNAMiCS, Petri Net Kernel, PIPE) using eXtensible Stylesheet Language Transformations (XSLT). In order to show how the ontology can be used we developed a simple educational Web application that uses RDF-annotated ontology-based Petri net learning materials.

Key words: Petri nets, Interoperability, Ontology development, Semantic Web, XSLT, OWL

Reference to this paper should be made as follows: Gašević, D. and Devedžić, V. (2006) 'Interoperable Petri Net Models via Ontology', *International Journal of Web Engineering and Technologies*, Vol. 2, No. 2, pp. ??-??.

Biographical notes: Dragan Gašević received the B.S., M.S. and PhD degrees in informatics and computer engineering from the Department of Computer Science, University of Belgrade, Serbia and Montenegro, in 2000, 2002, and 2004 respectively. He is a researcher at the GOOD OLD AI research group, FON – School of Business Administration, University of Belgrade, Serbia and Montenegro. He is also a lecturer of computer science at the Department of Informatics and Computer Engineering, Military Academy, University of Belgrade, Belgrade, Serbia and Montenegro. His current research interests are in the area of ontology development, Semantic Web, XML-based interoperability, integration between software engineering and knowledge engineering techniques, learning technologies and Petri nets. So far, he has authored/co-authored more than 80 research papers, several book chapters, and one book. He has been a referee and PC member for several international conferences. He can be reached at http://www.sfu.ca/~gasevic.

Vladan Devedžić is an associate professor of computer science at the Department of Information Systems, FON - School of Business Administration, University of Belgrade, Serbia and Montenegro. He has received all of his degrees from The School of Electrical Engineering, University of Belgrade, Serbia and Montenegro (BS, 1982; MS, 1988; PhD, 1993). His main research interests include intelligent systems, knowledge representation, ontologies, Semantic Web, intelligent reasoning, software engineering, and applications of artificial intelligence techniques to education and medicine. So far, he has authored/co-authored more than 220 research papers published in international and national journals and conferences. His major long-term professional goal is a continuous effort to bring close together the ideas from the broad fields of intelligent systems and software engineering. He has developed several practical intelligent systems and tools, and actively participates to several ongoing projects in industry and in academia. He can be reached at http://fon.fon.bg.ac.yu/~devedzic.

1 Introduction

The main idea of this paper is to propose a suitable approach to Petri nets [28] application on the Semantic Web, i.e. an approach that would enable full semantic interoperability of Petri net models. Currently, Petri net interoperability is possible at the syntax level. Syntax level interoperability was firstly introduced by Berthelot et al [4], who pointed out that it would be very useful if Petri net researchers could share their Petri net model descriptions, so that more software tools could analyze the same model. So far, all Petri net interchange attempts have been mainly tool-specific, with very low (or without any) general acceptance. The *Petri Net Markup Language* (PNML) [5] is a recent Petri net community effort that tries to provide XML-based model sharing. PNML tends to be a part of the future ISO/IEC High-level Petri net standard [18] [20]. A particularly important advantage of this approach is that XML documents can be easily transformed using *eXtensible Stylesheet Language Transformations* (XSLT) into other formats (that need not necessarily be XML-based).

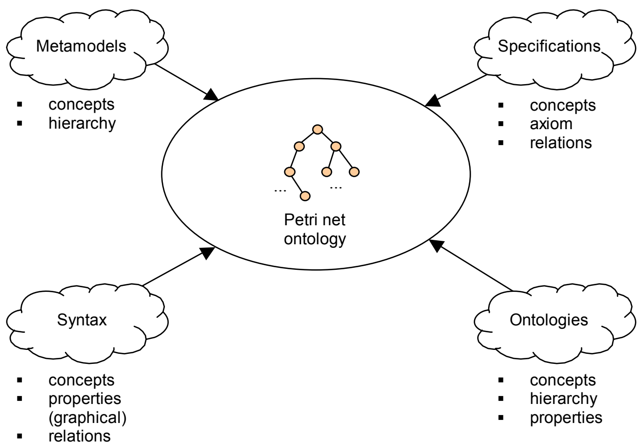
A suitable way to represent Petri nets is needed in order to reuse them more effectively on the Semantic Web. It requires defining the *Petri net ontology* for semantic description of Petri net concepts and their relationships. The Petri net ontology enables describing a Petri net using Semantic Web languages (e.g. RDFS, and OWL) [3] [13]. Petri nets described that way can be inserted into other, non-Petri net XML-based formats, such as *Scalable Vector Graphics* (SVG, the XML-based WWW Consortium (W3C) standard for 2D vector graphics) [19]), which makes possible to reconstruct Petri net models using metadata and annotations according to the Petri net ontology. We defined the Petri net ontology using experience from previous Petri net formal descriptions (metamodel, ontologies, and syntax). They indicate very useful directions for selecting key Petri net concepts and specifying their mutual relations. The PNML is of primary importance here – it is closely related to the Petri net ontology. Actually, it is a medium (syntax) for semantics. We additionally empowered the PNML usability by defining mappings to/from the Semantic Web languages (i.e. RDFS and OWL).

The next section describes main sources for defining Petri net ontology. We concentrate on Petri net syntax because most work has been done in solving this problem (we specifically discuss the PNML). Section three enumerates advantages of the Petri net ontology, and gives guidelines for its construction. Section four outlines development of the Petri net ontology – its initial design and implementation using UML and Protégé [25] (i.e., RDF Schema (RDFS)-based implementation), whereas Section five extends the ontology using an OWL-based UML profile in order to support diversity of Petri net dialects. In Section six, we present the tool we implemented to support the Petri net ontology, as well as an ontology-driven infrastructure for sharing Petri nets using PNML, XSLT, and RDF.

2 Sources for Petri net ontology

This section analyzes present Petri net: specifications, metamodels, ontologies, and syntax. Our main goal is to identify how each of these formal Petri net definitions can contribute to Petri net ontology. In Figure 1 we show relations between Petri net ontology (we are developing) and existing Petri net definitions. Also, this figure shows what we can use from all these sources to define the ontology.

Figure 1 Petri net ontology and elements that can be used from present formal ways for representing Petri nets



Specifications. We assume formal mathematical definitions as well as Petri net standards as Petri net specifications. Currently, there are many Petri net mathematical definitions for different Petri net dialects [23] [28]. On the other hand, there is an initiative to adopt ISO/IEC High-level Petri net standard [18]. We believe that from specifications we can obtain concepts of Petri net domain, axioms, and relations between Petri net concepts.

Metamodels. Some authors believe that metamodel is closely related to ontology. Accordingly, an illustrative and very comprehensive Petri net metamodel is proposed by Breton & Bézivin [7]. Their starting point is that a metamodel defines a set of concepts and relations, i.e. the terminology and a set of additional constraints (assertions). Note that this proposal is very important for development of Petri net tools. However, it does not show how Petri nets can be used on the Semantic Web with non-Petri net tool (i.e. annotation), and hence how Petri nets are mapped into Semantic Web language (e.g. RDF(S)). On the other hand, we can obtain useful guidelines how to develop taxonomy (hierarchy) of Petri net concepts. Defining a Petri net UML profile produces a solution similar to the metamodel-based one, because UML profiles extend the UML metamodel by introducing stereotypes, tagged values and constraints. Hansen proposes a Petri net UML profile [16]. Although this solution is metamodel-based, it is fairly awkward since it is based on the UML metamodel. This means that all UML concepts are introduced in the Petri net metamodel, but most of them are needless for the Petri net semantics. Also, this approach has the same limitation as the previous one in terms of support for Petri net dialects, Semantic Web use, and Petri net structuring mechanisms such as both pages and modules.

Ontologies. So far, only one Petri net ontology has been developed. Perleg and her colleagues developed a Petri net ontology using Protégé and a specific graphical user interface (GUI) that extends the standard GUI of the Protégé tool [27]. Actually, this GUI provides graphical tools for all Petri net concepts (Places, Transitions, and Arcs). In addition, the Petri net ontology is represented in RDFS, and concrete Petri net models are represented in RDF. This solution gives a solid starting point for defining the Petri net ontology. However, it has serious limitations. It covers only Time Petri nets, and no other kinds of Petri nets. It neither defines Petri net structuring mechanisms, nor provides precise constraints (e.g. types of an arc's source and target nodes that can be done using Protégé Axiom Language (PAL) constraints). Finally, it does not enable using other ontology languages for representing the Petri net ontology (e.g. DAML or OWL). This ontology can give us guidelines how to define conceptualization, properties, and taxonomy of the Petri net ontology.

Syntax. A lot of work has been done in defining and using Petri net syntax. We can classify present syntax in categories as follows: general-purpose and tool-specific. Tool-specific syntax are, for example, those that are used in the following tools: DaNAMiCS (regular text syntax) and Renew (XML). *Abstract Petri Net Notation* (APNN) is the first attempt to define a general-purpose Petri net syntax (i.e., it has ability to describe different Petri net dialects) [2]. The abstract notation for each Petri net class is defined in BNF. This grammar is useful from the extensibility and modularity perspectives. The Petri net community is working for three years already on development of the Petri Net Markup Language (PNML) [31] that might become a part of the future ISO/IEC High-level Petri nets standard [18] [20]. The PNML is a proposal that is based on XML. PNML specification is based on the PNML metamodel that formulates the structure of PNML documents. Actually, this metamodel defines the basic Petri net concepts (places, transitions, arcs), as well as their relations that can be presented in a PNML document. PNML, being more mature, is currently supported (or will be supported soon) by many Petri net software tools, for instance: Petri Net Kernel (PNK), CPN Tools, Worflan, PIPE, PEP, VIPtool, etc. For educational purposes, we developed *P3*, a Petri net tool that supports PNML [11] (see Section 6 for details). We believe that Petri net syntax give main contribution for Petri net ontology with: concepts, properties and their relations.

3 The Petri net ontology guidelines

As we have seen so far, Petri net formats use different concepts for defining their syntax. Some of these syntax-based approaches actually have problems with syntax validation. For instance, it is very difficult to validate a text-based document (i.e. DaNAMiCS) without a special-purpose software for checking the corresponding format. A slightly better solution is to use DTD for XML definition as the Renew does. But, DTD does not support inheritance, does not have datatype checking (for the primary semantics checking), does not support defining specific formats, and moreover a DTD document has non-XML structure. The W3C XML Schema overcomes most DTD's problems. However, XML Schema has no full support for describing semantics [21]. In fact, XML Schema is only a way for defining syntax. Furthermore, if we want to share Petri net models not only with Petri net tools on the Semantic Web, we must have a formal way for representing Petri net semantics since we can not expect that a non-Petri net tool performs semantic validation.

We believe that the concept of ontology can be used for formal description of Petri net semantics. In this paper, domain ontology is understood as a formal way for representing shared conceptualization in some domain [14]. Ontology has formal mechanisms to represent concepts, concept properties, and relations between concepts in the domain of discourse. With the Petri net ontology, we can overcome validation problems that we have already noticed. However, the Petri net ontology does not exclude current Petri net formats (especially PNML). Ontology has relations to syntax, in the sense that syntax should enable ontological knowledge sharing [8]. With the Petri net ontology, we can use ontology development tools for validation of Petri net models (e.g. Protégé). Also, having the Petri net ontology one can use Semantic Web languages for representing Petri net models (e.g. RDF, RDF Schema – RDF, DAML+OIL, OWL, etc.) [13]. Thus, we show how PNML can be used as a guideline for the Petri net ontology.

Figure 2 Conceptualization of the Core Petri net ontology: key concepts, their mutual relations, and cardinality

Net
- place (0..*)
- transition (0..*)
- page (0..*)
- place reference (0..*)
- transition reference (0..*)
- module instance (0..*)

Place
- name (1)
- marking (1)
- initial marking (1)
- position (1)

Transition
- name (1)
- position (1)

Arc
- from node (1)
- to node (1)
- position (0..*)
- mulitiplicity (1)

Page
- place (0..*)
- transition (0..*)
- page (0..*)
- place reference (0..*)
- transition reference (0..*)
- module instance (0..*)

Module
- name (1)
- place (0..*)
- transition (0..*)
- arc (0..*)
- page (0..*)
- place reference (0..*)
- transition reference (0..*)
- module instance (0..*)
- module interface (1)

Module instance
- name (1)
- place (0..*)
- transition (0..*)
- arc (0..*)
- page (0..*)
- place reference (0..*)
- transition reference (0..*)
- module instance (0..*)
- module interface (1)

Module interface
- input (0..*)
- output (0..*)

Place reference
- reference (1)
- name (1)
- position (1)

Transition reference
- reference (1)
- name (1)
- position (1)

Graphical information
- position (1)
- font style (1)
- color (1)
- ...

Marking
- token (0..*)
- graphical information (1)

Initial marking
- token (0..*)
- graphical information (1)

Name
- graphical information (1)

Token

–

Multiplicity
- graphical information (1)

Accordingly, we think that Petri net ontology should have common part that contains concepts common for all Petri net dialects. Afterward, this common part will be specialized for concrete Petri net dialect. Actually, this is the same principle that uses PNML [5]. In Figure 2 we show common part of Petri net ontology that we call Core

Petri net ontology. The Core Petri net ontology is extracted from the analyzed ontology sources.

We introduced some concepts that do not really exist in Petri net models in order to obtain more suitable concept hierarchy in the core ontology. We call these concepts synthetic concepts. Overview of these concepts is given in Table 1. The meanings of some of Petri net concepts we refer to in the table (e.g. module, structural element, etc.) we clarify in the next section. In the next section we define the Petri net ontology using UML and Protégé ontology development tool.

Table 1 Overview of the synthetic concepts in the Core Petri net ontology – generalizations of concepts from Figure 2

Synthetic concept	Generalize concepts
Node reference	place reference, transition reference
Node	place, transition, node reference
Structural element	page, module instance
Model element	structural element, arc, node

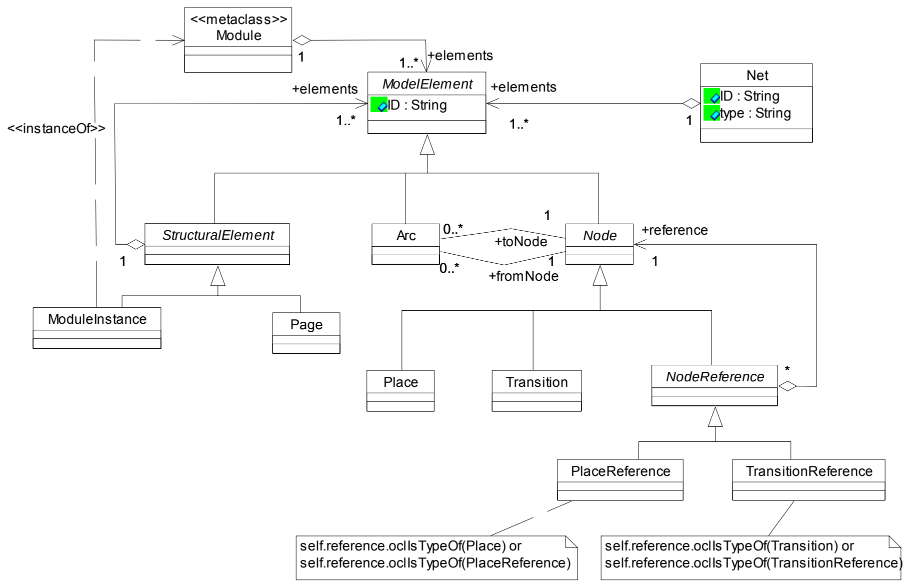
4 The Petri net ontology – initial implementation

In order to develop the Petri net ontology, we decided to use UML [22]. UML was suitable because it is a generally accepted and standardised tool for analysis and modeling in software engineering. We were also able to employ UML-based Petri net descriptions existing within the PNML definition [5]. However, neither UML tools nor the UML itself are intended to be used for ontology development. Thus, in order to achieve more precise Petri net definition than a UML model provides, it is necessary to use an ontology development tool. We decided to use Protégé 2000 [25] since it is a popular tool for ontology development and can import UML models. This is enabled by Protégé's UML backend that imports UML models (represented in *XML Metadata Interchange* (*XMI*) format) into a Protégé ontology.

4.1 The conceptual solution: A UML model

The hierarchy of core concepts of the Petri net ontology is shown in Figure 3. In our design of the Petri net ontology, there is a single root element that we call *ModelElement*. This element is the parent for all elements of Petri net structure. The name of this class is *ModelElement* because the UML metamodel uses the same name for its root class [26]. A Petri net (the *Net* class) can contain many different *ModelElement*s. Since we regards that a *ModelElement* may belong to one and only one Petri net we have unidirectional association from *Net* to *ModelElement*. *ModelElement* and *Net* have the ID attribute (unique identifier) of String type, and *Net* has also an attribute that describes the type of the Petri net. It is in accordance with PNML. The three main Petri net concepts (place, transition, and arc) define the structure of a Petri net, and they are represented in Figure 3 with the corresponding classes (*Place*, *Transition*, *and Arc*). Places and transitions are kinds of nodes (*Node*). In some Petri nets, an arc connects two nodes of different kinds. However, it is important to say that this is not a general Petri net ontology statement, since there are Petri net dialects where an arc can connect, for instance, two transitions [29]. Hence we did not include this statement in the core Petri net ontology, but it should be defined in ontology extensions for different Petri net dialects.

Figure 3 The Petri net ontology – Hierarchy of core Petri net concepts

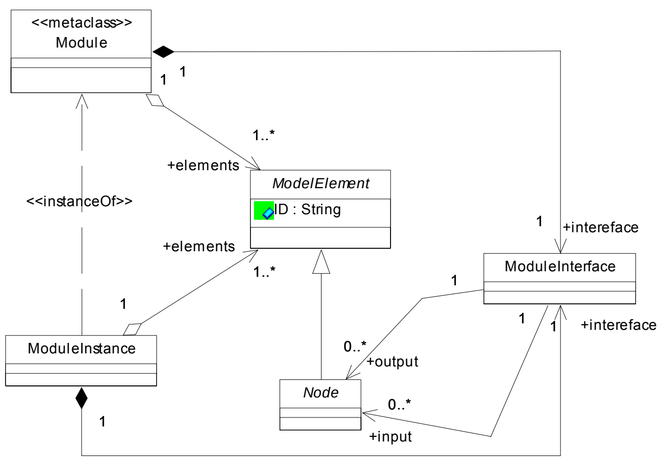


The *Node* class is introduced in the ontology in order to have a common way to reference both places and transitions. In order to make Petri net models easy to maintain, different concepts for structuring can be used. In the Petri net ontology, we have the class *StructuralElement*. This class is inherited from *ModelElement*, and we inherit from this class all classes that represent structuring mechanisms. Also, a *StructuralElement* may contain many *ModelElement*s, but *ModelElements* do not need the information which *StructuralElement* they belong to (if any). Accordingly, we have unidirectional aggregation from *StructuralElement* to *ModelElement*. We have decided to support two common mechanisms: pages (the *Page* class), and modules (the *Module* metaclass). A *Page* may consist of other Petri net *ModelElement*s – it may even consist of other *StructuralElement*s (e.g. *Page* and *ModuleInstance*). A *NodeReference*, which can be either a *TransitionReference* or a *PlaceReference,* represents an appearance of a node. Since it is not important for a *Node* to know what *NodeReference*s refer to it, we have unidirectional aggregation from *NodeReference* to *Node*. Here, there are also constraints: a *TransitionReference* can refer to either a *Transition* or another *TransitionReference*, while a *PlaceReference* can refer to either a *Place* or another *PlaceReference*. The *Decorator* design pattern [10] was used to represent referencing of a *NodeReference*. Of course, the pattern is not a part of the UML language, but it is a design matter. The reason we applied this design pattern is to have a more suitable model of Petri nets for some future implementations of the ontology. We show these constraints using OCL in Figure 3. These constraints also affect the OCL constraint for arcs that we have already described. Unlike the OCL statement for arcs, this statement can be applied on all Petri net dialects (see the OCL constraints in Figure 3).

The second kind of structuring mechanisms are modules. A *Module* consists of *ModelElements* (even other *StructuralElements*, i.e. *Page* and *ModuleInstance*), and it can be instantiated (much like an object is instantiated from a class in the object-oriented paradigm). Accordingly, *Module* is a metaclass (the stereotype in Figure 3), and *ModuleInstance* depends on *Module* (that shows a stereotyped *instanceOf* dependency from *ModuleInstance* to *Module*). Figure 4 describes both *Module* and *ModuleInstance* classes in detail. One can see that these two classes also have one *interface*. Type of this *interface* is the *ModuleInterface* class that has two collections of *ModelElements*: output (a collection of output *Nodes*) and input (a collection of input *Nodes*). Note that these input and output *Nodes* can only be *Places* and *Transitions*, and this is expressed by the following OCL constraint:

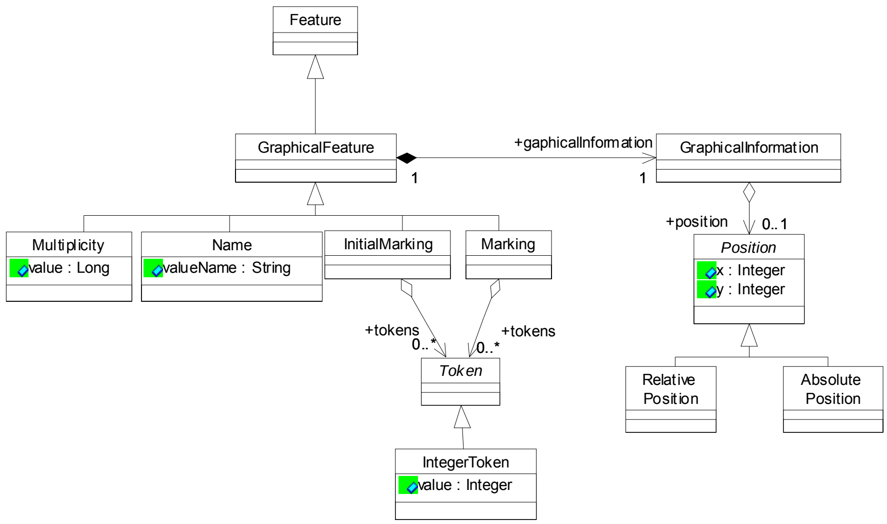
```
context ModuleInterface:
  inv:
     (self.output.oclTypeOf(Place) or self.output.oclTypeOf(Transition)) and
     (self.input.oclTypeOf(Place) or self.input.oclTypeOf(Transition))
```

Figure 4 Both *Module* and *ModuleInstance* classes have interface that consists of *input* and *output Nodes*



In Petri nets, an additional property (or feature) can be attached to almost every core Petri net element (e.g. name, multiplicity, etc.). Thus, we have included in the Petri net ontology a description of features and in Figure 5 we shortly depict how these features have been added. All UML model elements shown in the figure belong to the core Petri net ontology. The root class for all features is *Feature*. This is also similar to the UML metamodel. The Petri net ontology follows the PNML's classification of features: those that contain graphical information (annotation) and those that do not have them (attribute). In the Petri net ontology every feature directly inherited from *Feature* class is an attribute, whereas *GraphicalFeature* class represents annotations. *GraphicalFeature* has graphical information that can consist of, for instance, position (the *Position* class and its children *Absolute Position* and *Relative Position*). Examples of graphical features are: *Multiplicity*, *Name*, *InitialMarking*, and *Marking*. It is interesting to notice that marking and initial marking consist of tokens (the *Token* class). In order to support token colors, the *Token* class is abstract. In Figure 5 we show a case when there are no colors attached to tokens; instead, we just take into account the number of tokens (*IntegerToken*).

Figure 5 The Property hierarchy of the Petri net ontology. The hierarchy is a part of the core Petri net ontology



Attaching a new feature to a Petri net class requires just adding an association between a class and a feature. A UML description is a convenient way for representing the Petri net semantics. Also, this Petri net ontology can be used as a Petri net metamodel in future Petri net implementations that can take advantage of the MDA concept and repository-based software development [6]. However, it does not let us semantically validate Petri net models. For example, we cannot use OCL statements to perform this task. In addition, UML attributes and ontology properties are semantically different concepts. Unlike a UML's attribute, ontology property is a first-class concept that can exist independently of any ontology class [1].

There are two ways to further refine the Petri net ontology. The first one is to use a UML profile [22] for UML-based ontology development. The second one recommends using standard ontology development tools. We decided to use: 1. Protégé 2000, since it provides all the necessary ontology development features (constraints and support for ontology languages), but it also has the ability to use the UML models we have shown; 2. The Ontology UML profile [9] that is based on OWL.

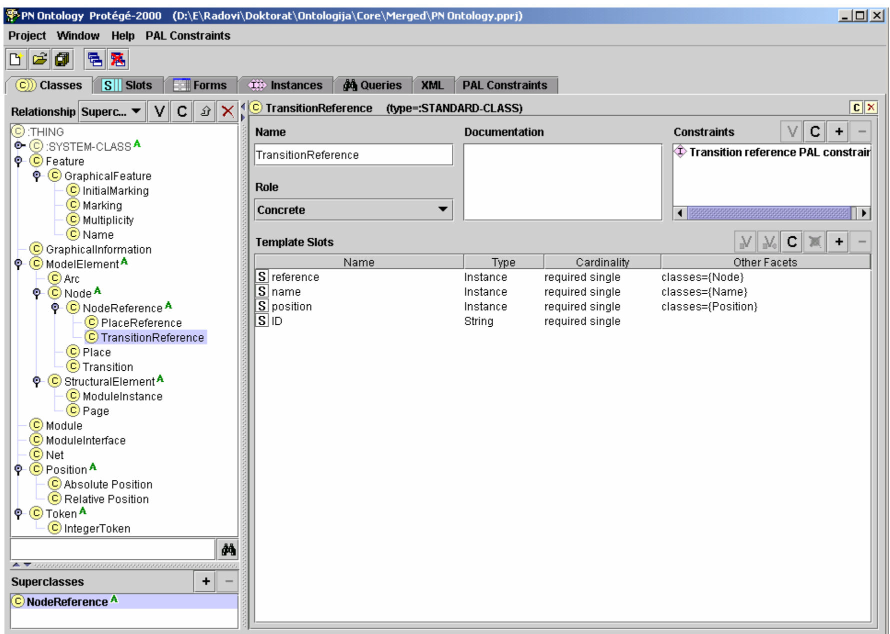
4.2 The Petri net ontology in the Protégé tool

We can precisely define the Petri net ontology in Protégé 2000. We can distinguish between a class and a metaclass (e.g. *Module* – a metaclass, *ModuleInstance* – a class), we can use different Semantic Web languages provided through Protégé's backends (RDF(S), OWL, DAML+OIL) to represent the Petri net ontology, and we can specify the constraints that we defined in the UML model using OCL (e.g. PAL). We can then validate all ontology instances using these constraints, and detect if there is any instance that does not conform to some of the constraints. For instance, here is the PAL constrain on the TransitionReference class, which is equivalent to the OCL constraint on the same classs from Figure 3:

```
(defrange ?transitionRef :FRAME TransitionReference)
(forall ?transitionRef (or
    (instance-of (reference ?transitionRef) Transition)
    (instance-of (reference ?transitionRef) TransitionReference)))
```

After the initial UML design of the Petri net ontology, it was imported into the Protégé using Protégé's UML backend (http://protege.standford.edu/plugin/uml). This plug-in has the ability to read an XML format (i.e., XMI) for representing UML models. The main shortcoming of this UML backend is that it is unable to map UML class associations. Thus we had to add manually all the slots that are represented in UML as association ends. A snapshot of the Petri net ontology after we imported it and inserted all slots (i.e., association ends) in Protégé is shown in Figure 6.

Figure 6 The Petri net ontology in Protégé 2000



Of course, Protégé does not have the ability to transform OCL constraints into PAL constraints. Thus we have also manually reconstructed all OCL-defined constraints from the UML model of the Petri net ontology into a set of corresponding PAL constraints.

Using Protégé we generated the RDFS that describes the Petri net ontology. One can use it for reasoning about a document that contains a Petri net model. Figure 7 shows an excerpt of this RDFS. This figure depicts how RDFS defines the classes for *ModelElement*, *Node*, *Transition*, *Place*, *Arc*, and *ArcType*. Also, this figure shows how RDFS defines *Feature*, as well as how *name* feature is defined and attached to classes that should have this property.

Since Protégé supports more concepts for ontology definition than RDFS does, one can notice some extensions of RDFS in Figure 7. Note that this RDFS/XML syntax generated by Protégé relays on the old RDFS semantics for `rdf:domain` property where the multiple domain of the property is regarded as a union of all domain classes. According to the new RDFS semantics if a property has more than one domain property this says that any resource that has that property is an instance of all of the classes specified as the domains (i.e. intersection of domain classes). These Protégé extensions are manifested by namespace `a`. For example, they are used to define cardinality (`a:maxCardinality`, `a:minCardinality`), to refer to a PAL constraint (`a:slot_constraints`), etc. Of course, this is neither a limitation of the Petri net ontology nor of the Protégé tool, but of RDFS itself. Most of such limitations are overcome in the forthcoming OWL [3], but this discussion is beyond the scope of this paper.

Figure 7 A part of the RDF Schema of the Petri net ontology

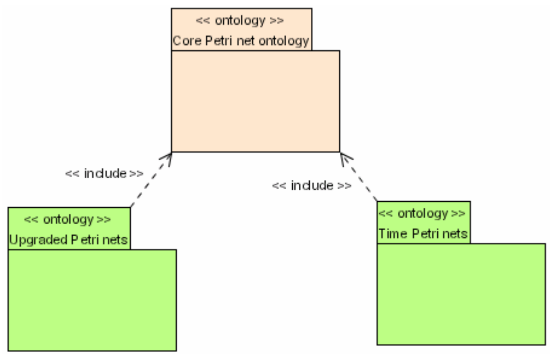
```
<rdf:RDF xmlns:rdf="&rdf;" xmlns:a="&a;" xmlns:rdfs="&rdfs;">
    <!-- … -->
    <rdfs:Class rdf:about="ModelElement" a:role="abstract"
        rdfs:label="ModelElement">
         <rdfs:subClassOf rdf:resource="&rdfs;Resource"/>
    </rdfs:Class>
    <rdfs:Class rdf:about="Node" a:role="abstract" rdfs:label="Node">
         <rdfs:subClassOf rdf:resource="ModelElement"/>
    </rdfs:Class>
    <rdfs:Class rdf:about="Place" rdfs:label="Place">
         <rdfs:subClassOf rdf:resource="Node"/>
    </rdfs:Class>
    <rdfs:Class rdf:about="PlaceReference" rdfs:label="PlaceReference">
         <rdfs:subClassOf rdf:resource="NodeReference"/>
         <a:_slot_constraints rdf:resource="PN Ontology_00023"/>
    </rdfs:Class>
    <rdfs:Class rdf:about="Feature" rdfs:label="Feature">
         <rdfs:subClassOf rdf:resource="&rdfs;Property"/>
    </rdfs:Class>
    <!-- … -->
    <rdf:Property rdf:about="name" a:maxCardinality="1"
         a:minCardinality="1" rdfs:label="name">
         <rdfs:range rdf:resource="Name"/>
         <rdfs:domain rdf:resource="Node"/>
         <rdfs:domain rdf:resource="Place"/>
         <rdfs:domain rdf:resource="PlaceReference"/>
         <rdfs:domain rdf:resource="Transition"/>
         <rdfs:domain rdf:resource="TransitionReference"/>
    </rdf:Property>
    <!-- … -->
</rdf:RDF>
```

On the other hand, one can see that the RDFS/Protégé Petri net ontology does not take into account Petri net dialects. In this version of the Petri net ontology we can add Petri net dialect-specific properties or constraints, but we have no ability to distinguish between the core concepts from the Petri net ontology and concepts Petri net dialect-specific concepts. One possible solution is to use XML/RDF namespace mechanism. But, this solution is also limited to use in Protégé. We need a better way to represent ontology modularization. Accordingly, we decided to use OWL and an OWL-based UML profile in order to overcome these RDF(S) limitations of the Petri net ontology.

5 OWL-based Petri net ontology

For ontology development we use the Ontology UML profile (OUP) (see [9] for details) that is based on the forthcoming ontology language OWL [3]. The OUP provides stereotypes and tagged values for full ontology development. OUP models can be (automatically) transformed into OWL ontologies (e.g., using XSLT) [12]. Using the OUP, one can represent relations between the core concepts of the Petri net ontology and the specifics of a Petri net dialect. For that purposes we suggest using the OUP's package mechanism. In the OUP, we attach `<<ontology>>` to a package. That means the package is an ontology. Accordingly, we can put all core concepts of the Petri net ontology in an `<<ontology>>` package. If we extend the Petri net ontology with concepts of a Petri net dialect we only need to create a new `<<ontology>>` that would be related with the core `<<ontology>>` through the `<<include>>` dependency. In Figure 8 we illustrate this extension principle.

Figure 8 Extension mechanism of the Petri net ontology: support for Petri net dialects



This example depicts how we extend the Core Petri net ontology (`<<ontology>>` *Petri net core*) with concepts of Upgraded and Time Petri nets (e.g. we attach new properties to the core classes for a Petri net dialect). An additional advantage of this approach is that we have the ability to merge concepts from a number of ontologies (i.e., `<<ontology>>` packages). As a result we obtain one ontology definition, for instance, in OWL (by applying XSLT). Comparing with the current PNML proposal for the Petri Net Definition Types [5] one can see that this approach improves the maintainability of Petri net concepts, and better supports reusability of the Petri net ontology concepts. So far, we have defined the Petri net ontology extensions for: P/T nets, Time Petri nets, and Upgraded Petri nets.

5.1 The Core Petri net ontology

The Core Petri net hierarchy, which is shown in Figure 3, is the same for the Petri net ontology represented in the OUP. Actually, there is a difference with regard to both associations and attributes in the model from Figure 3, since ontology development understands property as a fist-class concept. Thus, it is necessary to transform all association between classes as well as all class attributes into the OUP property stereotypes (`<<DataTypeProperty>>` and `<<ObjectProperty>>`). Note that in the OUP Petri net ontology we do not need the *Feature* class since property is the first class in ontology development. Accordingly, we have `<<ObjectProperty>>` and `<<DatatypeProperty>>` that represent properties in the Petri net ontology. On the other hand, we want to provide support for graphical features (*GraphicalFeature*). Figure 9 gives an example of the `<<ObjectProperty>>` *name* that has already been declared as a graphical feature. In this case, the *name* property has as its range (through the `<<range>>` association) the *NameDescriptor* `<<OntClass>>`. However, this class is inherited from the *GraphicalFeature*. *GraphicalFeature* is introduced in the Petri net ontology to be the root class for all the classes that constitute the range for a graphical feature. Similarly, we define other graphical features (e.g. marking). In addition, the *name* property has domain (the <<domain>> association): *Net* and *Node*.

Figure 9 An example of a graphical feature defined in the Ontology UML profile: name object property

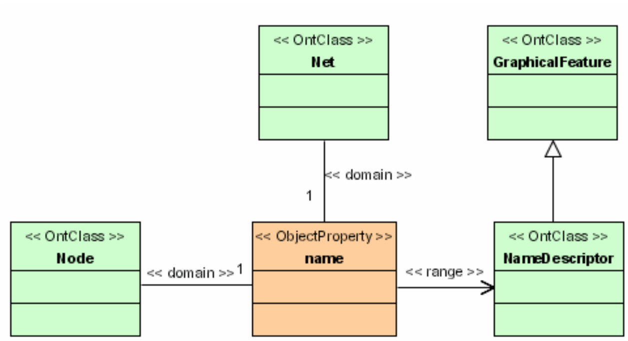


Figure 10 shows an excerpt of the Petri net ontology in OWL. It was generated using an XSLT for transformation from the OUP ontology (i.e., XMI) to OWL [12]. The figure illustrates a part of the OWL *TransitionReference* restriction on the *reference* property. This restriction states that *TransitionReference*'s property *reference* must take all values from (*allValuesFrom*) the union of the following classes: *Transition and TransitionReference.* It is important to note that in the OWL ontology logical expressions (*owl:unionOf*) take an XML form (e.g. the *TransitionReference* restriction), unlike the Protégé PAL constraints that are written in a Lisp-like form. It is more convenient to parse an ontology statement represented in an XML format using standard XML parser, as well as transform it using the XSLT mechanism.

Figure 10 A part of the Petri net ontology in OWL: the *TransitionReference* class restriction from Figure 3 expressed in OCL. This is also shown in PAL in Section 4.2

```
<owl:Class rdf:ID="TransitionReference">
    <rdfs:subClassOf rdf:resource="#NodeReference"/>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#reference"/>
            <owl:allValuesFrom>
                <owl:Class>
                    <owl:unionOf rdf:parseType="Collection">
                        <owl:Class rdf:about="#Transition"/>
                        <owl:Class rdf:about="#TransitionReference"/>
                    </owl:unionOf>
                </owl:Class>
            </owl:allValuesFrom>
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
```

5.2 An Extension example: Upgraded Petri nets

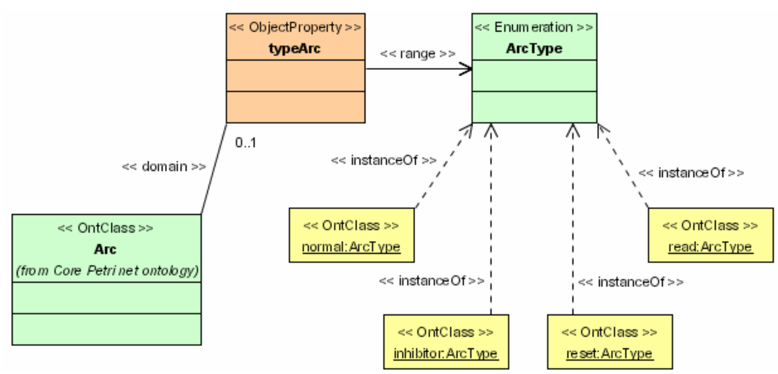
Here we illustrate the ontology that describes Upgraded Petri nets [30] (a Petri net dialect for modeling hardware), in order to show how the Petri net ontology can be extended. The same procedure can be applied to describe other Petri net dialects (e.g. Time Petri nets, Coloured Petri nets, etc.). Figure 11 shows the concepts that should be introduced in the ontology in order to support Upgraded Petri nets. Most of these concepts are ontology properties: *attribute X*, *attribute Y* are graphical features of the Place class; *function*, *function firing level*, and *time function* are features of the Transition class; and *typeArc* is a feature of the Arc class. The ontology extension for Upgraded Petri nets also requires a restriction on the Arc class: an arc can only connect a place and a transition.

Figure 11 Relation between the Core Petri net concepts and their Upgraded Petri net extensions

Core ontology concepts	**PLACE**	**TRANSITION**	**ARC**
Upgraded Petri net extensions	▪ *attribute X* – graphical feature ▪ *attribute Y* – graphical feature	▪ *function* – feature ▪ *function firing level* – feature ▪ *time function* – feature	▪ *an arc can only connect a place and a transition* – restriction ▪ *typeArc* – feature

In terms of OUP this extension means that we have a new `<<ontology>>` package, which contains all Upgraded Petri net specific concepts and restrictions. Figure 12 shows how we attached the *typeArc* property to the *Arc* class. In fact, the domain of the *typeArc* property is *Arc* whereas the enumeration *ArcType* is the range of the *typeArc*. The enumeration *ArcType* consists of four individuals: normal, inhibitor, reset, and read.

Figure 12 An extension of the Arc class for the Upgraded Petri nets: the typeArc property with its range (enumeration of the following values – normal, inhibitor, reset, and read)



Having introduced all Upgraded Petri net concepts and restrictions in the OUP model we can generate its OWL equivalent using XSLT [12]. Figure 13 contains an excerpt of the generated OWL ontology for the *Arc* class. On the left side (Figure 13a) we give the *Arc* class definition of the Core Petri net ontology. On the right side (Figure 13b) we show an excerpt of the *Arc* class definition of the ontology for Upgraded Petri nets. It should be noted that Figure 13b only depicts how the *typeArc* property is added in the OWL ontology. Firstly, we added the *typeArc* property as a definition of a new object property, which has the *Arc* class as its domain, and the *ArcType* class as its range. *ArcType* is an enumeration that consists of the individuals we have already mentioned. The *Arc* class only has a cardinality restriction on the *typeArc* property. Note that the *Arc* class for Upgraded Petri nets contains all definition of the *Arc* class from the Core Petri net ontology (i.e. from Figure 13a). Following the same principle the XSLT converter produced the other parts of the OWL ontology for Upgraded Petri nets.

Figure 13 OWL definition of the Arc class: a) for the Core Petri net ontology; b) for the ontology of Upgraded Petri nets

```
<owl:Class rdf:ID="Arc">
    <rdfs:subClassOf rdf:resource="#ModelElement"/>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#multiplicity"/>
            <owl:maxCardinality
            rdf:datatype="&xsd;#nonNegativeInteger">
            1</owl:maxCardinality>
        </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#fromNode"/>
            <owl:cardinality
            rdf:datatype="&xsd;#nonNegativeInteger">
            1</owl:cardinality>
        </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#toNode"/>
            <owl:cardinality rdf:datatype=
            "&xsd;#nonNegativeInteger">
            1</owl:cardinality>
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
```

a)

```
<owl:ObjectProperty rdf:ID="typeArc">
    <rdfs:range rdf:resource="#ArcType"/>
    <rdfs:domain rdf:resource="#Arc"/>
</owl:ObjectProperty>
<owl:Class rdf:ID="ArcType">
    <owl:oneOf rdf:parseType="Collection">
        <ArcType rdf:about="#normal"/>
        <ArcType rdf:about="#inhibitor"/>
        <ArcType rdf:about="#reset"/>
        <ArcType rdf:about="#read"/>
    </owl:oneOf>
</owl:Class>
<ArcType rdf:ID="normal"/>
<ArcType rdf:ID="inhibitor"/>
<ArcType rdf:ID="reset"/>
<ArcType rdf:ID="read"/>
<owl:Class rdf:ID="Arc">
    <!-- This extended Arc class has the same content
    as the Arc class in the Core Petri net ontology plus
    the following content: -->
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#typeArc"/>
            <owl:maxCardinality
            rdf:datatype="&xsd;#nonNegativeInteger">
            1</owl:maxCardinality>
        </owl:Restriction>
    </rdfs:subClassOf>
    <!--Restriction that an arc can only connect a place
    and transition-->
</owl:Class>
```

b)

6 Tools for the Petri net ontology

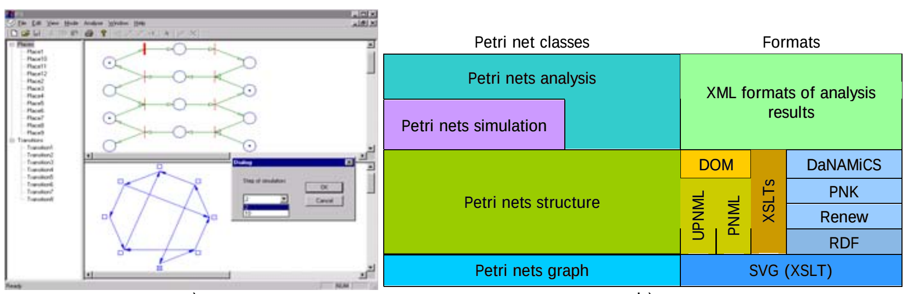
In order to show practical tool support for the Petri net ontology, we depict the P3 tool. This tool has been initially developed for Petri net teaching [11], but we extended it, and thus it can be used in conjunction with the Petri net ontology. Being based on the PNML concepts, P3 is compatible with PNML. The P3 tool supports P/T nets and Upgraded Petri nets. A P3 screenshot is shown in Figure 14a. The P3's architecture is shown in Figure 14b. P3's classes supporting different Petri net functionalities (Petri net classes) is shown on the left in Figure 14b, whereas the supported formats are on the right side.

The formats supported by P3 are the main point of interest for the Petri net ontology. The P3's model sharing mechanism is based on using PNML. All other formats are implemented in P3 using XSLT (from the PNML). Accordingly, P3 can export to the following Petri net tool formats: *DaNAMiCS* – a tool that uses an ordinary text format, *Renew* – a tool that uses another XML-based format; *Petri Net Kernel* – a tool that uses PNML, but since there are some differences between this PNML application and the P3's PNML, we had to implement an XSLT; *PIPE* – a tool that uses PNML (no need XSLT).

P3 tool has the ability to generate RDF description of a Petri net. This P3's feature is also implemented using XSLT. The generated RDF is in accordance with the Petri net ontology (in its RDFS form). We also implemented the XSLT for the opposite direction, i.e. to transform RDF into PNML, and hence we can analyze RDF-defined Petri nets using standard Petri net tools. P3 implements conversion of the PNML Petri net model description to SVG. Since this format can be viewed in standard Web browsers, it is suitable for creating, for instance, Web-based Petri net teaching materials. Learning objects, created in this way, have their underlying semantics described in RDF form, and can be transformed into PNML as well as analyzed with standard Petri net tools. P3 provides two kinds of RDF annotations [15] for SVG:

- As embedded metadata – an RDF description is incorporated in SVG documents. The standard SVG has the `metadata` tag as an envelope for metadata (Figure 15).
- As remote metadata – an RDF description is in a separated document.

Figure 14 P3, ontology-based Petri net tool: a) P3 screenshot; b) P3 architecture: classes supporting different Petri net functionalities (Petri net classes) and supported XML formats



a) b)

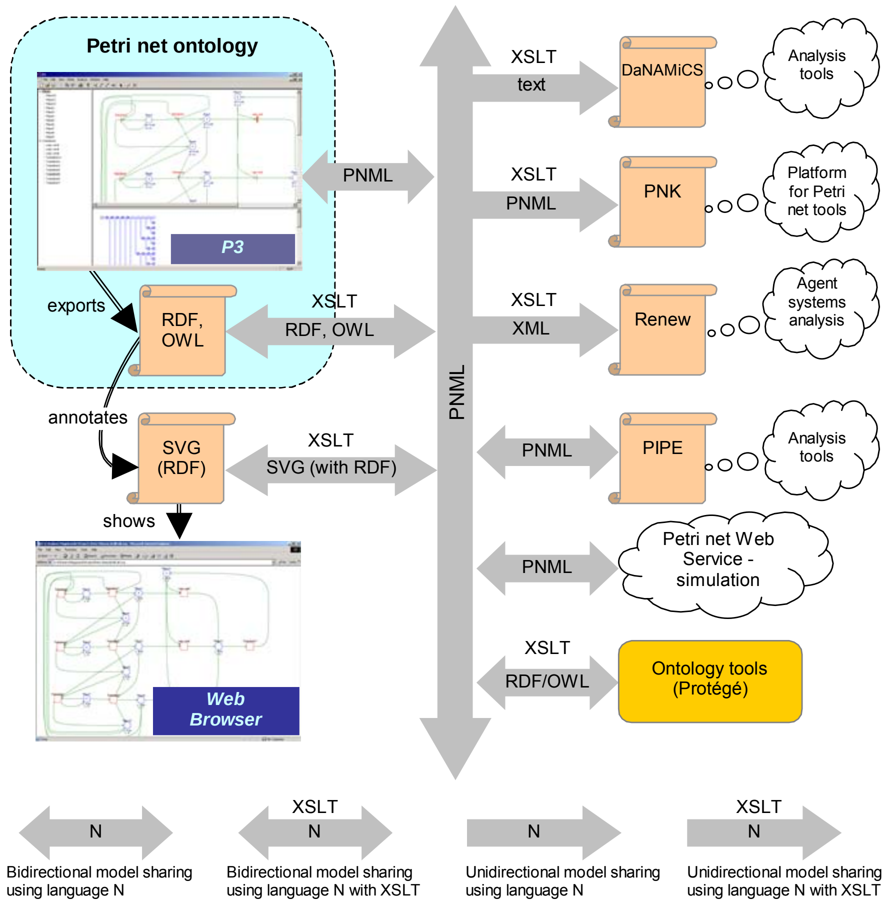
Figure 15 An example of an RDF-annotated SVG document that contains a simple Petri net (n1) consisting of a place (p1), a transition (t1), and an arc (a1) that connects p1 and t1

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<svg width="2000px" height="2000px">
   <metadata>
      <rdf:RDF>
      <!-- We omitted namespace definitions due to their
      length -->
         <Place rdf:about="p1" ID="p1" rdfs:label="p1">
            <position rdf:resource="p1posGIdesc"/>
            <initialMarking rdf:resource="p1initmarking"/>
            <marking rdf:resource="p1marking"/>
            <name rdf:resource="p1name"/>
         </Place>
         <Transition rdf:about="t1" ID="t1" rdfs:label="t1">
            <position rdf:resource="t1posGIdesc"/>
            <name rdf:resource="t1name"/>
         </Transition>
         <Arc rdf:about="a1" ID="a1" arcType="normal"
         rdfs:label="a1">
            <fromNode rdf:resource="p1"/>
            <toNode rdf:resource="t1"/>
            <multiplicity rdf:resource="a1multi"/>
         </Arc>
         <!-- RDF descriptions for properties of Place,
         Transition, and Arc are omitted due to their
         length  -->
         <Net rdf:about="n1" ID="n1" rdfs:label="n1">
            <elements rdf:resource="p1"/>
            <elements rdf:resource="t1"/>
            <elements rdf:resource="a1"/>
         </Net>
      </rdf:RDF>
   </metadata>
```

```
   <title>Petri net - SVG format</title>
   <desc>Exported from the P3 Petri net tool</desc>
   <g id="a1">
      <path class="Line" d="M76 176 L 217 176"/>
      <g transform="translate(217 ,176)">
         <g transform="translate(-22, 0)">
            <g transform=" rotate(0)">
               <path class="Line" d="M 0 -3 L 7 0"/>
               <path class="Line" d="M 7 0 L 0 3"/>
            </g>
         </g>
      </g>
   </g>
   <g id="p1">
      <circle class="Place" r="15" cx="76" cy="176"/>
      <text class="Label" text-anchor="middle" x="76"
         y="160">p1</text>
   </g>
   <g id="t1">
      <rect class="Transition" x="202" y="161"
         width="30" height="30"/>
      <text class="Label" text-anchor="middle" x="217"
      y="160">t1</text>
   </g>
</svg>
```

Although P3 uses RDF, it does not mean that we have abandoned PNML. On the contrary, since we implemented an XSLT (from RDF to PNML), we continued using PNML. Actually, we enhanced PNML because one can use P3 to convert a PNML model to RDF, and then the Petri net model can be validated against the Petri net ontology. That way, we achieved a semantic validation of Petri net models. Of course, PNML is very useful since it contains well-defined concepts for Petri net models interchange and it is now used by many Petri net tools. Furthermore, since we implemented the XSLTs from PNML to Petri net formats of other Petri net tools, we can also employ their Petri net analysis capabilities (e.g. Petri net invariant analysis).

Figure 16 Petri net infrastructure for the Semantic Web (that uses "PNML-based bus" for model sharing): the Petri net ontology, current Petri net tools, P3 tool, Web-based applications, Petri net Web Service, and ontology tools for validation of Petri net documents using the ontology



In Figure 16 we show the Semantic Web infrastructure for Petri nets, which is now implemented. This infrastructure summarizes all major features of P3. The central part of this infrastructure is PNML, since it would be (probably) a part of the future High-level Petri net standard [20]. P3 can be linked with other Petri net tools though PNML (e.g., with PIPE), or by using additional XSLTs on PNML models (DaNAMiCS, Renew, and PNK). Also, P3 has XSLTs for conversions between PNML and RDF in both directions. Besides, P3 generates SVG by using XSLT from PNML to SVG. An XSLT is developed to generate the RDF-annotated SVG from the PNML. We have also developed the XSLT that transforms RDF-annotated SVG documents to PNML. This XSLT is based on the XSLT form RDF to PNML. Hence we have XSLTs for conversions between PNML and SVG in both directions.

7 An application Example

Previously presented Petri net infrastructure for Semantic Web can have various applications in practice. Here, we introduce a Web-based educational system that uses Petri nets. This system was developed following proposed directions for the next generation of educational Web systems in the context of Semantic Web.

Systems of this type have two kinds of users: teachers and students. A teacher creates learning materials using P3 tool. P3 has capacity to save Petri net models in SVG format that can be annotated using RDF compliant to RDFS Petri net ontology. In this way, generated materials for learning Petri nets can be used in different Web systems (e.g. one can develop a dynamic Web learning application that uses Petri nets' SVG format to illustrate learning subject). Students use these materials for studying.

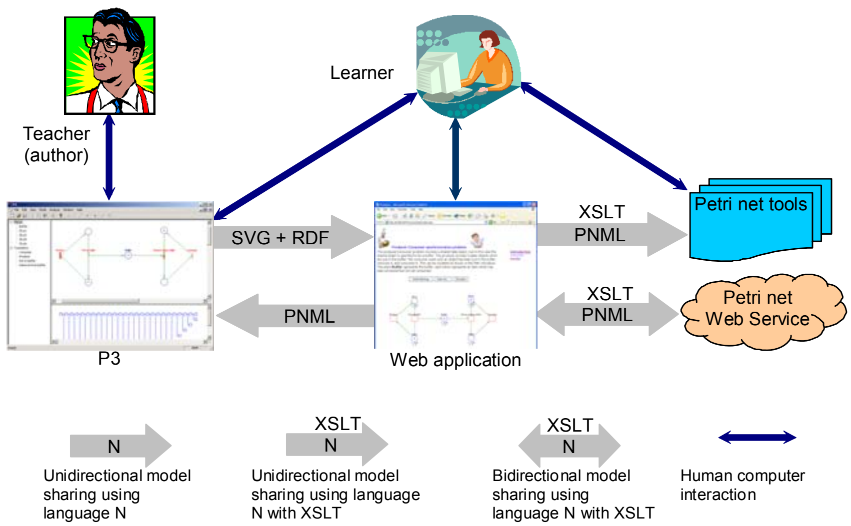
In order to empower Web applications with ability to perform interactive simulation of Petri net models, implementation of the logic of Petri net execution is needed. This can be achieved using Web service for Petri nets simulation presented in [17]. This is a PNML-based Web service, i.e. it uses PNML Petri net model as input, performs one simulation step and generates result, again, in PNML format. Web application should forward Petri net model to the Web service. This model is converted from RDF annotated SVG format into PNML format using an XSLT. Once the simulation is finished, another XSLT is used to transform the result from PNML to RDF annotated SVG format. Both XSLTs are part of proposed infrastructure. Web application should only implement calls to Web service procedures; the whole logic of Petri net model simulation is implemented in Web service. Figure 17 depicts suggested approach to educational systems development using proposed Petri net infrastructure for the Semantic Web.

Using these principles, we have developed a concrete Web application that helps students to understand and learn producer/consumer synchronization problem. This problem is a common part of many different courses in computer science. Very often it is used to illustrate how one can make use of Petri nets to model computer processes since their application in this domain offers numerous advantages.

Educational system, developed as a Web application is fairly simple and it consists of three main parts:

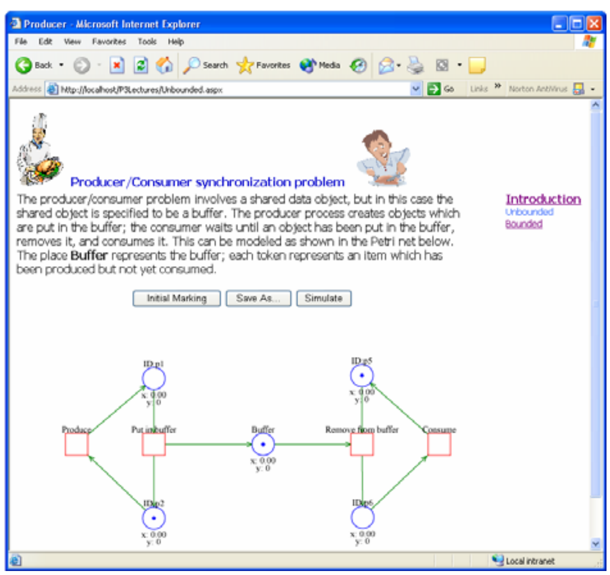
1. An introductory Web page that provides a description of the producer/consumer problem,
2. A Web page that describes Petri net solution of the problem with unbounded divided region (see Figure 18),
3. A Web page that describes Petri net solution of the problem with bounded divided region.

Figure 17 A proposed approach to using Petri net infrastructure for the Semantic Web in Web-based educational systems



Each Web page contains a graphical presentation of adequate Petri net model (based on RDF annotated SVG) and provides support for simulation with that model (using Web service for Petri net simulation).

Figure 18 A Web page that describes a Petri net solution to the producer/consumer synchronization problem with unbounded divided region



User begins his/her interaction with the application by pressing button **Initial Marking** in order to define initial marking of the Petri net model. Automatically, Petri net graph conforms to the specified data reflecting changes of the model. A press on the

Simulate button is a sign for system to start simulation of the model. Simulation is performed in collaboration with the Web service according to the previously explained scenario. Simulation results are shown on the Petri net graph. User can save a Petri net he/she is working with in PNML format by choosing button **Save as**.

8 Conclusions

The main idea of this paper is that the Petri net ontology should provide the necessary Petri net infrastructure for the Semantic Web. The infrastructure understands Petri nets sharing using XML-based ontology languages (i.e., RDFS and OWL). The Petri net ontology and Semantic Web languages do not abandon the PNML. On the contrary, we presented the "PNML-based bus" that takes advantage of the PNML together with the Petri net ontology. That way, we can exploit potentials of current Petri net tools in the context of the Semantic Web. We also presented P3, the Petri net tool that creates ontology-based Petri net models. Its abbreviated version, its technical description, as well as a few developed XSLTs can be downloaded from http://www15.brinkster.com/p3net.

The paper gives guidelines for putting Petri nets on the Semantic Web. It also shows complementary features of the Petri net syntax and semantics by the example of the PNML and the Petri net ontology. The example of RDF-based annotation of SVG documents indicates how to annotate other XML formats (e.g., Web Service Description Language – WSDL). This opens the door to incorporating "Petri net-driven intelligence" into Web-based applications (e.g., Web Service composition, Web Service analysis and simulation [24]).

In the future, the P3 tool will support OWL and OWL-based annotation of SVG documents with Petri net graphs. Furthermore, we will use this annotation principle to develop a Petri net Web-based learning environment, as well as to create Learning Object Metadata (LOM) repositories of Petri net models.

References

1 Baclawski, K., Kokar, M., Kogut, P., Hart, L., Smith, J.E., Letkowski, J. and Emery, P. (2002) 'Extending the Unified Modeling Language for ontology development', *International Journal on Software and Systems Modeling*, Vol. 1, No. 2, pp. 142-156.

2 Bause, F., Kemper, P. and Kritzinger, P. (1995) 'Abstract Petri Net Notation', *Petri Net Newsletter*, No. 49, pp. 9-27.

3 Bechhofer, S., van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D.L. Patel-Schneider, P.F. and Stein, L.A. (2004) 'OWL Web Ontology Language Reference', *W3C Recommendation*, February. http://www.w3.org/TR/2004/REC-owl-ref-20040210

4 Berthelot, G., Vautherin, J. and Vidal-Naquet, G. (1988) 'A syntax for the description of Petri Nets', *Petri Net Newsletter*, No. 29, pp. 4-15.

5 Billington, J., Christensen, S., van Hee, K., Kindler, E., Kummer, O., Petrucci, L., Post, R., Stehno, C. and Weber, M. (2003) 'The Petri Net Markup Language: Concepts, Technology, and Tools', *Proceedings of the 24th International Conference on Applications and Theory of Petri Nets (ICATPN2003)*, Eindhoven, The Netherlands, pp. 483-505.

6 Bock, C. (2003) 'UML without Pictures', *IEEE Software*, Vol. 20, No. 5, pp. 33-35.

7 Breton, E., and Bézivin, J. (2001) 'Towards an Understanding of Model Executability', *Proceedings of the International Conference on Formal Ontologies in Information Systems*, Ogunquit, Maine, USA, pp. 70-80.

8 Chandrasekaran, B., Josephson, J.R. and Benjamins, V.R. (1999) 'What Are Ontologies, and Why Do We Need Them?', *IEEE Intelligent Systems*, Vol. 14, No. 1, pp. 20-26.

9 Djurić, D., Gašević, D., and Devedžić, V. (2005) 'Ontology Modeling and MDA', *Journal on Object Technology*, Vol. 4, No. 1, pp. 109-128.

10 Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, USA.

11 Gašević, D. and Devedžić, V. (2004) 'Teaching Petri nets using P3,' ***Educational Technology & Society***, Vol. 7, No.4, pp. 153-166.

12 Gašević, D., Djurić, D., Devedžić, V., and Damjanović, V. (2004) 'Converting UML to OWL ontologies', *Proceedings of the 13th International World Wide Web Conference on Alternate track papers & posters*, NY, USA, pp. 488-489.

13 Gómez-Pérez, A. and Corcho, O. (2002) 'Ontology Languages for the Semantic Web', *IEEE Intelligent Systems*, Vol. 17, No.1, pp. 54-60.

14 Gruber, T. (1993) 'A translation approach to portable ontology specifications', *Knowledge Acquisition*, Vol. 5, No.2, pp. 199-220.

15 Handschuh, S., Volz, R. and Staab, S. (2003) 'Annotation for the Deep Web', *IEEE Intelligent Systems*, Vol. 18, No. 5, pp. 42-48.

16 Hansen, K.M. (2001) 'Towards a Coloured Petri Net Profile for the Unified Modeling Language – Issues, Definition, and Implementation', *Internal Center for Object Technology Report COT/2-52*, University of Århus, Århus, Denmark. http://www.daimi.au.dk/~marius/writings/cpn.pdf, 2001.

17 Havram, M., et al (2003) 'A Component-based Approach to Petri Net Web Service Realization with Usage Case Study', *Proceedings of the 10th Workshop Algorithms and Tools for Petri nets*, Eichstätt, Germany, pp. 121-130.

18 ISO/IEC/JTC1/SC7 Working Group 19 (2002) 'New proposal for a standard on Petri net techniques' http://www.daimi.au.dk/PetriNets/standardisation/

19 Jackson, D. ed. (2003) 'Scalable Vector Graphics (SVG) Specification v1.2', *W3C Working Draft*. http://www.w3.org/TR/2003/WD-SVG12-20030715/

20 Kindler, E. (2004) 'High-level Petri Nets – Transfer Syntax', *Proposal for the International Standard ISO/IEC 15909-2 Draft Version 0.3.0*, April. http://www.upb.de/cs/kindler/publications/copies/ISO-IEC-15909-2-Draft.0.3.0.pdf

21 Klein, M. (2001) 'XML, RDF, and Relatives', *IEEE Intelligent Sys.*, Vol. 16, No. 2, pp. 26-28.

22 Kogut, P., Cranefield, S., Hart, L., Dutra, M., Baclawski, K. Kokar, M. and Smith, J. (2002) 'UML for Ontology Development', *The Knowledge Eng. Review*, Vol. 17, No. 1, pp. 61-64.

23 Murata, T. (1989) 'Petri nets: Properties, analysis and applications', *Proceedings of the IEEE*, Vol. 77, No. 4, pp. 541-580.

24 Narayanan, S. and McIlraith, S. (2003) 'Analysis and Simulation of Web Services', *Computer Networks: Int. J. of Computer and Telecom. Networking*, Vol. 42, No. 5, pp. 675-693.

25 Noy, N. F., Sintek, M., Decker, S., Crubézy, M., Fergerson, R.W. and Musen, M.A. (2001) 'Creating Semantic Web Contents with Protégé-2000', *IEEE Intelligent Systems*, Vol. 16, No. 2, pp. 60-71.

26 OMG document formal/03-03-01 (2003) 'OMG Unified Modeling Language Specification v1.5', http://www.omg.org/cgi-bin/apps/doc?formal/03-03-01.zip

27 Peleg, M., Yeh, I. and Altman, R. (2002) 'Modeling Biological Processes using Workflow and Petri Net Models', *Bioinformatics*, Vol. 18, No. 6, pp. 825-837.

28 Peterson, J. (1981) *Petri net theory and the modeling of systems*, Prentice Hall, NJ, USA.

29 Semenov, A., Koelmans, A. M., Lloyd, L. and Yakovlev, A. (1997) 'Designing an Asynchronous Processor Using Petri Nets', *IEEE Micro*, Vol. 17, No. 2, pp. 54-64.

30 Štrbac, P. (2002) 'An Approach to Modeling Communication Protocol by Using Upgraded Petri Nets', *PhD dissertation*, Military academy, Belgrade, Serbia and Montenegro.

31 Weber, M. and Kindler, E. (2003) 'The Petri Net Markup Language', in H. Ehrig, W. Reisig, G. Rozenberg, H. Weber (eds.) *Petri Net Technology for Communication Based Systems* (*LNCS Vol. 2472)*, Springer, Berlin, pp. 109-124.