A Petri Net-based Approach to OWL Ontology Representation

Lorena Chavarría-Báez[1] and Oscar Díaz[2]

[1] Instituto Politécnico Nacional, Escuela Superior de Cómputo,
México, D.F., México

[2] Universidad del País Vasco, Facultad de Informática,
San Sebastián, España

lchavarria@ipn.mx, oscar.diaz@ehu.es

Abstract. *Ontologies* are an essential component to construct the *Semantic Web*. Therefore, their accuracy and quality must be assured. From the point of view of several fields of study, *models* (or *representations*) are useful to achieve that goal. *Petri Nets* are a graphical and mathematical modeling tool whose capabilities to express important systems characteristics makes them valuable for ontology representation, verification and reasoning. However, given that their application has been reduced to depict taxonomies, the verification and reasoning process are partially achieved. So, there is a need to have a comprehensive ontology representation. In this paper, we introduce the *Ontology Conditional Coloured Petri Net (OCCPN)* model, a Petri Net extension to represent ontologies. Unlike existing proposals, OCCPN not only gives details of the taxonomy but, by evaluating conditions, also provides modeling primitives to create complex classes and describe properties as well as additional restrictions.

Keywords: Ontology, modeling, Petri net

1 Introduction

Ontologies as an essential component of the *Semantic Web* [1]. This puts stringent demands on their accuracy and quality [11].

Different areas, e.g. knowledge representation, ontology engineering, and model–based engineering, highlight the fact that *"models are useful to ensure quality and discover errors in conceptual design"* [4]. In this sense, the first activity to develop high–quality ontologies should be to create an expressive enough and well–founded model that serves as their surrogate, and, then use it in the evaluation process.

By following this methodology, some works apply *model–theoretic notions* to the design and analysis of ontologies, so that, ontology representation and examination are performed by using the principles of a given mathematical

theory [6, 5]. Although promising, this procedure requires high understanding of formal tools. Other approaches use *Petri Nets (PNs)* to ontology representation and verification [8, 13]. Liu et al. [8] introduce the State Controlled Coloured Petri Net (SCCPN), which includes both structures to depict ontology concepts and their associated relationships and dynamic knowledge inference by means coloured tokens. The same authors, but in a posterior work [13], claim that standard ontologies are not sufficient to handle imprecise or fuzzy information and, as a result, extend SCCPN to model and verify fuzzy ontologies. Nevertheless, this work does not provide class constructors nor property characterization, which limits the scope of the verification and, in consequence, falls short to account for the rich scenario described in [11]. This begs what is the research question of this paper: *to what extend can Petri Nets capture the expressiveness required for modeling ontologies*. Up to date, PNs has been used for ontology modeling, only at a taxonomic level, given their capacity to describe concurrent, asynchronous, distributed, parallel, non–deterministic and/or stochastic systems [9]. Therefore, in this paper, we address the question based on previous experience on PNs, specifically, we tap into the *Conditional Coloured Petri Net (CCPN)* [7, 3], a PN extension designed for active rules [10]. We expand CCPN to represent ontologies and develop the *Ontology CCPN (OCCPN)*. By evaluating conditions, OCCPN not only depicts a taxonomy but also provides the means to create complex classes, describe properties and additional restrictions. This makes our model more comprehensive. As a PN extension, the OCCPN has both a graphical and mathematical model [9]. In this paper, only the former is presented.

The rest of the paper is organized as follows. Section 2 explains the set of OWL language constructs. Then, CCPN foundation is presented. Section 3 illustrates an example of the ontology representation using OCCPN. Section 4 and 5 gives details about classes and properties, respectively. Section 6 shows a complete OCCPN model. Finally, Section 7 formulates conclusions and future work.

2 Background

This section outlines two important notions: the *Web Ontology Language*, a means for authoring ontologies, and *Petri Nets*, particularly, the *Conditional Coloured Petri Net*, a useful modeling tool.

2.1 The Web Ontology Language

The *Web Ontology Language (OWL)* is a language designed to represent ontologies through the definition of their parts, namely: *classes (concepts)*, *individuals (instances)*, and *properties (relations)* [14, 12, 2]. A *class* assembles objects with similar characteristics. An *individual*, or *class instance*, is a particular element that satisfies the class description, the set of all class individuals constitute the *class extension*. *Properties* define relations, either between individuals or between individuals and data values. Next subsections give more details about these parts.

Classes OWL has two *predefined classes*: `owl:Thing`, whose extension comprises all individuals, and `owl:Nothing`, which does not have any element. Every new class is a subclass of `owl:Thing`, and `owl:Nothing` is a subclass of any class [2]. There are several forms to declare new classes: the easiest one is creating a *named class*, i.e., a named instance of `owl:Class`; other ways, known as *class constructors*, are those that establish constraints on the elements of their extension. However, this last method produces *anonymous classes*.

Class constructors cover enumeration, set operators, and property restrictions.

Enumeration, `owl:oneOf` clause, fabricates a class by the exhaustive listing of its instances. Set operators, involving `owl:intersectionOf`, `owl:unionOf`, and `owl:complementOf` instructions, describe classes by acting like their traditional set operator counterparts. Property restrictions include *value* and *cardinality* constraints. The former, regulate the *type* of the class extension, and, the latter, the *number*. In this way, `owl:someValuesFrom`, verifies an *existential* quantification, `owl:allValuesFrom`, validates a *universal* one, and `owl:hasValue`, examines a *particular value*. `owl:minCardinality` and `owl:maxCardinality` check the *minimum* and *maximum* quantity of instances in the property, respectively.

When a class is a subclass, by means of the `owl:subClassOf` clause, of another one, the extension of the former is a subset of the extension of the latter. If two classes, using the `owl:equivalentClass` assertion, are equivalent, their extensions are the same. On the contrary, if they are disjoint classes, using the `owl:disjointWith` declaration, the intersection of their extension is the empty set [14, 12].

Individuals Individuals are, either named or anonymous, class members [2]. Therefore, they contain data for each part of the class template: attributes and properties. Additionally, individuals specify facts regarding their identity by means of the sentences `owl:sameAs`, `owl:differentFrom`, and `owl:AllDifferent`, which establishes that two individuals are the same, two individuals are different, and all the individuals in a list are all different, respectively.

Properties *Properties* have *name*, *domain*, and *range*. The property links elements from the domain to those from the range. If components from both domain and range are individuals, then the type of the property is `owl:ObjectProperty`. But, if they are individuals and data types, then the kind is `owl:DatatypeProperty` [2]. A property can have additional features, which imposes new constraints on it. For example, *logical characteristics*, namely: `owl:TransitiveProperty`, and `owl:SymmetricProperty`, as their names suggest, verify if a property fulfills the conditions to be transitive and symmetric, respectively. *Functional features* include the statements `owl:FunctionalProperty` and `owl:InverseFunctionalProperty`. The former describes a condition in which there is, at most, one unique value from the range for each domain instance. The latter checks the functional constraint and its contrary, i.e., there is, at most, one unique value from the domain for each range instance.

Finally, properties are closely interrelated by means the `rdfs:subPropertyOf` statement, which indicates that if $\mathcal{P}_1$ is a sub-property of $\mathcal{P}_2$, then the pairs satisfying the former are a subset of those accomplishing the latter, and the `owl:inverseOf` clause, that defines $\mathcal{P}_2$ as the opposite of $\mathcal{P}_1$ [14, 12].

2.2 Petri Nets

A *Petri Net (PN)* is both a graphical and mathematical modeling tool. It is a directed, weighted, bipartite graph consisting of two types of nodes, namely: *places* and *transitions*, which graphically are represented as circles and bars (or boxes), respectively [9]. Arcs are from a place to a transition or vice versa. In the former case, places are known as input places, and, in the latter, as output places. A marking, or state, assigns a non–negative number of tokens, represented as black dots, to each place of the PN. A transition is enabled if in each one of its input places there is, at least, $w(p, t)$ tokens, where $w(p, t)$ is the weight of the arc from p to t. If the transition fires, the tokens are removed from each one of its input places and are added to each one of its output places [9].

Conditional Coloured Petri Nets (CCPNs) are a PN extension originally developed to model both structure and dynamics of active rules [10]. One of its main advantages is the ability to evaluate complex conditions in transitions. Fig. 1 shows its graphical part, for further details refer to [7].

A *primitive place* represents an indivisible event. A *copy place*, as its name alludes, reproduces its original one. A *virtual place* acts as an information warehouse. A *composite place* delineates complex events, for example, conjunction.

CCPN has the following types of transitions: *rule*, *composite*, and *copy*, which evaluates the condition of an active rule, creates complex events, and duplicates events, respectively. There are two types of *arcs*: *normal* and *inhibitor*. The *token* element defines CCPN dynamics since *a transition fires if it is enabled and*, for rule and composite typed ones, *its condition*, using the data contained in the token, *is evaluated to true*. The inhibitor arc exhibits the contrary behavior, i.e., its transition is enabled if there is no token in its input place.

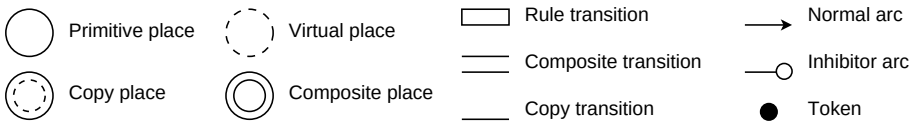

Fig. 1. CCPN basic elements.

3 Ontology Modeling through OCCPN

This section introduces *Ontology CCPN (OCCPN)*, a CCPN augmentation to represent OWL ontologies. The aim is to benefit from the ability of CCPNs to evaluate conditions in transitions and provides modeling primitives to depict the

features described in Section 2.1. In consequence, OCCPN is a richer ontology representation. This part specifically looks into how classes, individuals and properties can be modeled through OCCPN. The intention is to give an intuitive presentation. Section 4 and 5 will later provide a more formal account.

3.1 Classes and Individuals

In OCCPN, a *primitive place* represents a *named class* (see Fig. 2(a)). *Place color* includes the elements: 1) name, 2) axioms, indicating relationships among classes, 3) attributes, attached to class with datatype properties, and 4) operations, depicting object properties. When *defining a new named class*, both name and axioms take a constant value, the rest of the attributes acquire data with *each new class instance*. *Class instantiation* is represented by *depositing tokens* into a primitive place. Such as Fig. 2(b) shows, each token has a name, a type, data for place color, and identity facts.

3.2 Properties

A *property* is represented by the set of ordered pairs, (x, y), $x \in$ *domain*, $y \in$ *range*, for which the property is evaluated to true. Since *rule transitions* appraise conditions, they are useful to stand for properties.

Fig. 2(c) shows the OCCPN structure to model *object properties*: given a rule typed transition, T_0, its input places, C_0 and C_1, symbolize domain and range, respectively, and its output spot, C_2, depicts the set of ordered pairs that satisfy the condition attached to transition. C_2's color has the following elements: 1) name, corresponding to property's name, 2) logical characteristics, 3) property interaction, and 4) functional features.

Since *datatype properties* link individuals to data values, they do not need a rule typed transition, they are captured by *class attributes*. For a given datatype property, its *domain* matches with the *class name*, its *name* corresponds to the *attribute name*, and its *range* constraints the *attribute's datatype*.

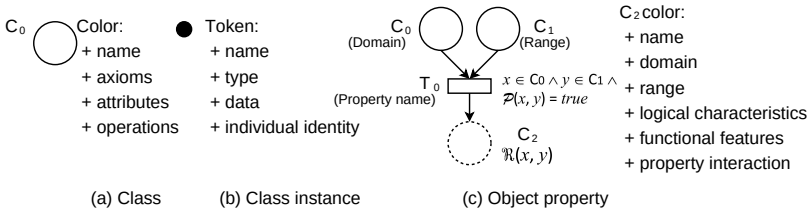

Fig. 2. OCCPN structures for basic ontology elements.

3.3 Example

This section shows an excerpt of a camera ontology which describes a little issue in the field of photography. The complete specification is in reference [14].

Fig. 3 demonstrates, on the left side, the OWL statements to describe two classes, namely: `Lens` and `ValueRange`, and some properties, both datatype and object. Its corresponding OCCPN structure is on the right side.

Classes, defined in lines 1–2 and 8–9, correspond to primitive places labeled as `ValueRange`, and `Lens`, respectively. Lines 3–7, which connect the datatype property named *minValue* to `ValueRange` class, match to attribute definition in the color of `ValueRange` place. A similar situation emerges around lines 10–14 which characterize a datatype property for `Lens` class.

Object property, declared in lines 15–18, ties, through `aperture` relation, individuals from `Lens` class to those coming from `ValueRange` category. In OCCPN, this behavior is achieved by using a rule typed transition having as input places those representing `Lens` and `ValueRange` classes. Its output place acts as a warehouse of those pairs that overcome condition evaluation.

Finally, lines 19–20 creates an individual of `Lens` class, which, in turn, is reflected in OCCPN by the marking in the `Lens` place. This token comprises the name and type for the focalLength attribute.

```
1. <owl:Class rdf:about="&myCamera;ValueRange">
2. </owl:Class>
3. <owl:DatatypeProperty rdf:about="&myCamera;minValue">
4.   <rdfs:domain rdf:resource="&myCamera;ValueRange"/>
5.   <rdfs:range rdf:resource="&xsd;float"/>
6. </owl:DatatypeProperty>
7. <rdfs:Datatype rdf:about="&xsd;float"/>
8. <owl:Class rdf:about="&myCamera;Lens">
9. </owl:Class>
10. <owl:DatatypeProperty rdf:about="&myCamera;focalLength">
11.  <rdfs:domain rdf:resource="&myCamera;Lens"/>
12.  <rdfs:range rdf:resource="&xsd;string"/>
13. </owl:DatatypeProperty>
14. <rdfs:Datatype rdf:about="&xsd;string"/>
15. <owl:ObjectProperty rdf:about="&myCamera;aperture">
16.  <rdfs:domain rdf:resource="&myCamera;Lens"/>
17.  <rdfs:range rdf:resource="&myCamera;ValueRange"/>
18. </owl:ObjectProperty>
19. <myCamera:Lens rdf:ID = "lens_instance">
20. </myCamera:Lens>
```

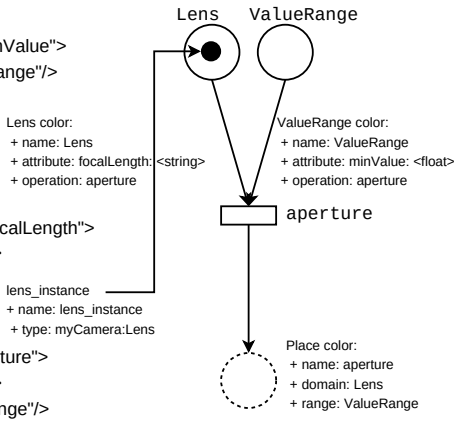

Fig. 3. An ontology part described by OCCPN.

4 Class Descriptions in OCCPN

OCCPN provides modeling primitives for the OWL statements to describe classes in detail, namely: predefined classes, class constructors, and class interrelation.

Fig. 4 shows the OCCPN structures for the OWL predefined classes: `owl:Thing` and `owl:Nothing`. A *source place* is a multicolor one since it represents the set

Fig. 4. OCCPN structures for predefined classes.

of all individuals. It is very similar to a CCPN virtual place in the sense that both act like token depot regardless of token's type; however, a source locality is unique and can have instances. A *sink transition* exemplifies `owl:Nothing` class extension: the *empty set*, since it consumes tokens, but does not produce any. In light of `owl:Nothing` is a subclass of every class, and, to be consistent with the notation follows next, in OCCPN, a sink transition ends with a half-round. A *virtual place* (see Fig. 1) only aids to consummate a class extension; hence, it either has no name nor instances. For this reason, it portrays *anonymous classes.*

4.1 Constructors

The easiest way to create a class in OCCPN is assigning a class name to a primitive place. However, OCCPN also provides arrangements to build more elaborated (anonymous) classes.

Enumeration. In OCCPN, a class defined by enumeration corresponds to a *constant virtual place*, i.e., that containing particular elements, only those specified in its color. Elements outside the color specification are not allowed.

Set operators. Fig. 5 shows the composition for `owl:intersectionOf`, `owl:unionOf`, and `owl:complementOf`. The performance of each structure is analogous to its corresponding logical operator. Place C_{n+1} in Fig. 5(a) stores those individuals that are members of each input place of composite typed transition T_0. T_0 is responsible to verify class membership. Place C_{n+1} in Fig. 5(b) allocates elements which are in C_1, C_2, ..., C_n or in all of them. Finally, Fig. 5(c) uses an inhibitor arc to check complement of a place C_1, i.e., those individuals not in C_1.

Property restriction. Fig. 6 describes the general form of a *restriction on an object property*. Both *cardinality* and *value* constraints are verified by adjusting the condition of transition T_1, which represents the property limitation.

If any cardinality constraint proceeds, the arc (C_1, T_1) modifies its weight with a new value, either w or the interval (w_1, w_2). Then, the transition T_1 verifies that the number of tokens in its input places is enough to fire it according to this new restriction.

Depending on the type of *value constraints*, the condition in transition T_1 verifies either *existential quantification*, *universal quantification*, or *a precise value*.

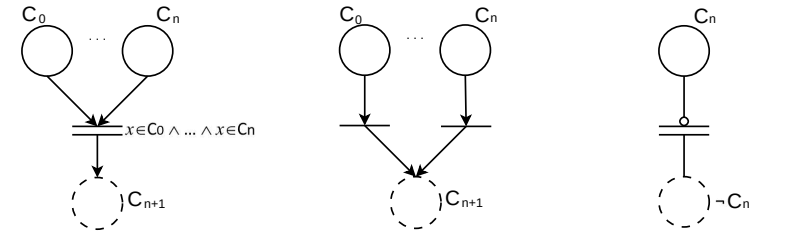

Fig. 5. OCCPN structures for set operators.

Restrictions on a datatype property add new constraints on the attribute that represents it, which, in turn, is in the place color. *Value constraints* delimits *attribute's domain*. On the other hand, *cardinality restrictions* limit the number of values that an attribute can take. Therefore, an attribute can be *not null*, if `owl:minCardinality` has a number, different from zero, as its argument; *mono-valuated*, when its cardinality is exactly one; or *multi-valuated*.

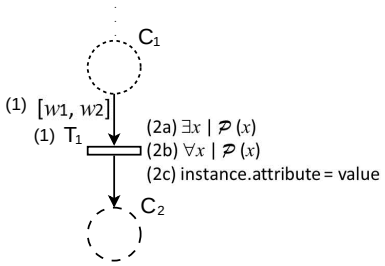

Fig. 6. General form of a restriction on an object property: (1) for `owl:cardinality` constrains, and (2a), (2b), (2c) for `owl:someValuesFrom`, `owl:allValuesFrom`, and `owl:hasValue` constraints, respectively.

4.2 Relation to other Classes

Subclass. In OCCPN, subclasses are pictured as the structure showed in Fig. 7(a). The color of a subclass place comprises its own description and, by extension, the specification of its superclass. Fig. 7(b) presents the case of disjoint subclasses, and, Fig. 7(c), the situation of overlapping ones. Notice that rule typed transitions can replace the copy ones to represent subclassed defined by a predicate.

Disjoint. The OCCPN structure for the disjoint statement is in Fig. 8. The composition encircled in dashed lines acts like a permanently disabled switch, so

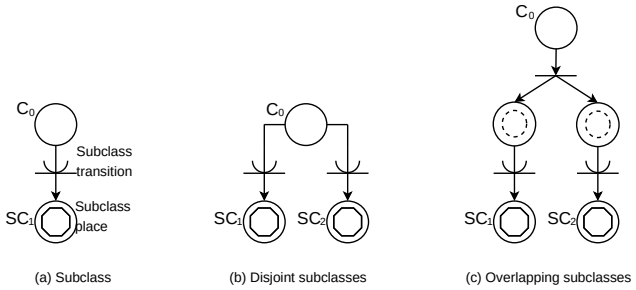

Fig. 7. OCCPN structures for subclasses.

that, place C_1 is never reachable from place C_0 and vice versa, which guarantees that they do not have individuals in common.

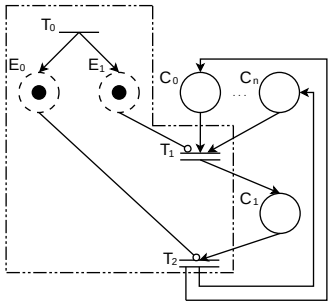

Fig. 8. OCCPN structure for the `owl:disjointWith` clause.

Equivalent. OCCPN does not have any special structure to represent the `owl:equivalentClass` sentence because it is useless to include two different places to describe the same class extension. The section *"class axioms"*, in the place color, describes the equality among classes.

5 Property Definition in OCCPN

In the OCCPN model, most of the property features described in Section 2.1 are either declared as implicit conditions on the attributes in the place color or considered as additional constraints in the rule typed transition. In this last case, property characteristics are also declared in the sections "logical characteristics", "functional properties", or "property interaction" of the place color, as appropriate.

5.1 Logical Characteristics

In OCCPN, logical characteristics of an object property describe new subclass structures, see, for example, Fig. 9(a) that shows the `owl:TransitiveProperty` statement. However, for the sake of the model, the rule typed transition that represents the object property also verifies the condition that describes each one the logical relations, such as Fig. 9(b) shows. By so doing, place C_3 stores all the individuals satisfying the original condition and all the individuals that overcome the second restriction. The color of the place C_3 also indicates the type of the relation.

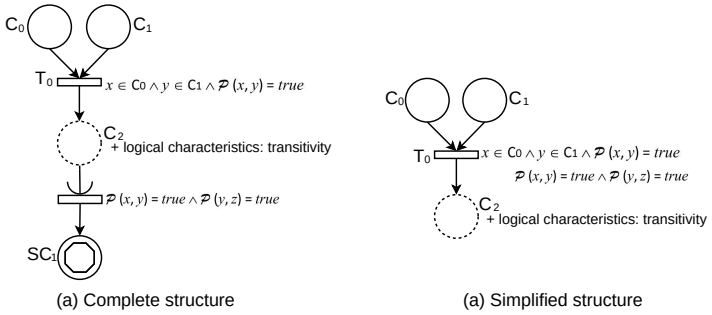

Fig. 9. OCCPN structure for `owl:TransitiveProperty`

5.2 Functional Features

In OCCPN, `owl:InverseFunctionalProperty` and `owl:FunctionalProperty` features on object properties are similar to logical characteristics, i.e., there is a nested condition attached to the rule typed transition.

When the `owl:InverseFunctionalProperty` applies on datatype properties, it defines a *not null*, *mono-valuated*, and *key* attribute. There is a *not null* and *mono-valuated* attribute if the `owl:FunctionalProperty` is declared.

5.3 Relation to other Properties

Inverse. The clause `owl:inverseOf` does not have any special construction in OCCPN. The color of the output place of the rule typed transitions that symbolize the object property captures this information.

Subproperty. The OCCPN structure for the `rdfs:subPropertyOf` clause is similar to that showed in Fig. 9(a), but T_1 evaluates the condition of the sub-property instead of the logical/functional restriction. As it can be observed, individuals in place C_4 fulfill the conditions of both property and sub-property.

6 Running Example

Fig. 10 presents the complete OCCPN model for the camera ontology described in reference [14]. Section 3 covers the basic transformation from ontology to OCCPN, therefore, we only focus on the `ExpensiveDSLR` complex class creation. As OWL statements in Fig. 10 shows, this class (place E0) is the conjunction (transition T0) of two subclasses (places E1 and E2): one of them comes from `DSLR` class (place E3), and, the other one, is the result of a property restriction (place E4) which establishes that, at least, one camera owner must be a professional (transition T1). This corroborates that OCCPN is able to represent the OWL expressiveness.

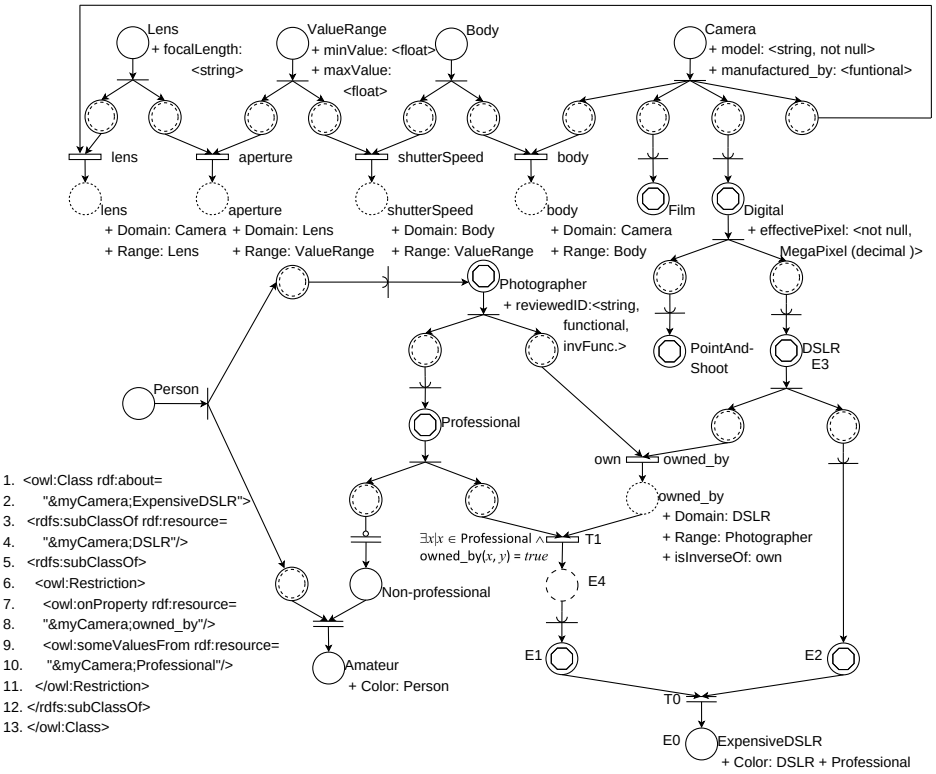

Fig. 10. The camera ontology [14] described by OCCPN.

7 Conclusions

This paper introduces OCCPN, a model to represent OWL ontologies. Previous approaches focus on the creation of classes through assigning them an identifier,

and relations of specialization, exclusion, and instantiation. By contrast, OCCPN provides modeling primitives for the whole set of OWL language constructs, and in so doing accounts for a comprehensive approach. Future work includes the development of the formal reasoning process on OCCPN. Additionally, we plan to address the usefulness of OCCPN to detect not only structural errors, such as redundancy, circularity, and contradiction, but also those involving semantics.

References

1. W3C Semantic Web Activity. Available in: http://www.w3.org/2001/sw/
2. Bechhofer, S., Van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D.L., Patel-Schneider, P.F., Stein, L.A.: W3C Recommendation, chap. OWL Web Ontology Language Reference (2004)
3. Chavarría-Báez, L., Li., X., Palma-Orozco, R.: Estimating the number of test cases for active rule validation. In: Proc. of the 12th Mexican International Conference on Artificial Intelligence (MICAI 2013). pp. 120–131 (2013)
4. Dubois, C., Famelis, M., Gogolla, M., Nobrega, L., Ober, I., Seidl, M., Völter, M.: Research Questions for Validation and Verification in the Context of Model–Based Engineering. In: MoDeVVa@MoDELS. pp. 67–76 (2013)
5. Grüninger, M.: Verification of the OWL–Time Ontology. In: Proc. of the Tenth Intl. Semantic Web Conference. pp. 225–240 (2011)
6. Grüninger, M., Ong, D.: Verification of Time Ontologies with Points and Intervals. In: Proc. of the Eighteenth Intl. Symposium on Temporal Representation and Reasoning. pp. 31–38 (2011)
7. Li, X., Medina-Marín, J., Chapa-Vergara, S.: Applying Petri Nets in Active Database Systems. IEEE Transactions on Systems, Man, and Cybernetics–Part C: Applications and Reviews 37(4), 482–493 (2007)
8. Liu, J., Wang, K., He, Y.L., Wang, X.Z.: Formal Representation and Verification of Ontology Using State Controlled Coloured Petri Nets. In: Dai, H., Liu, J., Smirnov, E. (eds.) Reliable Knowledge Discovery. pp. 269–290. Springer–Verlag (2012)
9. Murata, T.: Petri Nets: Properties, Analysis and Applications. Proceedings of the IEEE 77(4), 541–580 (1989)
10. Paton, N.W., Díaz, O.: Active Database Systems. ACM Comput. Surv. 31(1), 63–103 (1999)
11. Poveda-Villalón, M., Gómez-Pérez, A., Suárez-Figueroa, M.: OOPS! (OntOlogy Pitfall Scanner!): An On–line Tool for Ontology Evaluation. Intl. Journal on Semantic Web and Information Systems 10(2), 7–34 (2014)
12. Staab, S., Studer, R.: Handbook on Ontologies. Springer Publishing Company, Incorporated (2009)
13. Wang, K., Liu, J., Ma, W.: Towards the Detection of Potential Contradictions in Fuzzy Ontology Using a High Level Net Approach Integrated with Uncertainty Inference. In: Proc. of the IEEE Intl. Conf. on Data Mining Workshops. pp. 884–890 (2010)
14. Yu, L.: OWL: Web Ontology Language. In: Yu, L. (ed.) A Developer's Guide to the Semantic Web. pp. 155–238. Springer, Heidelberg (2011)