A Language for Transforming the RDF Data on the Basis of Ontologies

Pavel Shapkin and Leonid Shumsky
Department of Cybernetics and Information Security,
National Research Nuclear University MEPhI (Moscow Engineering Physics Institute),
31 Kashirskoe Shosse, Moscow, Russian Federation

Keywords: RDF, OWL, Transformations, Ontology.

Abstract: Languages such as XSLT are used to transform XML documents based on their syntactical structure. The emergence of Semantic Web technologies brings new languages such as RDF and OWL which are able to represent the semantics of data. Currently there is no data transformation language which is capable of using this semantic information. In this paper a language is described which is aimed at transforming the RDF data. Like XSLT, it is based on templates. The transformation is driven by ontologies. Among the features of this language is the ability to check the validity of template systems, which guarantees that the transformation will terminate successfully without raising any errors. In order to prove this property the formal model of the language is studied.

1 INTRODUCTION

The development of web technologies resulted in the appearance of new languages, such as RDF and OWL, which are aimed at representing the semantics of data (Lassila et al., 1998; Smith et al., 2004). These languages form the basis of the Semantic Web — a Web of "Linked Data" accompanied with ontologies which enable to capture the meaning of data and expose it in machine-readable form.

As a universal data representation language RDF is suitable for solving data integration tasks. An interesting use case for RDF occurs with the proliferation of cloud computing and SaaS (Software as a Service) applications. Being an open Web standard RDF suits very well for integrating SaaS applications, which operate in the Web environment. Since RDF is not used in every system it is often needed to generate the RDF representation from other formats and to transform RDF graphs to different representations.

Among the systems that are suitable for RDF data transformations we should mention the systems described in (Furche et al., 2004). There also exist a number of systems which adopt an XSLT-like approach to the RDF transformations (Kawamoto et al., 2006; Davis, 2003). A major drawback of existing systems is inability of inference on ontologies. More to say, most of these systems do not even use OWL documents as knowledge bases. There exist ontology manipulation languages such as OPPL (Egana et al., 2008) which is designed to automate routine knowledge engineering tasks but is not suitable to develop end-user ontology-based applications. Another approach to RDF transformation is to rely on SPARQL CONSTRUCT statements (Corby et al., 2014; Alkhateeb and Laborie, 2008), but since SPARQL is originally a general-purpose query rather than data manipulation language these transformations are not structured very well comparing to our approach that organizes transformations around ontology classes.

Our main idea is to structure transformation templates around ontology classes the same way as methods for object manipulation in object-oriented languages are structured around classes of objects. Thereby the whole transformation process could be modularized and driven by the structure of corresponding ontologies. At the same time we gain possibilities to check templates for well-formedness and validate them: these features prevent run-time errors in the transformation process.

The paper is organized as follows: in section 2 we give an example to illustrate the problems that can be solved using the described language; section 3 defines semantic templates — a formalism used for processing the RDF data using ontologies; section 4 describes the resulting language; in section 5 we give and discuss a series of template systems; the last section discusses some additional features of the template language.

Shapkin P. and Shumsky L..
A Language for Transforming the RDF Data on the Basis of Ontologies.
DOI: 10.5220/0005479505040511
In *Proceedings of the 11th International Conference on Web Information Systems and Technologies* (WEBIST-2015), pages 504-511
ISBN: 978-989-758-106-9

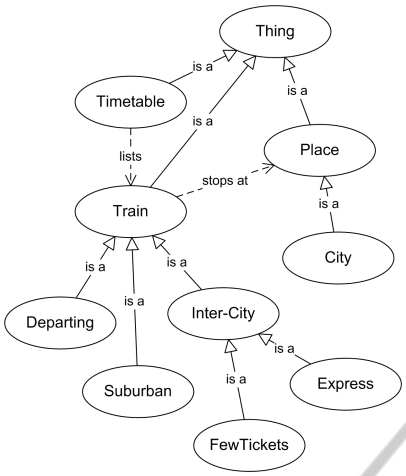


Figure 1: Train ontology.

2 MOTIVATING EXAMPLE

First we would like to introduce an example which will be used later to illustrate how the language works.

Consider the task of generating a train timetable. The source data is represented in the form of an RDF document which corresponds to the ontology illustrated on figure 1. The ontology is presented in OWL.

This ontology is a simplified one used for example. Let's consider its classes. Trains are listed in the timetable. Each train is described by a number, the departure time and the list of stops. Trains stop at different places, described by their names. A place might be a city. Trains are divided into inter-city and suburban, which are disjoint classes. Inter-city trains are those that stop at least at one city, all the other trains are considered suburban. Moreover, an inter-city train is called express train if it stops only in cities. We would like to be able to render an HTML representation of the timetable, which contains a list of trains. Each type of train has to be represented differently. Additionally, we would like to mark trains that are departing in an hour (Departing concept). Also, inter-city trains have a limited number of tickets and we would like to highlight those for which there are less than 10 tickets left (FewTickets concept).

The natural way to describe the transformation given is to bind the transformation rules to ontology classes. Then the transformer has to execute atomic transformations for the RDF resources contained in the source document, choosing the most suitable rules basing on the classes that correspond to these resources and the subsumption relations between them. To compute the subsumption hierarchy and do instance checking an OWL reasoner might be used.

We will represent such transformation rules in form of "templates", and the whole transformation will be defined by the "template system". We will get back to this example after introducing the language proposed. We will start with some formal definitions, then we will proceed to the syntax for the language, and in the end we will show the solution of this problem.

3 A FORMAL MODEL FOR THE TEMPLATE-BASED RDF TRANSFORMATION LANGUAGE

3.1 Basic Ideas

We will use an approach similar to XSL (Clark et al., 1999). Every transformation is defined as a set of templates — a *template system* . Each template is bound to a class in the ontology and can transform only its instances. By doing so we make the whole transformation process ontology-driven. We involve the hierarchy of concepts in the computational process. Moreover, if an OWL reasoner is used, we can leverage the decidability of subsumption. It enables us to dynamically compute the subsumption hierarchy of concepts and thus to dynamically change the transformation process in response to the changes in the ontology.

When a template system is applied to an RDF resource that represents an individual this resource is processed using the "best matching template". The best matching template is a template which corresponds to the most specific concept for that individual. Situations might occur in which the best matching template cannot be chosen unambiguously, e.g. when there are two templates and each of them corresponds to a concept that has the input object as an instance. To avoid such situations we will introduce additional limitations on the structure of template systems. We call these limitation "well-formedness".

3.2 Template Matching Algorithm

First, we will introduce the concept of template system base and well-formedness. We will use conventional notation for description logic terms and operations (Baader et al., 2007). Because our language is aimed at OWL, we will mostly use the term "class" for "concept" and "property" for "role" throughout the paper.

Definition 1. *If ts is a template system then its base base(ts) is a set of concepts which correspond to the templates in that system.*

Definition 2. *The template system ts is well-formed iff the following conditions hold:*

1) *ts contains at most one template for each concept;*
2) *the set of concepts extbase = base(ts) ∪ {⊥} forms a lower semilattice, i.e. for every C,D ∈ extbase their intersection C⊓D is also in extbase.*

Figure 2 illustrates the semilattice conditions. Assume that none of the concepts involved is empty (i.e. not equal to ⊥). In this case the concepts marked on the figure 2(a) form a set that is not a lower semilattice: e.g. it contains concepts A and B, but does not contain their nonempty intersection F. On the contrary, the set of concepts marked on the figure 2(b) is a lower semilattice: all intersections of its concepts are also included in this set, as well as the empty concept which is an intersection of disjoint concepts.

Well-formedness has to guarantee that if the template system is applicable to an RDF resource, i.e. there is a template suitable for the individual that this resource represents, this template can be chosen unambiguously. Consider an individual i which is an instance of $A \sqcap B$, but is not an instance of I or E. We cannot choose the most specific concept for i from the concepts marked on the figure 2(a): both A and B are candidates for it. On the other hand, concepts marked on the figure 2(b) contain the concept F which is the most specific concept for i.

We will call the process of checking a template system for well-formedness the *verification*.

In order to construct the template matching algorithm we have to introduce the notion of the template depth.

Definition 3. *The depth of a template τ with a concept C in template system ts is the value of function $depth(\tau, T)$. The function is evaluated by the following rules:*

1. *If there is no template in ts with a concept D such that $C \sqsubset D$ then*
$$depth(\tau, ts) = 0.$$
2. *If there is a number of templates $t_i^{\{i \in 1..n\}}$ in ts which correspond to concepts $D_i^{\{i \in 1..n\}}$ such that $C \sqsubset D_i$ for any i, then*
$$depth(\tau, ts) = max_{i \in 1..n}(depth(t_i, ts)) + 1.$$

The depth shows how deep the concept of a template lies in the subsumption hierarchy of the template system concepts. The following theorem is relatively easy to prove:

Theorem 1. *For a well-formed template system T and individual i if τ ∈ θ(T) has a maximum depth among the templates for which i is an instance of the corresponding concept, then*

1) *such τ is unique in ts;*
2) *this template concept is most specific of concepts from ts for i, that is there is no template in the template system with more specific concept that has i as an instance.*

The proof of this theorem is pretty straightforward and we will omit it here. This theorem enables to construct the algorithm for choosing the best matching template for an individual. The main idea is that templates must be traversed in the descending order of depth. By the stated theorem the first template whose concept has the given individual as an instance is the best matching template. This algorithm relies on the subsumption relation, therefore the corresponding function that chooses the matching template for a given individual is computable as long as the subsumption is decidable. Thus if the ontology used in a template system is defined in OWL DL and the system is well -formed there is a guarantee that the transformations will be unambiguous.

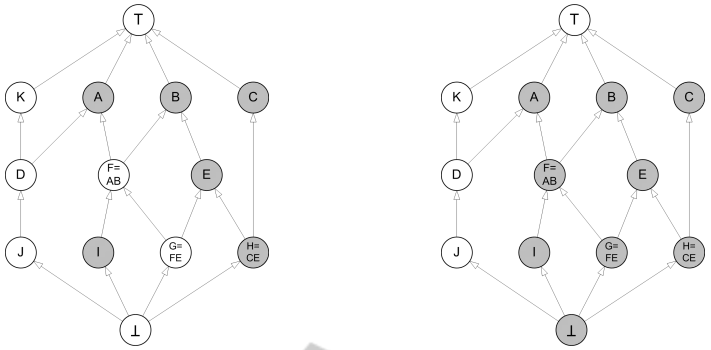
4 THE RESULTING LANGUAGE

4.1 Syntax

We would like to give a detailed description only for the "core language", i.e. the part of our language that is responsible for binding transformations to concepts and leveraging the subsumption hierarchy. Besides this part our language also includes different expressions which are needed to make the transformations useful: conditional expressions, counting and sorting expressions, value processing expressions such as string manipulating functions etc.

We will use BNF notation to describe the syntax of our XML-based transformation language. The syntax of the transformation language is shown on figure 3. Only the mandatory attributes are included, the optional ones are omitted. The root element (`templates`) contains a list of `template` elements.

A single template system is interpreted as a specific transformation. Often a single transformation combines different representations of the same objects: e.g. in our example with train timetable we might have one template to generate train information and another template that generates "outer" appearance to mark departing trains. For these purposes we will need two different template systems that act



(a) Marked concepts do not form a lower semilattice

(b) Marked concepts do form a lower semilattice

Figure 2: Concepts semilattice example (AB denotes $A \sqcap B$)

templates ::= *template* — *template templates*
template ::= `<template class="`*class-uri*`" mode="`*system-id*`">` *individual-exprs* `</template>`'
individual-exprs ::= *individual-expr* — *individual-expr individual-exprs*
individual-expr ::= *apply-templates* — *for-each* — *common-expr*
common-expr ::= *content* — `<uri/>` — *annotation-value*
annotation-value ::= `<annotation-value prop="`*annotation-uri*`" />`
for-each ::= `<for-each prop="`*object-property-uri*`">` *individual-exprs* `</for-each>` —
`<for-each prop="`*datatype-property-uri*`">` *for-each-value-exprs* `</for-each>`
for-each-value-exprs ::= *for-each-value-expr* — *for-each-value-expr for-each-value-exprs*
for-each-value-expr ::= `<value/>` — *content*
apply-templates ::= `<apply-templates mode="`*system-id*`"/>`

Figure 3: The RDF transformation language syntax.

as different "modes" of the same transformation. To handle this situation we added a possibility to combine different template systems in one transformation: each template element has a `mode` attribute which contains an identifier of the corresponding template system.

The transformation process is understood as a recursive application of templates. Each template application is substituted by its result. The resource to which the template currently applies is called the current resource. Places where the next application has to occur are marked with the `apply-templates` tag. In its `mode` attribute contains the identifier of the template system that has to be applied. Besides this element contents of templates might include some common expressions and the `for-each` element.

By common expressions we mean

- arbitrary XML elements or text content;
- special elements that are used to extract URI of the current resource and to extract the values of annotation properties.

The `for-each` element is used to traverse the properties of the current individual. When a `for-each` element is processed all the values of the corresponding property of the current individual are extracted; afterwards this element is substituted by the concatenation of the results of processing each of these values. The contents of the `for-each` element differ depending on whether an object or a datatype property is specified. When an object property is used, the element may have the same contents as template elements, in other cases its contents are limited to static content or the `value` element that extracts the current value.

It is possible to use the construct

`<value prop="`*property-uri*`"/>`

as a shortcut for

`<for-each prop="`*property-uri*`"><value/></for-each>`

4.2 Validating Templates

The same way as the XML documents might be verified for well-formedness and validated against the XML Schema, we would like to have an opportunity to validate templates against a given ontology. The validation process is pretty simple and is used to guarantee that whenever a template is applicable to an individual this individual will have all the properties

used in this template. In turn, it guarantees that application of valid templates to suitable individuals will not raise any "property does not exist" errors.

Suppose that we would like to check the validity of a template which corresponds to a class C. We have to traverse the body of that template and check every access to the properties of the current individual (e.g. by means of the `for-each` construct). If for every property p accessed the assumption

$$C \sqsubseteq \exists p.\top$$

holds then the template is valid. This assumption guarantees that every instance of C will have at least one property p with a value belonging to a top concept (`Thing` class) — e.g. any individual. This assumption does not depend on the range of p, because we only have to check the existence of this property. If the ontology is consistent and the property exists, it will point to an object from its range.

A special case is when we check property accessors that occur inside a `for-each` element. In this case we have to use a concept that corresponds to the range of the property used in this `for-each` element instead of C in the above mentioned assumption.

4.3 Executing Transformations

To execute the transformations one has to supply the following parameters to the transformation processor

- a set of RDF documents representing the ontology and the input data;
- the URI of a resource to start the transformation with.

The execution of a template system should solve an important problem of processing cycling links. In order to avoid infinite computations we have to detect these situations. The algorithm for solving this problem uses the notion of context:

1. The transformation starts from an empty context.
2. Whenever a template is applied to an individual we add this individual to the context of that template.
3. When the template finishes processing the processed individual is removed from that templates' context.
4. If a template is applied to an individual and this individual is already in the context for that template, we conclude that a cycle is encountered.

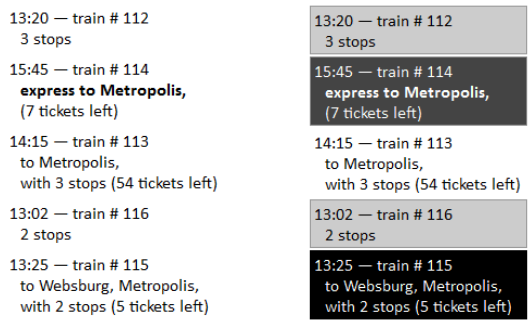
When a loop is detected the transformation stops with an error.

```
<templates>

<template mode="outer" class="Timetable">
  <html> ... <body>
    <apply-templates prop="lists"/>
  </body></html>
</template>

<template mode="outer" class="Train">
  <div><apply-templates mode="header"/></div>
</template>

<template mode="header" class="Train">
  <value prop="departsAt"/> -
  #<value prop="hasNumber" />
  <div class="info"><apply-templates
      mode="info"/></div>
</template>

<template mode="info" class="Train">
  <count-of prop="stopsAt" /> stops
</template>

<template mode="info" class="InterCity">
  to <apply-templates prop="stopsAt" />
  with <count-of prop="stopsAt"
      class="NotCity"/> stops
  (<value prop="ticketsLeft" /> tickets left)
</template>

<template mode="info"
    class="Place"></template>

<template mode="info" class="City">
  <value prop="hasName"/>,
</template>

<template mode="info" class="Express">
  <b>express to <apply-templates
      prop="stopsAt" /></b>
  (<value prop="ticketsLeft"/> tickets left)
</template>

</templates>
```

Figure 4: Transformation example.

5 TEMPLATE SYSTEM EXAMPLE, VERIFICATION AND VALIDATION

Now we can construct the template systems needed to solve our example task. These templates are shown on figure 4. To count the number of stops of a train a counting expression `count-of` is used. We omitted the code that is used to format `dateTime`-typed values of departure times and the definitions of URI prefixes. Using these templates to transform an input we



(a) (b)

Figure 5: Renderings of example transformation results.

```
<template mode="outer" class="Departing">
  <div class="light"><apply-templates
      mode="header"/></div>
</template>

<template mode="outer" class="FewTickets">
  <div class="dark"><apply-templates
      mode="header"/></div>
</template>
```

Figure 6: Example 2.

can obtain an HTML document rendering of which is shown on figure 5(a).

By constructing the templates shown on figure 4 we have not solved our task completely. Originally, we also needed to mark trains that are departing during the next hour (suppose that the current time is 13:00) and trains that are running out of tickets. Assume that we have to mark these trains with different colors: departing trains have to be rendered with a light-gray background, trains that are have tickets left — on a dark background.

We already separated “outer” templates that generate outer appearance of a train record, so we have to include two additional outer templates for corresponding classes. These templates are shown on figure 6. Now trains # 111, 114, 115, 116 will be marked. Nevertheless there is an inconsistency in the resulting template system: there are less than 10 tickets left for train # 115 and it is departing in an hour, but we have no template to deal with this situation. This inconsistency is exactly the one that can be caught using the verification.

In fact the “outer” template system in our example is not well formed: there are templates for Departing and FewTickets classes, but no template for their intersection is present. At the same time this intersection is not empty: train # 115 is an instance of this intersection concept.

When we add the missing template for the intersection, all template systems will be well formed. The template that we have to add is shown on figure 7, it renders a train on a black background. The final rendering is shown on figure 5(b).

```
<template mode="outer">
  <class>
    <owl:Class>
      <owl:intersectionOf
          rdf:parseType="Collection">
        <rdf:Description
            rdf:about="Departing"/>
        <rdf:Description
            rdf:about="FewTickets"/>
      </owl:intersectionOf>
    </owl:Class>
  </class>
  <div class="black"><apply-templates
      mode="header"/></div>
</template>
```

Figure 7: Example 2.

In this template we do not use a predefined class, we give its definition in the `class` element of the template instead. This is a feature of the transformation language that uses the power of OWL as an algebraic language for class definitions: every class definition is an algebraic expression and we can use these expressions instead of predefined classes. Combined with a reasoner we can compute subsumption relations for a class that corresponds to an expression “on-the-fly”.

In addition to verification we can also validate our template systems according to the procedure described in the section 4.2. The templates are valid and this guarantees that transformations will be carried out without errors.

6 ADDITIONAL FEATURES

As it was mentioned before we described here only the “core language” that leverages the benefits of ontology model in order to operate the transformations. The language has additional constructs that improve the expressiveness of transformations which we omit in this article. However there are some constructs that correspond to several features of OWL that we would like to mention. Some of them are already implemented, others are in development.

Besides the class subsumption hierarchy, OWL supports the subsumption of properties and `same-as` relationships. Currently we have support for these features in our language.

To leverage subsumption of properties one can write special templates for property values. These templates apply whenever a value of the correspond-

ing property or its subproperty is processed using the `value` tag. Inference of property subsumption granted by the reasoner makes it possible to select the most suitable template in every case and to verify property templates. Property templates can be understood as reusable snippets that reduce repeatable code for property value processing.

The *same-as* reasoning is used in another variant of templates — individual templates. These templates are bound to concrete individuals in the ontology. Provided that there is a *same-as* relation between individuals i and i' and the individual template is present for i then this template is used whenever a template needs to be applied to i'. We can validate an individual template using known assumptions about this individual and by computing the most specific concept for it.

Another feature that is currently under development are class and property templates. These templates apply directly to classes and properties in contrast to previously mentioned templates that apply to class instances and property values. Class templates can be used to render a description of a class which can be based on the values of annotation properties. These templates could also be used to process (i.e. apply templates to) the set of known instances of the corresponding class as well as the sets of its known subclasses or superclasses.

A class template is applicable to all the classes that subsume the corresponding class. The same holds for property templates. Class and property template systems might be verified in the same way as other template systems.

7 CONCLUSIONS AND FUTURE WORK

We described a language for ontology-driven transformations of the RDF data. Transformations consist of templates. Each template is bound to a class from an ontology and is suitable to transform its instances. Template bodies can be verified and validated against the corresponding class definition in order to to eliminate possible run-time transformation errors. We have shown an example which illustrates the process of verification.

The transformation syntax is based on XML but the transformation result can be of any text-based format; thus, one can define RDF to XML, RDF to HTML, RDF to plain text or even RDF to RDF transformations and so on. The language is currently implemented on the Java platform using the Scala language. Jena framework in conjunction with Pellet reasoner are used for ontology processing.

Utilization of the reasoner helps to transform documents relying upon their semantic structure:

- the transformation process will pick up the right templates not only for individuals that state their type explicitly but also for individuals that are implicitly classified by the reasoner;
- the transformation result for different documents which are semantically equivalent will be the same, e.g. if the ontology states that p' is the inverse of property p then documents containing $a\,p\,b$ and $b\,p'\,a$ will be equally transformed.

There is a number of extensions that we plan to implement in our future work, e.g. the possibility to bind templates to "anonymous" classes expressed as description logic formulas ('married people', 'married people with kids' etc.): right now it is only possible to bind to "named" classes by their URI.

The principles of ontology-based template transformations are utilized in a more general framework for the processing of RDF data. Whereas the template language is suitable only to generate different representations, i.e. construct some strings, from the RDF resources, the framework is suitable to execute arbitrary operations. This result is achieved by substituting templates with functions whose return type is not limited to strings. Interesting point is that this framework might be considered as a means of object oriented programming (OOP) on ontologies. Instance templates correspond to instance methods and class templates — to static methods. This enables to use ontologies as class hierarchies in OOP programs without losing reasoning capabilities.

REFERENCES

Alkhateeb, F. and Laborie, S. (2008). Towards extending and using SPARQL for modular document generation. In *Proceedings of the eighth ACM symposium on Document engineering*, pages 164–172. ACM.

Baader, F., Calvanese, D., McGuinness, D., Nardi, D., and Patel-Schneider, P. (2007). *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press.

Clark, J. et al. (1999). XSL transformations (XSLT) version 1.0. *W3C Recommendation 16 November 1999*.

Corby, O., Faron-Zucker, C., and Gandon, F. (2014). SPARQL template: A transformation language for RDF. report, Inria RR-8514.

Davis, I. (2003). RDF template language 1.0. Specification Draft.

Egana, M., Antezana, E., and Stevens, R. (2008). Transforming the axiomisation of ontologies: The ontology pre-processor language. *Proceedigns of OWLED*.

Furche, T., Bry, F., Schaffert, S., Orsini, R., Horroks, I., Kraus, M., and Bolzer, O. (2004). Survey over existing query and transformation languages. *REWERSE*.

Kawamoto, K., Kitamura, Y., and Tijerino, Y. (2006). Kawawiki: A semantic wiki based on RDF templates. In *Proceedings of the 2006 IEEE/WIC/ACM international conference on Web Intelligence and Intelligent Agent Technology*, pages 425–432.

Lassila, O., Swick, R. R., Wide, W., and Consortium, W. (1998). *Resource Description Framework (RDF) Model and Syntax Specification*. World Wide Web Consortium.

Smith, M. K., Welty, C., and McGuinness, D. L. (2004). OWL web ontology language guide. *W3C Recommendation 10 February 2004*, 10.