Ontology Transformations

Laurent Wouters, Marie-Pierre Gervais

Ontology Transformations

Laurent Wouters
EADS Innovation Works
European Aeronautic Defence and Space Company
Suresnes, France
laurent.wouters@eads.net

Marie-Pierre Gervais
Université Paris Ouest
Laboratoire d'Informatique de Paris 6 (LIP6)
Paris, France
marie-pierre.gervais@lip6.fr

***Abstract*—This paper deals with the problem, coming from an industrial context, of ontology transformations. EADS, as a major aircraft manufacturer faces the problem of integrating works of experts from different domains using different notations. Addressing this Domain-Specific Language (DSL) problem, we previously developed a solution based on OWL2 ontologies for the integration of multiple domains at the abstract syntax level. Our next step is then the production of visual concrete syntaxes from this abstract syntax, for each domain. Considering this problem as a transformation issue, we raise the challenge of ontology transformations. We provide an OWL2-based rule language for the expression of such transformations. Validating this approach, our rule language has been implemented in a rule and transformation engine and tested on applications coming from the industry.**

***Keywords*-Ontology Transformation, MOF-based Model Transformation**

I. Introduction

These last years, Domain Specific Languages (DSL) are gaining in importance in the Model-Driven Development (MDD) community. They enable Domain Experts to model themselves their knowledge, rather than delegate this task to a software engineer. This lowers the risk of misunderstanding, or loss of information during the knowledge elicitation phase. However this requires making available a *visual concrete syntax* intuitive enough and close to the experts' domain. EADS, as a major actor of the aircraft industry, faces such a challenge. The design of complex systems such as aircrafts typically involve experts from multiple domains, e.g., System Engineering, Thermal Engineering, Structural Engineering and Human Sciences. All these experts have their own specific knowledge, know-how and skills. They do not have the same background regarding modeling. They work with their own modeling tool, if any, using their own domain-specific notations and vocabulary. However, together they have to model the same final product and consequently cooperatively build a single *artifact*. In a previous work, we focused on how to express such an *artifact*. We illustrated the requirements coming from our industrial area, and advocated for a language having specific characteristics: native multi-level modeling capabilities, support of inferences, and behavior modeling constructs [1]. We consequently proposed xOWL, a structural and behavioral modeling language extending the W3C OWL2 standard [2]. xOWL has been designed for the expression of a common abstract syntax for all domains. Thus, its meta-model includes the OWL2 metamodel enhanced with behavioral concepts.

In an approach similar to DSLs, domain-specific concrete syntaxes can be developed for each class of domain experts. Thus, said experts can access a common *artifact* through visual notations specific to their respective domains.

Providing the visual notations from the common abstract syntax requires, for each domain, mapping concepts of this abstract syntax to drawing elements (e.g., Rectangle, Ellipse, Image, Text, etc). A single concept may have different notations in different domains. For example, the same engine in an aircraft is noted as an electrical generator in electric schemas and as a heat source in heat transfer schemas. Consequently, for each domain, mapping rules must be expressed and then applied in order to obtain concrete syntax elements from the abstract syntax.

We choose to deal with this issue as an ontology transformation problem, considering that the drawing primitives can also be represented within an ontology. Each transformation would take the common artifact as input and outputs the corresponding drawing primitives.

Contrary to ontology transformations, model transformations are well investigated with metamodel-based approaches. Most of them provide ways to express the relations between an input and an output metamodel, which are usually expressed in the Meta-Object Facility language (MOF). A transformation engine can then implement the transformation, operating at the model level. Operational and declarative, as well as hybrid approaches have been devised for expressing the transformations themselves. Our issue at hand then seems to be solved using existing transformation techniques and tools.

Because we are using OWL2 ontologies as input and output, we face the following dilemma, summarized in Fig. 1. We can either:

1) Use the MOF-based model transformation technologies, represented by the τ arrow in Fig. 1. This would require a paradigm shift from the OWL2 to MOF world, or
2) Investigate a native OWL2 ontology transformation approach, represented by the τ' arrow in Fig. 1.

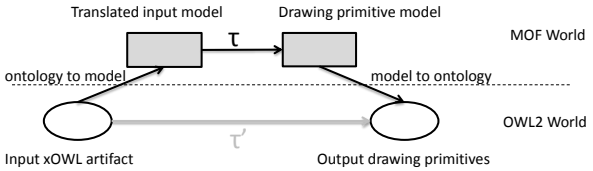

Figure 1. Solutions for the transformation of OWL2 ontologies

The first solution mentioned hereinbefore seems appealing because it reuses existing model transformation solutions. However, it also has hidden costs and limitations. We first have to translate the original input ontology into a MOF-based model conforming to a metamodel, which itself conforms to MOF. This intermediate representation can then be transformed and the result translated back to OWL2. In addition to the fact that three transformations are operated where only one is really necessary, the translation of an ontology to a MOF-based model is not an easy task and has several limitations [3]. Mainly, the complete semantics of OWL2 ontologies cannot be mapped to exactly equivalent MOF-based models [4]. In particular, OWL2 natively supports the expression of multi-level models through a mechanism called "punning", enabling a model element representing a domain entity to be a class and an instance at the same time. Such native OWL2 constructs require to be expressed using substitutions constructs [5]. This leads to the loss of information. Consequently, in this paper we propose a native OWL2 ontology transformation approach.

Hereafter in Sect. II, we first provide a simple but representative example of two domains with their respective notations. Because the transformations themselves must be specified in some way, we investigate in Sect. III the existing model transformation languages and technologies. We then propose in Sect. IV our own rule language for OWL2, building upon the existing state of the art. This proposition is validated by the prototype implementation described in Sect. V, its application to an industrial example, as presented in Sect. VI and a study of the implementation's performances presented in Sect. VII. We finally conclude and present some perspectives in Sect. VIII.

II. Example

In this section we describe a small but representative use case example in order to illustrate our objective. In the industry, the design of a product's manufacturing process is as important as the design of the product itself. This means, engineers will have to design the manufacturing workshops, assembly lines, as well as the associated processes along the product itself. Following an MDD approach, engineers model the workshops, lines and manufacturing processes. The resulting model will be used to ensure that, for example, the products can indeed be physically built under given time constraints.

In this example, a WORKSHOP is defined as a set of WORKSTATIONs. A WORKSTATION corresponds to a physical location dedicated to an activity in the assembly line. There are multiple kinds of WORKSTATIONs, such as WELDING WORKSTATIONs and ASSEMBLY WORKSTATIONs. In addition, WORKSTATIONs can have a list of STEPs describing the work of the assigned operator. When a WORKSTATION does not have any STEP, it is unmanned.

Experts from different fields have to access this common model:

- *Workshop layout* experts want to see how the different WORKSTATIONs are physically organized and connected.
- *Manufacturing process* experts want to see the manufacturing STEPs associated to each WORKSTATION in order to ensure the consistency of the overall process.

These two fields refer to common concepts (WORKSTATION in this example). However, according to the habits and customs of each field, they use different visual notations.

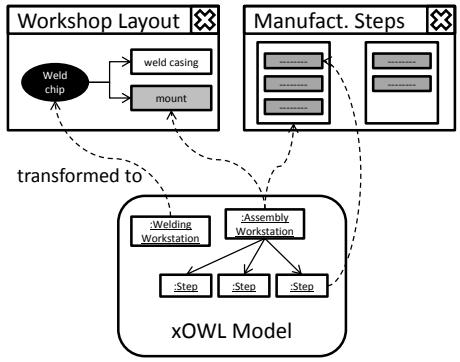

Figure 2. Multiple concrete syntaxes for the same xOWL model

In the *Workshop layout* domain, WELDING WORKSTATIONs are represented as white rectangles with their name within them. ASSEMBLY WORKSTATIONs are represented as grey rectangles with their name within them. However, all unmanned WORKSTATIONs (of any kind) are represented as black discs with their name within them.

In the *Manufacturing process* domain, only manned WORKSTATIONs can be represented. All of them are rendered as white rectangles containing the representation of their respective STEPs. These are represented as grey rectangles with the STEP's description within them. A visual example of these notations is given in Fig. 2.

In each domain, the respective experts have identified a set of representational mappings for the respective domain concepts. These mappings associate model elements from the domain to their respective notational elements, expressed as drawing primitives. The following listings give an informal representation of this knowledge in the form of rules.

Listing 1. Informal rules for the *Workshop layout* domain

```
If ?w is a Welding Workstation named ?n
    and ?w has at least one Step
Then ?wv is a White Rectangle with ?n in it

If ?w is an Assembly Workstation named ?n
    and ?w has at least one Step
Then ?wv is a Grey Rectangle with ?n in it

If ?w is a Workstation named ?n
    and ?w does not have any Step
Then ?wv is a Black Disc with ?n in it
```

Listing 2. Informal rules for the *Manufacturing process* domain

```
If ?w is of type ?wst
    and ?wst is a sub-class of Workstation
    and ?w has at least one Step
Then ?wv is a White Rectangle
    and ?wv is traced to ?w

If ?s is a Step for ?w
    and ?s is described with ?d
    and ?wv is traced to ?w
Then ?sv is a Grey Rectangle with ?d in it
    and ?sv is contained by ?wv
```

Our aim is then to provide a general transformation language for OWL2. To validate our proposal, we apply it to the example of producing the drawing primitives from the domain concepts using the above mentioned rules.

III. State of the Art

Because model transformation is a strong and mature field in the MDD community, we shall first investigate technologies from this field in order to have a better understanding of the possibilities and to identify relevant approaches. Numerous approaches have been devised over time, to the point where the Object Management Group (OMG) specified the Query/View/Transformation (QVT) standard. QVT specifies a set of model transformation languages operating over MOF (Meta Object Facility) models [6]. QVT itself offers two approaches to the definition of model transformations. The QVT-Operational language supports their expression in an imperative style, whereas QVT-Relations and QVT-Core support a declarative approach. These languages have been implemented in different projects, from SmartQVT [7] to mediniQVT [8] and ModelMorf [9]. QVT-based technologies provide interesting properties, such as the traceability of input to output model elements and the support of incremental transformations. Approaches combining imperative and declarative aspects have also been proposed, most notably the ATLAS Transformation Language (ATL) [10]. Moreover, approaches relying on Triple Graph Grammars (TGG) [11] use a declarative definition of the relations between two models. Interpreted in the context of model transformations these relations enable bidirectional and incremental transformations with explicit traceability elements. All these approaches show a very strong support for model transformations in a MOF-based paradigm.

OWL-based transformations are not offered the same extensive support. Although the idea of OWL-based transformations has been proposed in [12], an approach has yet to be devised. In order to cope with this issue, multiple works provide some kind of mapping between MOF-based models and OWL-based ontologies. First and foremost, the Ontology Definition Metamodel (ODM) is an OMG standard for the representation of OWL ontologies in the form of a MOF-based model [13]. However, as shown in Fig. 3, leveraging

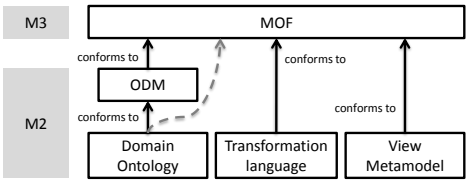

Figure 3. Leveraging ODM for MOF-based ontology transformations

ODM for MOF-based ontology transformation introduce a mismatch in the OMG stack architecture. With ODM at the M2 level, the domain ontology should be at M1, but is required to be at M2 for using it in the MOF-based definition of transformations. To avoid this problem, the semantics of the OWL2 domain ontologies must be mapped directly to metamodels expressed in MOF. This is represented by the dotted arrow in Fig. 3. Clues for this mapping have been provided in [14], although the complete mapping is never demonstrated. The serious limitation of this approach is that the full semantics of OWL2 ontologies cannot be mapped to MOF or UML models [15], [16].

Conversely, other works have bridged the gap in the other direction in providing ways to represent MOF or UML models in OWL2 ontologies. The conversion of UML models to OWL2 ontologies has been investigated in [17] and standardized in the ODM specification document [13]. This direction of the mapping is easier to achieve because the semantics of MOF can be straightforwardly mapped to OWL2 ontologies. Nevertheless, it is still difficult to map arbitrary OWL2 ontologies into MOF-based models and we risk losing the original semantics that may be necessary for the transformation at hand. Consequently, focusing on providing an OWL2-native transformation technology, we investigate the possibilities.

The OWL2 world does not yet provide means for ontology transformations. However, some works have been achieved that shall be investigated in this regard. In particular, an issue at hand is to find a language for the expression of ontology transformations. To achieve this, declarative rules could be used. The Semantic Web Rule Language (SWRL) [18] has precisely been designed for this case. SWRL allows the expression of rules for OWL ontologies, although these are usually used as inference rules. That is to say the semantics of SWRL rules are slightly different from transformation rules. SWRL is clearly designed to operate over individuals (instances) and thus provides easy-to-use language constructs in this regard. However, it lacks the capability to operate over classes and properties [19]. For

example, it is possible with SWRL to match all individuals related by a known property. But it is not possible to match all the properties relating two known individuals. In order to express ontology transformations leveraging the complete range of OWL2's language constructs, one cannot use SWRL. To summarize, to the best of our knowledge at this time, ontology transformations can only be achieved through back and forth translations to MOF-based models in order to use the existing model transformation approaches. This comes at the cost of losing some of the original's ontology's semantics due to the greater expressiveness of OWL2 over MOF or UML. Consequently, we propose a native approach to ontology transformations lifting these limitations. We hereafter define our rule language that can operate over the complete set of OWL2's language constructs.

IV. Rule Language for OWL2

In this section we define the general-purpose rule language for OWL2. In order to demonstrate how this proposal is consistent with the OWL2 philosophy, we first describe in the following paragraph the fundamental concepts of OWL2.

A. OWL2 Fundamentals

OWL2 ontologies are formally defined as sets of axioms [2], where an axiom is a unit of information stating what is true in the domain described by the ontology. The OWL2 language defines multiple kinds of axioms. For example, one can use the `ClassAssertion` axiom in order to state the class-instance relationship between two concepts. The `SubClassOf` axiom can be used to state the sub-classing relationship between two OWL2 classes. Other kinds of axioms are used to state the relations between ontological concepts. The fundamental idea is that axioms being units of information, they can be treated independently from each other's.

Although software must treat OWL2 ontologies as sets of axioms, human users usually interpret them as the set of entities they represent, that is to say, the set of classes, individuals and properties described within them. The OWL2 axioms refer to the ontological entities using their name. For example, expressing that B is a sub-class of A is achieved as:

Listing 3. OWL2 axioms

```
1 SubClassOf(:B :A)
```

In the rule language described hereafter, we build upon this property.

B. Rule Representation

Our proposal integrates with OWL2 by making rules first-class citizens in OWL2 ontologies. That is to say, as illustrated in Fig. 4, rules are defined as elements of an OWL2 ontology in the same way axioms are. As stated above, ontological entities are identified by their name, in fact an Internationalized Resource Identifier (IRI). Being consistent, rules are then also identified by IRIs. Also as shown in Fig. 4,

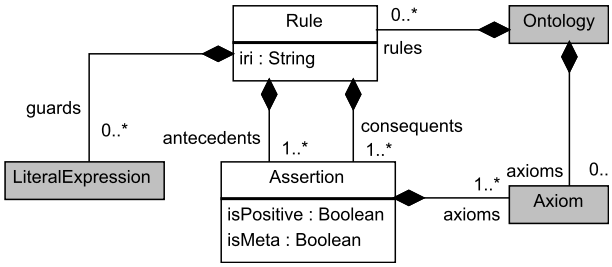

Figure 4. The rule metamodel

rules are composed of a set of antecedent assertions, a set of consequent assertions and a set of guards. Antecedents are conditions to be matched for the rule to trigger and consequents are the results of the eventual rule's triggering. Figure 4 shows an attribute for assertions called *isMeta*. It is used as a flag for transformations, as explained in IV-D. In addition, both antecedent and consequent assertions are given an *isPositive* attribute. A positive antecedent simply describes what shall be matched in the input ontology prior rule's execution. On the contrary, a negative antecedent describes what should not be matched. If a negative antecedent matches the input, it prevents the rule's execution. Consequents shall be regarded as modifications to be applied to the output ontology. Therefore positive consequents are added to the ontology and negative consequents are removed from it. All positive antecedents must be matched and all negative ones not matched in order for the rule to trigger.

A fundamental idea of our proposition is that because OWL2 axioms are units of information that can be treated independently, rules must operate over axioms. That is to say, a rule must match a set of axioms; and its execution results in another set of new axioms. To achieve this, all assertions are defined as a conjunctive set of at least one axiom. With this definition, we are able to express negative conjunctions of axioms. This property is important for the expression of complex rules.

In addition, rules can be guarded by literal expressions that must evaluate to true in order to be fired. Literal expressions not being part of the standard OWL2 language, we rely on xOWL for this purpose.

C. Integration with xOWL

This rule language is integrated with our previous extension of OWL2, called xOWL. xOWL provides procedural algorithms modeling facilities within ontologies. Consequently, xOWL offers extensive language constructs to build literal expressions with usual mathematical operators, string concatenations, etc. xOWL also embeds a query language for OWL2. The queries are simply patterns of OWL2 axioms that can use logical variables to be matched on the repository.

The same language constructions are used to express rules' antecedents and consequents.

For example, the first and last rules shown in Listing 1 can be implemented as:

Listing 4. Implementation of example rules

```
1  Rule(:WeldingWorkstationToRectangle
2    Antecedents(
3      ClassAssertion(model:WeldingWorkstation ?w)
4      DataPropertyAssertion(model:hasName ?w ?n)
5      ObjectPropertyAssertion(model:hasStep ?w ?s)
6    )
7    Consequents(
8      ClassAssertion(view:Rectangle ?r)
9      ObjectPropertyAssertion(view:content ?r ?l)
10     ClassAssertion(view:Label ?l)
11     DataPropertyAssertion(view:value ?l ?n)
12   )
13 )
14
15 Rule(:AssemblyWorkstationToRectangle
16   Antecedents(
17     ClassAssertion(model:AssemblyWorkstation ?w)
18     DataPropertyAssertion(model:hasName ?w ?n)
19     ObjectPropertyAssertion(model:hasStep ?w ?s)
20   )
21   Consequents(
22     ClassAssertion(view:Rectangle ?r)
23     ObjectPropertyAssertion(view:content ?r ?l)
24     ClassAssertion(view:Label ?l)
25     DataPropertyAssertion(view:value ?l ?n)
26   )
27 )
28
29 Rule(:UnmannedWorkstationToDisc
30   Antecedents(
31     ClassAssertion(?wt ?w)
32     SubClassOf(?wt model:Workstation)
33     DataPropertyAssertion(model:hasName ?w ?n)
34     Not(ObjectPropertyAssertion(model:hasStep ?w ?s))
35   )
36   Consequents(
37     ClassAssertion(view:Ellipse ?e)
38     ObjectPropertyAssertion(view:content ?e ?l)
39     ClassAssertion(view:Label ?l)
40     DataPropertyAssertion(view:value ?l ?n)
41   )
42 )
```

Due to simplifications from the real industrial case, these examples only show one-to-one mapping. However, the rule language presented here does not restrict its users to this property. Arbitrarily complex queries can be written using logical variables, as explained hereafter.

These are written in rules as an arbitrary name preceded by the '?' mark. They can be used wherever an ontological entity or a literal can be used. "?w" at line 3 of Listing 4 is an example of a logical variable where an expression of ontological entity is expected. Conversely, "?n" at line 4 is an example of a logical variable where a literal expression is expected. When a rule is matched, the logical variables are bound to the values corresponding to the matched OWL2 axioms. They can be referenced in the rules' consequents as placeholders for their bound values. Rules' consequents also allow the use of logical variables that are not bound by matching the antecedents. That is to say, they do not appear in them. "?r" at line 8 is an example. These variables mean the rule engine will automatically create a new ontological entity (a name) to bind to them. This means, whenever the first rule in Listing 4 is triggered, a new rectangle is created. Thus, a separate rectangle is created for each found WeldingWorkstation.

The use of a negative antecedent assertion is also illustrated in Listing 4 at line 34. It prevents the rule's triggering whenever the workstation "?w" is related to a step ("?s"). Conversely, the first rule in Listing 4 uses the same assertion in its positive form at line 5 in order to match only manned workstations.

Moreover, xOWL expression can be leveraged in order to express computations with rules. For example, we have a family with members and want to transform them to individual persons with full names:

Listing 5. Dynamic values in rule's consequents

```
1  Rule(:MemberToPerson
2    Antecedents(
3      ClassAssertion(m1:Family ?f)
4      DataPropertyAssertion(m1:familyName ?f ?fn)
5      ClassAssertion(m1:Member ?m)
6      DataPropertyAssertion(m1:givenName ?m ?gn)
7      ObjectPropertyAssertion(m1:memberOf ?m ?f)
8    )
9    Consequents(
10     ClassAssertion(m2:Person ?p)
11     DataPropertyAssertion(m2:fullName ?p Plus(?gn ?fn))
12   )
13 )
```

In Listing 5 at line 11, the *Plus* construct coming from xOWL is used to concatenate two strings. The result is then used as the value for the "fullName" property.

D. Transformations

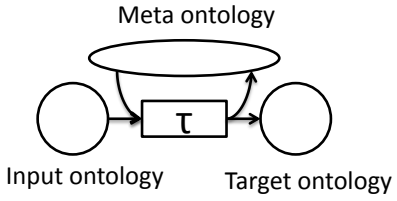

Figure 5. A transformation τ with the "meta" ontology

Building on the rule language described in this section, we now define transformations as sets of unordered rules. Within a transformation, all rules are equal in that there is no notion of priority between them. A transformation matches the axioms within an input ontology and produces modifications to a target ontology. In addition, a transformation may use what we call the meta ontology in order to store information about the current state of the transformation. For example, traceability links between input elements and target elements are to be stored in the meta ontology. This information can then be leveraged in order to support incremental transformations. This meta ontology is accessible from either side of the transformation. That is to say, transformation rules may require matching information within the meta ontology, or conversely add or remove information for it as a rule's

consequent. This mechanism, allowing us to write rules in a flexible manner, is implemented using the *isMeta* attribute of assertions. In this approach, antecedent assertions marked with "Meta" will have to be matched in the meta ontology and not in the input ontology. Conversely, consequent assertions marked with "Meta" will apply to the meta ontology and not the target ontology. Figure 5 summarizes this approach.

In the following sections we will focus on the validation of our rule and transformation language for OWL2 ontologies. This validation is three-fold. Firts, the rule and transformation language has been implemented, as shown in Sect. V. Second, the real-world applicability of this approach is demonstrated on an industrial use case in Sect. VI. Third, further ensuring the real-world applicability of this work, a performance study has been conducted. Its results are presented in Sect. VII.

V. Implementation

In order to validate our approach we first implemented this rule language in the form a rule engine. In our previous work, an interpreter for the xOWL language has been developed. Building on this, the rule engine has been integrated within the interpreter. This allows us to leverage the existing xOWL implementation for the evaluation of rule's guards.

In order to produce an efficient implementation of the rule's engine, we decided to rely on an existing tried and tested pattern-matching algorithm. In this regards, the RETE algorithm [20] was deemed a good candidate because it can be implemented for handling RDF triples and has useful properties such as the support for incremental matches. The OWL2 recommendation provides RDF-based semantics of OWL2 ontologies, detailing how OWL2 axioms can be translated to RDF triples [21].

Relying on a RETE network for pattern-matching, the rule engine is notified whenever a match is detected or invalidated. The rule engine is then responsible for firing the corresponding rule. Because rules' consequents are also patterns, the rule engine "instantiates" them in the sense they are specialized using the value of the matched variables. The rule engine can then be seen as a black box which is fed RDF patches, i.e. the adding or removal of RDF triples. It also outputs RDF patches. This is because rules can have positive and negative consequents, or the firing of a previous rule can be cancelled. An important property of the RETE algorithm is that is works incrementally. Consequently, our implementation of the rule engine is able to perform incremental model transformations. This is useful in the case described in the example from Sect. II because users will be able to modify the transformation's input ontology at any time. Re-executing the whole transformation would be inefficient.

In the following section, we present the application of this approach and its implementation on an industrial example.

VI. Application to an Industrial Example

As a second validation step, the proposed approach has been tested on industrials examples. In this section, we present a complete application case from which the initial example presented in Sect. II has been extracted. In this application, multiple experts from different fields are collaborating in order to design the assembly `workshops`[1] and the manufactoring `process` for a product. The complete `workshops` and manufacturing `processes` have to be modeled. The resulting model will be used in order to simulate the real workshops and processes. The experts previously used standard office software in order to represent their knowledge, but obviously had to do calculations and run the test scenarios by hand. We are expected to provide them modeling tools implementing their respective visual notations so that they are able to build the common artifact with each other's.

Firstly, the xOWL language has been used in order to integrate the concepts of the different domains. The complete ontology for all domains has over 15 classes and 50 relations and properties in order to represent the different kinds of `workstations`, their properties, the `manufacturing steps`' description, etc. In this application case, we have identified three broad fields of expertise, corresponding to different visual notations:

- The *Workshop Layout* experts, as previously mentioned focus on the physical organization of `workstations`. They determine which `workstation` can exchange materials, products, etc. with which `workstation`. In this context, `workstations` are to be understood as reusable resources of skills for certain kinds of tasks.
- The *Manufacturing Process* experts design the complete list of `manufacturing steps` for the product. These experts do not focus on which particular `workstation` will be executing these steps. They rather provide a sequential list of `manufacturing steps`.
- The *Workshop Supervision* experts take the `manufacturing steps` from the previous experts and split them across the `workstations` defined by the *Workshop Layout* experts. Their job is to orchestrate and time the work of the different `workstations` for the `manufacturing steps` in the most efficient way possible.

The *Workshop Layout* experts use a particular representation for `workstations`, as shown in Fig. 6, where the specific notations for `workstations` depend on whether they are automatic or not, and their field of work (pressing, welding, painting, etc.). In Figure 6, automatic `workstations` have a distinctive thick black border. In addition, the `workstations` can be linked by `routes`

[1]This font will be used for domain concepts

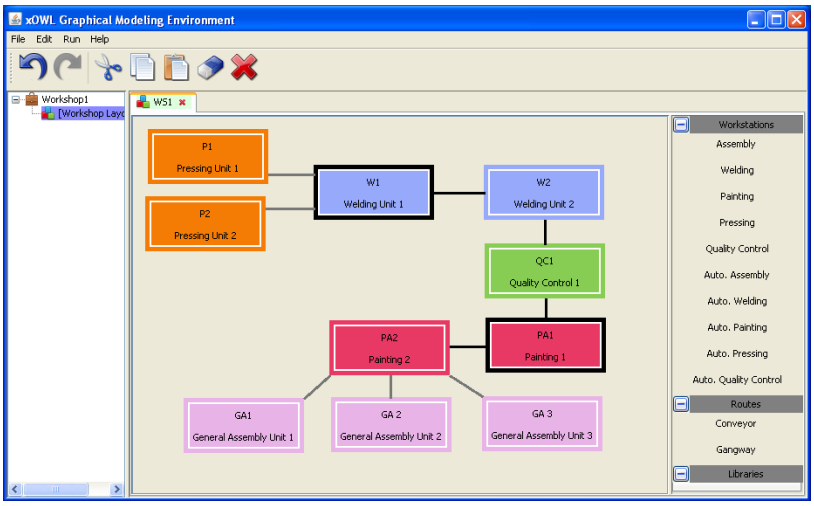

Figure 6. Visual notation for *Workshop Layout* experts

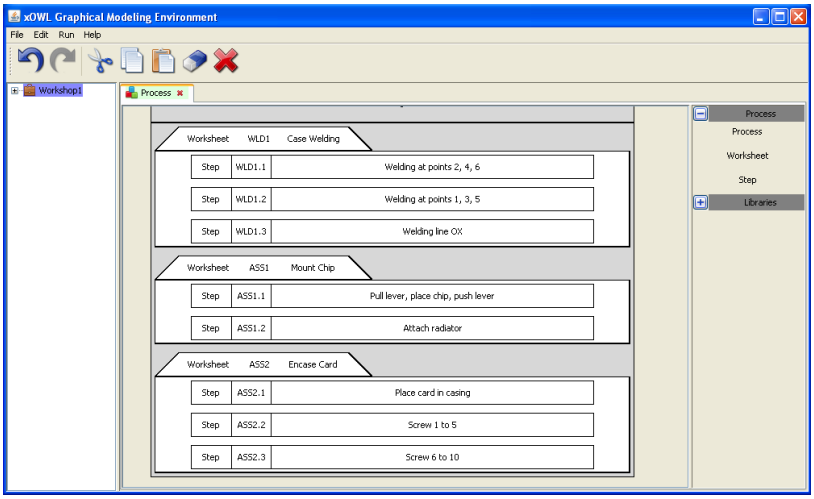

Figure 7. Visual notation for *Manufacturing Process* experts

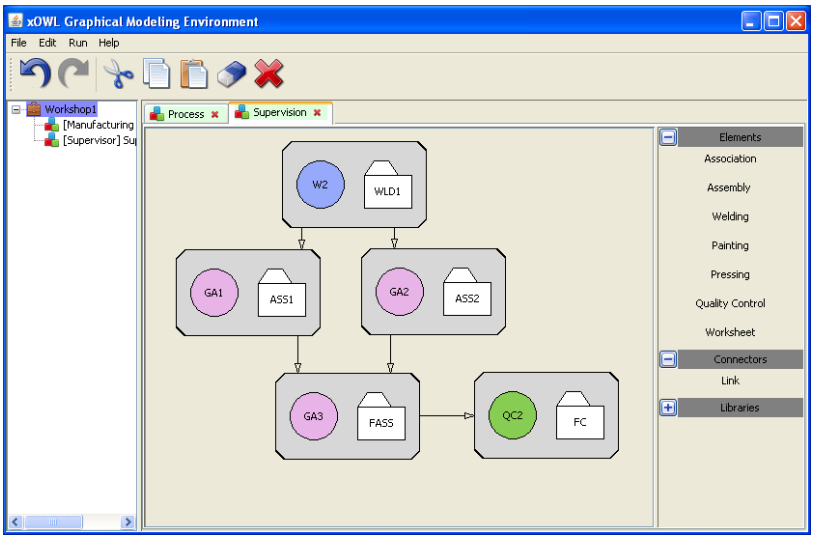

Figure 8. Visual notation for *Workshop Supervision* experts

that specify where materials can be exchanged. These `routes` are not directional and may be interpreted as physical `conveyors` or manually operated `wagons` in the real workshop. In this notation, the rule for transforming manned painting `workstations` in their corresponding notation elements is implemented as shown in Listing 6.

Listing 6. Implementation of the rule for manned painting workstations

```
1  Rule(:MannedPaintingWorkstation
2    Antecedents(
3      ClassAssertion(model:PaintingWorkstation ?w)
4      Not(ObjectPropertyAssertion(model:hasRobot ?w ?r))
5    )
6    Consequents(
7      ClassAssertion(view:Rectangle ?r)
8      ClassAssertion(view:Label ?lname)
9      ...
10   )
11 )
```

In this listing, a negative antecedent assertion is used at line 4 in order to check that the workstation is manned, or has no robot in this application case.

The *Manufacturing Process* experts represent `manufacturing steps` in a table, as shown in Fig. 7. Each `step` is affected a unique identifier and a description of the work to be executed. `Steps` are grouped together in order to form consistent activities, which are called `worksheets`. In this notation, the rules for transforming `worksheets` and `manufacturing steps` in their corresponding notation elements are implemented as shown in Listing 7.

Listing 7. Implementation of the rule for manufacturing steps

```
1  Rule(:WorksheetRule
2    Antecedents(
3      ClassAssertion(model:Worksheet ?ws)
4      ...
5    )
6    Consequents(
7      ClassAssertion(view:Container ?c)
8      ...
9      Meta(ObjectPropertyAssertion(meta:trace ?ws ?c))
10   )
11 )
12 Rule(:ManufacturingStepRule
13   Antecedents(
14     ClassAssertion(model:Step ?s)
15     ObjectPropertyAssertion(model:hasStep ?ws ?s)
16     Meta(ObjectPropertyAssertion(meta:trace ?ws ?c))
17   )
18   Consequents(
19     ClassAssertion(view:Container ?sc)
20     ...
21     ObjectPropertyAssertion(view:contains ?c ?qsc)
22     Meta(ObjectPropertyAssertion(meta:trace ?s ?sc))
23   )
24 )
```

In this listing, the first rule transforms `worksheets` in the model into their representations and makes traceability links between the `worksheets` and their representations. The second rule transforming `manufacturing steps` reuse these links in order to attach steps' representations to their respective parent worksheets' representations. The traceability links are stored in the meta ontology.

Last but not least, *Workshop Supervision* experts associate available `worksheets` to `workstations` and specify how they are orchestrated. As can be seen in Fig. 8, `workstations` are simply represented using a color code corresponding to their line of work. This time, the experts link the `workstations` with arrows, thus specifying the orchestration. An arrow means the work at the origin `workstation` is a prerequisite for the target `workstation`. Also shown in Fig. 8, `workstations` are affected particular `worksheets`, here simply represented by their ID and the recognizable icon. In this notation, the rule for transforming assembly `workstations` into their corresponding notation elements is implemented as shown in Listing 8.

Listing 8. Implementation of the rule for assembly workstations

```
1  Rule(:AssemblyWorkstation
2    Antecedents(
3      ClassAssertion(model:AssemblyWorkstation ?w)
4    )
5    Consequents(
6      ClassAssertion(view:Ellipse ?r)
7      ...
8      Meta(ObjectPropertyAssertion(meta:trace ?w ?r))
9    )
10 )
```

This industrial application shows how we leveraged ontology transformations in order to produce drawing primitives implementing different visual notations, based on the same data. As a final validation step, we conducted a study of the implementation's performances. The results are presented in the following section.

VII. Performances Study

This study aims at determining whether our approach and its implementation deliver sufficient performances in order for industrial customers to use it. The performances of the implementation is an important issue to industrial users because it determines the responsiveness of their application. Thus, in the example presented in the previous section, when experts open a particular notation, the common artifact will be transformed into drawing primitives. This is called the *initial transformation*. Experts will then interact with their notations, adding, removing and modifying elements, leading to modifications to the common artifacts. These modifications must be propagated to the respective notations through the same ontology transformation in order to given feedbacks to experts. This is called the *increment transformation*.

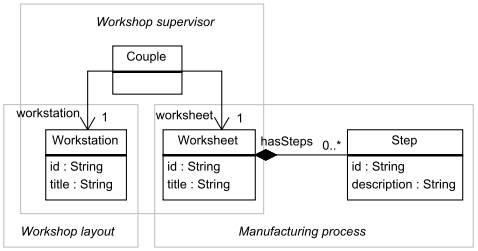

Figure 9. Domains of each transformation

In order to assess the performance of our approach and its implementation for both *initial* and *increment transformations* on realistic data, we used the three transformations from the industrial example presented hereabove. The transformation for the *Workshop Layout* notation is composed of 52 rules, the one for the *Manufacturing Process* notation is composed of 17 rules and finally the one for the *Workshop Supervision* notation is composed of 31 rules. As shown in Fig. 9, the three transformations operate on different but overlapping domains in the same global ontology. In this figure, the different kind of `workstations` are not represented but belong to the same domains as the `Workstation` class.

A. Performance of Initial Transformations

The performance for the three transformations have been tested on input ontologies of increasing sizes. A dataset of input ontologies have been automatically generated as follow: The n^{th} ontology will contain $n \times 10$ `workstations`, $n \times 10$ `worksheets`, $n \times 20$ `steps` (2 per `worksheet`) and $n \times 10$ `couples` associating a `workstation` to a `worksheet`. In this way, generated ontologies are balanced in that no particular transformation taking these as input is favored. They still are fairly representative of ontologies produced by human experts. The produced dataset contains 100 ontologies. The first of them contains 170 OWL2 axioms and the last contains 17000 axioms.

We then measured in 10 distinct experiments the transformation time of each input ontology with the three transformations. The results are aggregated in Fig. 10.

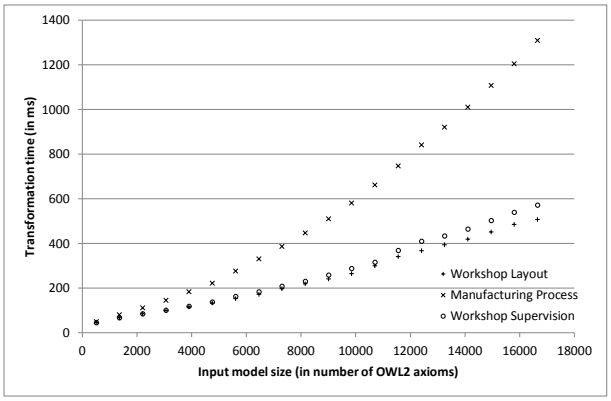

Figure 10. Initial transformation times by input size

We first observe that the transformation time for a given input ontology depends on the applied transformation. The number of rules in the transformation cannot explain alone this discrepancy. The worst performing transformation, the one for the *Manufacturing Process* notation, contains 17 rules, although the other two containing respectively 52 and 31 rules have nearly identical performances. This difference can be explained by the writing of the rule themselves. In the database domain, the writing of a SQL query can heavily impact the performance of the engine. The same phenomenon ought to be expected here because the matching of rule's antecedents is similar to querying a database.

Another interesting result is that the correlation coefficient between the size of the input ontology and the transformation time is around 0.99 for each transformation. This means that in the case of these transformations, the transformation time is close to a linear function of the input ontology's size.

B. Performance of Increment Transformations

The performance of increment transformations have been measured in a similar fashion. We hypothesize the increments, i.e. the modifications to the common artifact, are always small in regard to the input ontology. We then measured for each of the three transformations, the performance of a small increment transformation for input ontologies of various sizes. The previous dataset ontologies are reused in this purpose. For each transformation and ontology in the dataset, the input ontology is transformed and then the transformation time of a small increment is measured. The increments have been designed as follow: The increment for the *Workshop Layout* notation consists in a new `workstation`, expressed using 3 OWL2 axioms. The increment for the *Manufacturing Process* notation consists in a new `worksheet` and two new `steps`, expressed using 11 OWL2 axioms. The increment for the *Workshop Supervision* notation consists in a new `workstation`, a new `worksheet` and a new `couple`, expressed using 9 OWL2 axioms. The results are presented in Fig. 11.

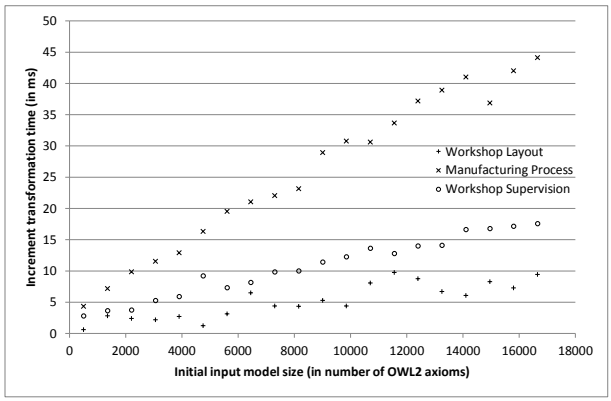

Figure 11. Increment transformation times by initial input size

We observe that the performance of the increment transformations depends primarily on the transformation's rule, as was the case with initial transformations. Also, the correlation coefficient between the size of the initial input ontology and the transformation time is at least 0.97 for the *Manufacturing Process* transformation. The performance of the increment

transformation is then close to a linear function of the initial input ontology's size.

The performances presented in this study demonstrate that ontology transformations can be executed at runtime in this case. Hence, the drawing primitives for each notation can be dynamically generated whenever an expert needs to access the common artifact through his notation. The increment transformation performances also show that the drawing primitives for a notation can be synchronized fast enough for an expert to modify the common artifact and immediately see the result. With these results, we validated our approach and its implementation.

VIII. Conclusion and Perspectives

In this paper, we proposed an OWL2-based rule language for the expression of ontology transformations. This approach has been validated by its implementation [2] and its application to an industrial case. However, some parts go beyond the scope of this paper. In particular, a second set of transformations describes how visual elements are mapped back to model elements.

In this approach, a domain-specific notation is defined through the expression of its corresponding transformation. Although valid, it is difficult to really design the notation itself only through the transformation. In addition, the notation alone does not define how domain experts can use it in their modeling tool integrating the notations. How this integration can be achieved is still to be investigated.

References

[1] L. Wouters and M.-P. Gervais, "xOWL an Executable Modeling Language for Domain Experts," in *EDOC*. IEEE, 2011, pp. 215–224.

[2] W3C, "OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax," Oct. 2009. [Online]. Available: http://www.w3.org/TR/owl2-syntax/

[3] K. Baclawski, M. Kokar, P. Kogut, L. Hart, J. Smith, W. Holmes, J. Letkowski, and M. Aronson, "Extending UML to Support Ontology Engineering for the Semantic Web," in *UML*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2001, vol. 2185, pp. 342–360.

[4] G. Kappel, E. Kapsammer, H. Kargl, G. Kramler, T. Reiter, W. Retschitzegger, W. Schwinger, and M. Wimmer, "Lifting Metamodels to Ontologies: A Step to the Semantic Integration of Modeling Languages," in *MoDELS*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2006, vol. 4199, pp. 528–542.

[5] C. Atkinson, M. Gutheil, and B. Kennel, "A Flexible Infrastructure for Multilevel Language Engineering," *IEEE Trans. Softw. Eng.*, vol. 35, 2009.

[6] OMG, *Meta Object Facility Query/View/Transformation Version 1.1*, http://www.omg.org/spec/QVT/1.1/, OMG Std., Jan. 2011.

[2] available at http://xowl.codeplex.com/

[7] L. M. Surhone, M. T. Timpledon, and S. F. Marseken, *SmartQVT*, V. P. House, Ed. VDM Publishing House, 2010.

[8] ikv++ technologies ag, "medini QVT," Jan. 2012. [Online]. Available: http://projects.ikv.de/qvt

[9] H. Giese and S. Hildebrandt, "Incremental Model Synchronization for Multiple Updates," in *Proceedings of the third international workshop on Graph and model transformation*. ACM, 2008.

[10] F. Jouault and I. Kurtev, "Transforming Models with ATL," in *MoDELS Satellite Events*, ser. Lecture Notes in Computer Science, vol. 3844. Springer Berlin / Heidelberg, 2006, pp. 128–138.

[11] J. Greenyer and E. Kindler, "Comparing Relational Model Transformation Technologies: Implementing Query/View/-Transformation with Triple Graph Grammars," *Software and System Modeling*, vol. 9, pp. 21–46, 2010.

[12] S. Roser and B. Bauer, "Ontology-Based Model Transformation," in *MoDELS Satellite Events*, ser. Lecture Notes in Computer Science, vol. 3844. Springer Berlin / Heidelberg, 2010, pp. 355–356.

[13] OMG, *Ontology Definition Metamodel*, http://www.omg.org/spec/ODM/1.0/, OMG Std., May 2009.

[14] K. Falkovych, M. Sabou, and H. Stuckenschmidt, "UML for the Semantic Web: Transformation-Based Approaches," in *Knowledge Transformation for the Semantic Web*. IOS Press, 2003, vol. 95, pp. 92–106.

[15] F. Silva Parreiras and S. Staab, "Using Ontologies with UML Class-Based Modeling: The TwoUse Approach," *Data & Knowledge Engineering*, vol. 69, no. 11, pp. 1194 – 1207, 2010.

[16] D. Djuric, D. Gasevic, and V. Devedzic, "Ontology Modeling and MDA," *Journal of Object Technology*, vol. 4, pp. 109–128, 2005.

[17] D. Gasevic, D. Djuric, V. Devedzic, and V. Damjanovi, "Converting UML to OWL Ontologies," in *Proceedings of the 13th international World Wide Web conference*. ACM, 2004, pp. 488–489.

[18] W3C, "SWRL: A Semantic Web Rule Language Combining OWL and RuleML," May 2004. [Online]. Available: http://www.w3.org/Submission/SWRL/

[19] I. Horrocks, P. F. Patel-Schneider, S. Bechhofer, and D. Tsarkov, "OWL Rrules: a Proposal and Prototype Implementation," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 3, pp. 23 – 40, 2005.

[20] C. L. Forgy, "Rete: a fast algorithm for the many pattern/many object pattern match problem," *Artificial Intelligence*, vol. 19, pp. 17 – 37, 1982.

[21] W3C, "OWL 2 Web Ontology Language RDF-Based Semantics," Mar. 2012. [Online]. Available: http://www.w3.org/TR/owl2-rdf-based-semantics/