Definition Methods and Implementation of Domain-Specific Modeling Language Tools

Renārs Liepiņš
Do Action1?

Yes -> Action 1

No -> Action 2
How to build such tools?
Do Action1?

Yes

Action 1

No

Action 2

Graph

Do Action1?

Yes

Action 1

No

Action 2
• Model Based Approach
- Model Based Approach
- Ontology Based Approach
Model Based Tool Building
Figure 2: The tool definition meta-model.
Language for Model Transformations
lQuery
Patterns would repeat, e.g. navigation through multiple link chains, or filtering by some condition. To make transformations more readable, the redundant parts need to be abstracted away. lQuery functions help to do it.

Example Model

In Figure 5 we can see a simple model and an instance diagram. We will use it throughout the rest of the chapter for demonstrating lQuery constructs. The model is on the left; it consists of two classes: Person and Animal. A person has name and age attributes and associations to other persons that are his parents and children, and an association to Animals that are his pets. On the right side, we can see some instances of this model.

Typical queries that we would like to make on this model are: get instances of a particular class (e.g. all persons), get instances with a particular attribute value (e.g. persons with name "John"), or get all pets of a person's children. If we needed to perform these queries using only the repository API, then the code would mostly contain...
One object → Collections of objects
Design

• Primitive Selectors
• Selector Combinators
• jQuery Selector Shorthand
obj_col
:find("Node /compartment [input = C2]")
:attr("input", "Class2")
Results

- Transformation Language for Tool Building
- Extensively validated
Model Based Tool Building – Results

- Fast & easy tool definition
- Multiple tools used in industry
Domain Specific Behavior

Types:
- FlowChart
- Start
- End
- Decision
- Action
- Flow

Styles:
- Colors
- Shapes
- Strokes
- Fonts
  - Helvetica Bold 12pt
  - Helvetica Regular 12pt

Graph:
- Do Action1?
  - Yes
  - Do Action1
  - No
    - Action 1
    - Action 2

Behavior:
- Universal
- Domain Specific
Domain Specific Behavior

• Dynamic styles

• Semantic validation
Basic Semantic Validation Example

Every activity diagram has exactly one start element.

Every activity diagram has exactly one end element.
Ontology Based Tool Building
What is Needed

• UML based notation and metamodel for OWL
What is Needed

• UML based notation and metamodel for OWL

• Integration of IQery with OWL
What is Needed

• UML based notation and metamodel for OWL
• Integration of IQuery with OWL
• Architecture
UML based notation and editor for OWL

- Metamodel extension
- Notation
- Editor
patterns would repeat, e.g. navigation through multiple link chains, or filtering by some condition. To make transformations more readable, the redundant parts need to be abstracted away. lQuery functions help to do it.

4.3 Example Model

In Figure 5 we can see a simple model and an instance diagram. We will use it throughout the rest of the chapter for demonstrating lQuery constructs. The model is on the left; it consists of two classes: Person and Animal. A person has name and age attributes and associations to other persons that are his parents and children, and an association to Animals that are his pets. On the right side, we can see some instances of this model.

Typical queries that we would like to make on this model are: get instances of a particular class (e.g. all persons), get instances with a particular attribute value (e.g. persons with name “John”), or get all pets of a person’s children. If we needed to perform these queries using only the repository API, then the code would mostly contain...
4.3 lquery

4.3.1 Example Model

In Figure 5 we can see a simple model and an instance diagram. We will use it throughout the rest of the chapter for demonstrating lQuery constructs. The model is on the left; it consists of two classes: Person and Animal. A person has name and age attributes and associations to other persons that are his parents and children, and an association to Animals that are his pets. On the right side, we can see some instances of this model.

Typical queries that we would like to make on this model are: get instances of a particular class (e.g. all persons), get instances with a particular attribute value (e.g. persons with name "John"), or get all pets of a person's children. If we needed to perform these queries using only the repository API, then the code would mostly contain...
The image shows a UML diagram of a metamodel, which is an extension of the core metamodel. The diagram includes various classes such as `NamedElement`, `DataType`, `DataProperty`, `ObjectProperty`, `Class`, `Generalization`, and `EnumExpr`, among others. Each class is represented with attributes and relationships, such as `name` for `NamedElement` and `range` for `DataProperty`. The diagram is labeled with extensions and variations, indicating the model's complexity and the relationships between different elements.
3.1. Equivalent and disjoint classes and properties

The simplest extensions to the UML metamodel are equivalent and disjoint classes and properties which are introduced in the extended UML by `eqClass` and `disjClass` relations from UML `Class` to UML `Class` and `eqProperty` and `disjProperty` relations from UML `Property` to UML `Property`.

The OWL class equivalence is modeled by the `eqClass` relation, and it can be visually represented in the diagram in two ways – either as a connector with `<<equivariant>>` stereotype linking two classes, or as a note symbol with `<<equivariant>>` stereotype connected to all equivalent classes. The OWL class disjointness is modeled by the `disjClass` relation, and it can be visually represented in the diagram either as a connector with `<<disjoint>>` stereotype linking two classes, or as a note symbol with `<<disjoint>>` stereotype connected to all disjoint classes (see Figure 5 where the disjointness of `Person`, `Level` and `Course` classes is asserted). We note that the class disjointness can be asserted also by means of attaching a `disjoint` tag to the `Generalization` set already present in the original `UMLOWLCore` metamodel. There are other options available for denoting the class equivalence and disjointness using class expression notion that is explained later.

Fig. 5 A mini-university ontology (UMLOWLCoreExtended notation)
The editor for the proposed notation – owlgrEd – joint, then it is preferable to group the subclass relations visually with a "fork" symbol that possesses the disjoint label. This is a much more compact representation than the alternative notation where each subclass line is by itself, and there are explicit disjoint labeled edges between all subclasses. By using the visual refactoring, the graphical reorganization can be done with one click.

Automatic layout and search facilities are crucial when ontologies become large (more than 100s of classes), and their management becomes more difficult. A good automatic layout is significant for understanding large ontologies. Also, searching for the specific element in large ontologies may become irritating without an appropriate service. Therefore several alternative automatic layout modes and searching mechanism allowing finding the necessary element by the value of one of its text fields (e.g. searching a class by its name) is supported in our editor.

With Protégé. The editor is built using TDA [1 4, 1 5] technology. Figure 6 shows an African Wildlife ontology [1 6] in our editor.

Graphical refactoring is one of the most important services that allow modifying graphical notation without changing semantics as long as the same concept can be expressed through different constructs. This feature allows the user to choose the most compact graphical format depending on the context and taste. One of the typical situations illustrating the need for graphical refactoring is generalization and fork: if there is a single super class with multiple incoming generalization lines, a fork can be added to reduce multiple lines into a single line, and vice versa.

Automatic layout and search facilities are crucial when ontologies become large and their management becomes more difficult. A good automatic layout is significant for understanding large ontologies, whereas searching for the specific element in large ontologies may become irritating without an appropriate service. Therefore several alternative automatic layout modes and searching mechanism allowing finding the necessary element by the value of one of its text fields, e.g. searching a class by its name is supported in our editor.

A more advanced service is full interoperability with Protégé [9], a tool that is widely used by ontology developers. The interoperability is implemented via a custom Protégé plug-in that allows to send and receive (via TCP/IP socket) an active ontology between our editor and Protégé. In both directions ontologies are sent in interchange format, but generally any OWL serialization is acceptable. Interoperability allows ontology developers to use Protégé without changing their habits and only afterwards to
Figure 13: OWLGrEd downloads by cities outside Latvia from Nov 1st, 2013 till May 31st, 2015.

That this familiarity would enable them to adopt easily the new formality we propose.

The application of UML class diagram notation to OWL is not an entirely new idea; it has been implemented in the TopBraid Composer\cite{15}. However, that implementation is based on a simplified UML class diagram model, it lacks graphical editing facilities, and the available graphical services are limited. Some other solutions have been proposed for the graphical UML-style representation of OWL ontologies; the most notable is ODM (see \cite{6}, Chapter 14) that defines a UML profile for OWL. The main advantage of ODM approach is the possibility to use existing UML tools for ontology modeling. Meanwhile, the price for this compatibility is a more verbose notation that does not facilitate comprehensibility.

Conclusions
In this chapter, we created an OWL metamodel that is a constraint layer above UML class diagram metamodel. This will enable transformation languages to work simultaneously with OWL constraints. We also developed a graphical notation for OWL ontologies that is an
Every activity diagram has exactly one start element.
Every activity diagram has exactly one start element.
What is needed

✓ UML based notation and metamodel for OWL

→ Integration of IQery with OWL

• Architecture
Extending OWL with lQuery selectors

- Metamodel extension
- Graphical notation extension
- Integration with semantic reasoners
Figure 10: UML OWL metamodel. Classes in bright yellow are equivalent to the classes shown in Figure 4.
The lQuery language also needs to be integrated with the OWL metamodel from the previous chapter. The metamodel is extended with a class `lQueryExpression`, that represents an lQuery selector expression. There is a link `lQueryEquals` from the intuitive class `Class` to the class `lQueryExpression`. This link means that the intuitive class contains all the instances that are returned by the connected lQuery selector, when it is executed on the entire repository. The extended metamodel is shown in Figure 16.
information that is directly known and assumes that everything that is not known is

InactiveStudent

have a link

InactiveStudents

expression that describes that class, let us recall that the OWL definition of class

NoKnownCourses

were not able to define using OWL class expressions. First let us return to the

graphically, we will look at some examples from the University Ontology that we

3.2 Some Examples from the University Ontology

Now that we have seen a survey of the lQuery expressions and how they are shown

Fig.

2

Annotations:

lQuery expressions as annotation properties in Manchester syntax

Class: PassedStudent

----------

Datatype: lQueryExpression

rdfs:comment "Students that have earned at least 20 credit points"

lQueryEquals "Student:has(/grade/course@creditPoints:sum() >= 20)"^^lQueryExpression

The lQuery selectors work only with the

In-ActiveStudent

ActiveStudent

could possess a link or not. Therefore, it is not classified as either

StudentWith-NoKnownCourses

– “not (takes some Course)” – does not describe the instances we want, because the class

– the individual

Course)" – does not describe the instances we want, because the class

In-ActiveStudent

ActiveStudent

could possess a link or not. Therefore it is not classified as either

StudentWith-NoKnownCourses

– “not (takes some Course)” – does not describe the instances we want, because the class

In-ActiveStudent

ActiveStudent

could possess a link or not. Therefore it is not classified as either

StudentWith-NoKnownCourses

– “not (takes some Course)” – does not describe the instances we want, because the class

In-ActiveStudent

ActiveStudent

could possess a link or not. Therefore it is not classified as either

StudentWith-NoKnownCourses

– “not (takes some Course)” – does not describe the instances we want, because the class
Integration Algorithm

1. Classify with OWL & materialize
2. Classify with lQuery & materialize
3. If new inferences
   goto step 1
else
   finish
Activity diagram without a start element is red. Activity diagram with a start element is white.
Activity diagram without a start element is **red**.
Activity diagram with a start element is **white**.
What is needed

✓ UML based notation and metamodel for OWL
✓ Integration of IQuery with OWL

➡ Architecture
Engine 1

Engine 2

Model Transformations

TDA Core

RAAPI
TDA Core

Engine 1

Engine 2

Model Transformations

RAAPI

TDA Core

OWL with lQuery Extension MM

OWL reasoner

IQuery runtime

Old Model Repository

OWL with IQuery Extension MM
Tool Definition Metamodel – Logical View

- Diagram
- Element

Tool Definition Metamodel – Internal Representation

- Class
- ObjectProperty
  - element:ObjectProperty
  - Diagram:Class

New Model Repository

OWL reasoner
lQuery runtime

RAAPI

Logical view
rep in internal view
Every activity diagram has exactly one start element.

Activity diagram without a start element is red.

Activity diagram with a start element is white.
Every activity diagram has exactly one start element.

Activity diagram without a start element is red.

Activity diagram with a start element is white.
Every activity diagram has exactly one start element. Activity diagram without a start element is red. Activity diagram with a start element is white.
Summary
Summary

• Model Based Tool Building

• Transformation Language: IQuery
Summary

- Model Based Tool Building
  - Transformation Language: lQuery
- Ontology Based Tool Building
  - UML based model and notation for OWL
  - OWL extension with lQuery selectors
  - Architecture
Do Action 1?

Yes → Action 1
No → Action 2
Thank you!