DL-workbench: a meta-modeling approach to ontology manipulation

Mikhail KAZAKOV[1, 2] Habib ABDULRAB[2]

[1] Research division, Open Cascade S.A. 91400, Saclay, France
mikhail.kazakov@opencascade.com
[2] PSI laboratoire, INSA de Rouen, BP 8, 76131, Mont Saint Aignan, France
abdulrab@insa-rouen.fr

Nowadays many ontological editors are available. However, several specific requirements forced us to develop a new ontology edition platform which we call DL-workbench. DL-workbench contains three main modules. First module defines a meta-model for description of ontological formalisms. It provides an API that allows management of ontological entities, containers of entities, and many other features that are useful when one needs to use an ontological model within its project. A second module of DL-workbench is a formalism-independent processing module with integrated GUI for edition of elements based on the meta-model. This module uses the meta-model and is implemented as a plug-in to the IBM Eclipse platform. A third module defines the SHIQ description logic formalism using the meta-model. The third module customizes also the user interface (images, etc.). DAML+OIL is used within this module as a persistent format. DL-workbench pays much attention to edition of logical equations and to management of a group of ontologies within a project. DL-workbench allows easy integration of ontological model with other data inside one standalone or distributed application. DL-workbench can be used both as the ontological editor and as an ontology manipulation platform integrated with other tools and environments. This paper describes our motivation for creation of DL-workbench, implemented features and lessons learned from implementation of ontological editor.

1. Introduction

Ontologies have become an increasingly important research topic. This is a result of their usefulness in many application domains (software engineering, databases, medical domain, conceptual modeling, etc.) and the role they will play in the development of emerging Semantic Web activity. It is clear that development and manipulation of ontologies have to be supported by corresponding software tools (annotation tools, editors, reasoners, etc.) and standards (OWL [9], etc.).

Nowadays many application domains are looking for use of ontologies. Each domain, that uses ontologies, sets its own requirements for tools. One of such domains is software integration. The main challenge of software integration is: how to make working together software entities that were not initially created to work with each other. We are currently working on the topic of semi-automated integration of

various numerical simulation multi-physics solvers [2]. Here the use of ontologies (as formal description models) helps to share a common view on specification of solvers that come from different vendors. Integration topic is out of the scope of this article and will be published in a separate paper. Some preliminary results are given in [16].

Since we use logical models in our specific domain, we need a formalism that will be used for creation of these models and tools that support the formalism (creation of ontologies and reasoning). Also we need an API that allows us to integrate all these tools with software, specific to the integration domain. Taking into account the research origin of our work, we have formulated a set of requirements for an ontological tool that would be convenient for us (the same requirements arise often within the variety of other domains):

- Full support of at least one of ontological formalisms. Several formalisms are preferred for research purpose. During the work on our research project dedicated to semi-automated integration we experimented with several ontological formalism and at the moment of DL-workbench creation we were not sure, whether description logics are appropriate or not for our research.
- Convenient user interface to work with complex logical expressions (with ability to modify the structure of expressions via the graphical user interface). While trying several ontological editors, we found that most of the ontological editors don't implement expression editors. Those who implement them were not convenient from our point of view.
- Presence of reasoner connection for implemented formalisms. Within our research we have to work with complex logical axioms and ontological models. These require the presence of reasoner both for validation of ontologies and for performing of additional reasoning tasks that are dedicated to semi-automated integration of software components.
- Ability to integrate that tool into a specific domain environment. Our research is dedicated to integration environments that already exist as prerequisite. The next requirement is coming from the same origin.
- Ability to manipulate ontologies and their elements from specific domain environment. We need that in order to merge several ontological formalisms within the same environment. For example the ontological data is merged with specifications of Java interfaces.
- Ability to work with structured "ontological projects" but not with "files". Following our methodology of semi-automated integration we need to work with several separated ontological files at the same time. Managing of these files within the "project"-like environment seems to be an acceptable solution. Most of the people in software engineering who will use our products are familiar with this concept.
- Fast and portable user interface and presence of extension points

Studying the state of the art, we did not find any tool that could comply with our requirements. Thus the decision to create own platform was taken. The name of the platform is DL-workbench (short of Description Logic Workbench, since SHIQ description logic [7] is the main formalism that is used). DL-workbench is published now under open source license (http://www.opencascade.org/dl-workbench).

Semantic Web activity is an important world-wide effort that combines many techniques and touches many application domains. We hope that our experience in creating of software tools for manipulation of ontologies (and DL-workbench itself) will be useful for the semantic web community. This paper describes a meta-model approach that was used for creation of Dl-workbench. First we give the concept of DL-workbench and our motivations. Further, we describe the meta-model kernel and other modules. At the end we share some of lessons learned during creation of this tool.

2. DL-workbench conception

The next list of principles and it constitutes the main concept of DL-workbench:

- DL-workbench is based on a meta-model that is capable to describe the structure of ontological formalisms and data that is used within a specific domain.
- DL-workbench has a modular (plug-in based) architecture with clearly specified dependencies among modules.
- Main processing module of DL-workbench is based only on the meta-model and does not depend on any of specific formalisms. This module implements features such as persistence skeleton, tracking of changes, transactions[1], lifecycle of instances and many others.
- Each generic module provides a set of documented extension points that can be used by other modules / software to customize the platform.
- The work with ontologies is performed using a notion of a project. A project here is a structured set of ontological files (ontological resources) and other domain specific data if needed.
- DL-workbench defines an internal data model and "UI-ready" data model. That allows organization of different views on the same data (i.e. project view, namespace view, taxonomy view)
- DL-workbench supports manipulation/edition of complex expressions and axioms. In many application domains, the complete and "reasoning-able" ontologies require the use of logical expressions and axioms.
- DL-workbench provides the user interface of an ontological editor.

We strongly believe that an ontological manipulation platform shall be based on these principles to be interoperable and useful for researchers, software architects and end users. DL-workbench source code is public and we hope it will be useful for vendors of ontology edition tools.

DL-workbench can be viewed both as a meta-model based platform for creating ontology-manipulation tools and as an ontological editor that supports SHIQ description logic (i.e. the meta-model is used to define SHIQ model). DL-workbench

[1] Transaction support is not implemented in the current version of DL-workbench

uses DAML+OIL[2] [8] as persistent format and Racer [11] as DL reasoner. DL-workbench is implemented as a set of plug-ins to IBM Eclipse platform [10]. Eclipse is an emerging open source environment for creation of project-based tools.

3. DL-workbench structure

Meta-model

The major advantage of the DL-workbench is its meta-model based architecture. It allows easy-to-use definition of entities and relations of an ontological formalism that has to be used within the workbench. The meta-model is implemented as a set of Java interfaces for the convenience of use from programming environments. Java programming is involved for instantiation of the meta-model. This is an explicit design choice. We benefit of existing type checking system and absence of another compiler. That makes the objects instantiated from meta-objects working very fast. The power of Java language can be used for definition of behavioral parts of the meta-model. Moreover most of the people working in integration area are familiar with Java.

A meta-model is a language that is used for description of ontological formalisms by specifying their elements, structure and invariants (invariants have to be satisfied when instances of these elements are created or modified). For example: one needs to work with a simple propositional logic that includes notions of "atomic proposition" and "composite proposition" expressed via logical expression with conjunction, disjunction and negation operands.

Our meta-model is implemented as a light module and is independent from Eclipse or any other tool. The main goal of the module is to achieve the maximal level of independence from specific formalism and to enable implementing generic software features (object lifecycle, transactions, etc.). The module also helps achieving the interoperability among different tools by sharing the same meta-model.

The meta-model is basically intended for specifying structural models. Semantics of the meta-model were inspired by Description Logics [1].We tried to keep the meta-model as simple as possible, but powerful enough for many possible needs.

The main element of meta-model is a meta-concept (`IMetaConcept` Java interface). It represents any typed element with a set of properties. For example: "Expression", "proposition", "atomic proposition", "logical operand" are the instances of meta-concept. Each meta-concept has a meta-name that is represented in a form of java interface. The taxonomy of these Java interfaces defines the taxonomy (i.e. subsumption relationships) of corresponded meta-concept instances. For example: an "atomic proposition" is a "proposition" and an "expression" with atomic propositions is also a "proposition".

---

[2] We consider using the OWL language [9] instead of DAML+OIL as soon as OWL parsers will appear on the market.

Another important element of a meta-model is a meta-property (`IMetaProperty` Java interface). A meta-property represents a property that can potentially restrict a meta-concept (similar to description logic semantics of "property"). For example, the "atomic proposition" concept has "name" property; an "expression" concept has "operand", "left part" and "right part" properties. Meta-property also has a meta-name that is represented in the same way as for meta-concepts (i.e. in the form of Java interfaces). Each meta-concept instance may have a set of meta-property instances as its definition. An instance of meta-concept that inherits other meta-concepts also takes all their properties. Generally, the most specific meta-concept is restricted by the transitive closure of all the properties

Any meta-property has the notion of domain restriction, range restriction, inverse property restriction and transitivity. Semantics of all these notions (meta-concept, meta-property, domain, range, etc.) are very close to semantics of corresponded elements from description logics [7].

In addition to these semantics, each meta-property can be augmented with a pre-post processing handler for checking the invariant conditions of given property values. For example: the "name" property of the "proposition" concept has a "String" type. It must be complaint with the URI specification (i.e. no spaces, quotes, etc). An invariant handler that does such check can be written for the "name" property.

The meta-model includes a simple type system. Types are used to define meta-property range restrictions and to type instances of meta-model elements. Type system includes singular types and collection types. A singular type can be primitive (String, Boolean, Integer, Float, and Enumeration) or a meta-concept instance. Collection type is defined by the type of its elements (any other type).

In order to specify some logical model (formalism), meta-concept and meta-property classes must be instantiated. A special meta-model factory is implemented within DL-workbench to facilitate this task. It is worth to mention, that many formalisms can exist at the same time, can share their definitions and can be searched for specific meta-concepts and meta-properties.

Here it's necessary to mention, that three meta-modeling levels exist. `IMetaConcept` interface carries the notion of "meta-concept", its Java instance is a specific instance of the "meta-concept" (model concept) and `IModelInstance` interface specify an instance of the model concept (i.e. model instance). In order to instantiate the formalism, the notion of value is defined (`IValue` interface). Every primitive type and collection has its value object (`IPrimitiveValue`, `IModelCollection`). Every instance of meta-concept within some formalism can have many model instances represented by instances of `IModelInstance` interface.

Each model instance has its type represented by a meta-concept instance and a set of values according to properties of the model concept. For example: a model instance of "expression" meta-concept can be created with three values of their properties:

- "left part": model instance of type : "atomic prop." :"name" = "MARY" : String
- "operand" : model instance of type : "conjunction"
- "right part" : model instance : type "atomic prop." : "name" = "PETER": String

As we can see the instances of the above mentioned simple propositional formalism are expressed in terms of meta-model. The structure of the ontological formalism can be easily traversed by any application. Consequently, an ontological editor that supports only the notion of (concept – property – value) triple can be easily

implemented. That is extremely important for specific application domains, where the ontological information must be included within some other application.

The last element of meta-model is a "Container". Container is a collection of model instances or other containers that supports basic operations of addition, removal, iteration, checking of containment and size request. Creation of dynamic groups of elements is useful when sending information to reasoner, saving sub-ontology, classifying elements, etc.

We use the same meta-model for description of Java interfaces within the software integration domain. The meta-model is generic enough while having rich semantics. Many structural data formalisms can be easily expressed in the presented meta-model.

The meta-model kernel has a small abstract model expressed by means of meta-model. It contains the most useful semantic such as named object, namespace, generic expression and many others. We believe that this model is useful for expressing different formalisms. The DL-workbench meta-model documentation [5] may be consulted for further information.

The meta-model kernel publishes an API that allows defining and manipulating the meta-model instances (models), instantiation of these models, manipulation of instances of models (for example ontological elements) and many other features.

Processing module

A generic model processing module implements manipulation and edition of ontologies. This module is integrated with Eclipse workspace and depends only on the meta-model module. Eclipse framework provides us with the project-oriented workspace. The framework enables transparent connection to many environments of software integration domain (IBM Web Sphere, some UML tools, etc.).

The processing module implements the "view" and "controller" concepts according to the Model-View-Controller paradigm. Here the "model" concept is represented by a formalism that is an instance of the meta-model. "Controller" can be viewer as the UI operations. "View" data model is built on top of meta-model and enables many representations for specified formalism. An ontological project can be viewed by default as:

- a structured set of persisted ontologies (files)
- a set of namespaces (in case if formalism supports namespace)
- each element of some formalism is presented by a tree including properties of corresponded meta-elements and their typed values

The processing module defines all the generic UI operations for lifecycle of model instances independently of the formalism used. Processing module generates user interface controls following the structure of a given model instance. For example: the "expression" meta-concept instance has two properties of type "proposition" and one property of type enumeration. In this case, when a user asks for edition of an instance of "expression", three groups of controls will be generated independently on the end-user semantics of "expression". This principle works for any operation within the module. User interface elements are created only once for each type of elements

and cashed to reduce unnecessary OS interactions. Each module, dependent on this one, must specify the formalism itself and its UI resources (icons, names, order, etc.). Several concurrent/joint formalisms can be also supported.

The processing module publishes an API for manipulation and configuration of the user interface. It has an UI ready API that encapsulates instances coming from meta-modeling kernel allowing their visual presentation. Further details of implementation, extension points and API can be found in [5].

SHIQ module

SHIQ module implements a model of SHIQDn_ description logic formalism [7] that is based on the meta-model, implements DAML+OIL reader and writer and defines DIG interface [4] connection for the solver. The implementation of SHIQ formalism can be found in [5] and is not repeated in this paper. The module implements an additional view – taxonomical view: Selecting of any SHIQ "concept" or "object property" concepts within the Eclipse workspace causes the dynamic building of the taxonomy tree for this concept/property. The view is shown by default on the right edge of the Eclipse window within the DL-workbench perspective. The taxonomy is dynamically rebuilt using "subClassOf" and "sameAs" properties of the SHIQ "concept"/"properties". SHIQ module (as any formalism specific module) also specifies a set of Eclipse extensions: association of *.daml files with the plug-in, association of specific icons with menu items and others UI features. Integration with Eclipse is described in DL-workbench documentation [5].

As one can observe, the inclusion of specific formalism and customization of the user interface represent a very small part of the ontological editor code. SHIQ logic formalism was described via the meta-model within one working day. GUI customization was done within one week. The F-logic formalism was prototyped as an example during several hours using the same meta-model and benefits all the editor features (F-logic module is not published with DL-workbench due to incompleteness of GUI customization and absence of persistent format connection.).

Racer reasoner [11] is used via DIG interface. We choose Racer due to its support of ABox reasoning. However, the DL-workbench itself uses a reasoner only for satisfability and subsumption checks, thus FaCT [12] or any other DIG-complaint reasoner can be used. DAML+OIL reading support is done with the help of Jena DAML parser [17]. DAML+OIL writing was done via Xerces XML parser. Since DAML+OIL is a superset of RDFS, RDFS files can be also read by DL-workbench.

More details on the user interface and use of DL-workbench as ontological editor can be taken from [5]. The product itself and its source code can be downloaded following link in [5].

4. Lessons learned

In this section we'd like to indicate some positive experience and observations that were received during the creation of DL-workbench and use of ontologies for a specific application domain.

Axioms and logical expression are extremely important for creating of complete and reasoning-ready ontologies. However, it's extremely difficult to have a convenient user interface (GUI) for their edition. We've implemented our own GUI of expression editor; however we strongly believe that some deep research must be conducted on ergonomics of expression editor. It can be something between text input with dynamic compilation and GUI-based editor that chooses elements from lists. Current implementation of the expression editor is based on the same principle as other editors of DL-workbench: editor of "Expression" entity with a set of properties such as "left part", "right part" and "operand". Expression entities can be nested (i.e. "left part" can be also an expression). The rules of expression building are defined in the form of invariants and pre-post conditions in a generic model from meta-modeling module. We came from considerations that the given structure of expressions is common for most of the possible ontological formalisms. End user has always the possibility not to use the presented model or replace it according to its needs. The correctness of expressions is also supported by invariants and all structurally inconsistent equations are highlighted for the user. The processing module recognize "equation" as a special type and provide light edition mechanism using drag and drop and dynamic management of lists of possible elements. Same mechanism tries to assure that nested equations are not lost when containing structure is modified within the top level editor (loss happens sometimes in OilEd equation editor).

From our point of view, working with ontologies must follow the project-oriented paradigm. It's hard to imagine a real industrial ontology that is saved in one file and has no references to other files. The use of URI as a physical location of imported ontology is not always suited for industrial use due to possible unavailability of some URI at some time. Here the notion of project as a complete set of needed ontological resources can facilitate manipulation of ontologies. It can clearly separate a physical structure of files from logical structure of ontologies (i.e. namespaces, taxonomies). However it is worth to mention that needs of Semantic Web can be different.

Presence of meta-model for implementation of ontological formalism and connection with other data structures is very important. Above we said many things about these benefits.

The ability to have many different views (by namespaces, by taxonomies, by files, graphical view[3] and many others) on the same ontological structure helps a lot in many real cases.

Support of several formalisms and several GUI views is extremely important since it allows creation of different views for different groups of users on the same domain data and its ontological semantics. For example the same ontology may be presented by two formalisms with different expressivity to different groups of people.

We found it useful to introduce several macro-semantics into the SHIQ editor that are computed from basic SHIQ semantics[4]:

- XOR, IMPLIES and EQUALS logical operators can be easily introduced into any expression. These operands are easily convertible into AND/OR/NOT sequences and vice versa. That adds more high level semantics to the user.

[3] Graphical view is not provided in current version of the DL-workbench

[4] These macro-commands are excluded from the first open source version of DL-workbench

- In the same way: “class or equivalents” or “class of disjoints” elements can be defined on top of basic SHIQ axioms. When writing/reading to DAML+OIL corresponded transformations are performed.

Easiness of integration of ontological model / ontological edition user interface is crucial when ontologies are used within some application domains. There is an evident help from modern frameworks such as Eclipse and use of component technologies. Especially this is important, when research projects with sharp time frames are conducted.

By developing DL-workbench we have achieved all the requirements that were described at the beginning of this paper. Our current research for software integration is based on DL-workbench. We use described concepts for creation of extensions of DL-workbench that facilitate our experiments with integration of numerical solvers and creation of “good enough” ontologies verified by reasoner.

5. Other editors and APIs

Many ideas of user interface were inspired by OilEd [6] ontological editor. OilEd is the first editor that implements most of the features of SHIQ description logic, reasoner connection and expression edition. That was an ontological editor we used before creating DL-workbench. However, despite of all its benefits, some elements of user interface, such as choice among “subClassOf” and “sameClassAs”, semantic of some axioms and some others elements are not always clear for the end user. We tried to resolve these issues in DL-workbench. OilEd is an open source project, but OilEd API seemed to us difficult to integrate with other tools. Presence of meta-model level within the DL-workbench gives more flexibility and ease of use together with other tools.

WebODE project [3] is an ontological workbench that provides various ontological services. The project has a highly flexible architecture and provides many viewpoints on data. The meta-modeling approach chosen by DL-workbench allows on-fly changing of ontological formalism and easy integration with non-ontological data structures. In addition, DL-workbench is an open source project.

Protege [18] ontological editor has a convenient plug-ins API for its extension. However our intention was to integrate an ontological editor as a plug-in into software development tools but not vice-versa.

KAON API and a set of related KAON tools [14] define a distributed ontology manipulation infrastructure that is based on client-server architecture and provides many useful features. KAON has a hard coded API for its ontological formalism, that is mostly RDFS based and doesn’t support extended semantics of equations nor very expressive description logics (such as SHIQ). We found the OI-modeler ontological editor of KAON to be difficult for the end user.

Many other ontological editors are present nowadays. Due to the absence of space in the paper we can’t compare many tools. [13] is a good survey of ontological tools. The deliverable 1.3 [19] of OntoWeb project gives a comprehensive comparison of tools.

6. Conclusions and future plans

We have presented DL-workbench, both an Eclipse-based ontological editor for SHIQ logic and a meta-model based platform for manipulation of ontologies in conjunction with other tools. We've shown the benefits to use meta-model for creating of ontology-based products especially when working within specific application domains.

Today the DL-workbench is a research prototype and it lacks the stability that is needed for industrial development of ontologies. It lacks the functionality of merging and aligning of ontologies and more extensive support of reasoners features on the user interface level. All of that is planned to be improved soon. The user interface ergonomics and usability study is required.

We use DL-workbench for development of our domain specific extensions and integration with other tools. That assures the constant evolution and support of the DL-workbench. In the future we plan to introduce a transaction mechanism with undo-redo operations, merging/alignment of ontologies, graphical representation of ontological information and make many other improvements.

References

1. F. Baader et all, "*The Description Logic Handbook: theory, implementation and applications*", Cambridge University Press, 2003 ISBN 0-521-78176-0
2. The SALOME project, Online: http://www.opencascade.org/salome
3. WebODE project, Online : http://delicias.dia.fi.upm.es/webODE
4. S. Bechhofer, "The DIG description logic interface: DIG/1.0", 2002, Online: http://www.fh-wedel.de/~mo/racer/interface1.0.pdf
5. DL-workbench project web site. Online: http://www.opencascade.org/dl-workbench
6. S. Bechhofer et all, "*OilEd: a Reason-able Ontology Editor for the Semantic Web*", Springfied-Verlag, LNCS, 2001
7. I. Horrocks, lU. Sattler, and S. Tobies. "*Reasoning with individuals for the description logic SHIQ"*. LNAI number 1831 pp. 482-496. Springer-Verlag, 2000
8. DAML+OIL language, Online: http://www.w3.org/TR/daml+oil-reference
9. OWL language, Online: http://www.w3.org/TR/owl-absyn
10. IBM Eclipse 2.1 platform, project page, Online: http://www.eclipse.org
11. Racer reasoner. http://www.fh-wedel.de/~mo/racer
12. FaCT reasoner, http://www.cs.man.ac.uk/~horrocks/FaCT
13. M. Denny, "*Table 1. Ontology editor survey results*", 2002, Online: http://www.xml.com/2002/11/06/Ontology_Editor_Survey.html
14. KAON API, Online: http://km.aifb.uni-karlsruhe.de/kaon/Members/rvo/kaon_api
15. OMG MOF repository specification, Online: http://www.omg.org/technology/documents/formal/mof.htm
16. M. Kazakov, H. Abdulrab, E. Babkin, "*Intelligent integration of distributed components: Ontology Fusion approach*", In proceedings of CIMCA 2003 conference, 2003, ISBN 1-740-88069-2
17. Jena DAML and RDF parser, Online: http://www.hpl.hp.com/semweb/index.html
18. Prot•g• environment, Online: http://protege.standord.edu
19. OntoWeb project, "Deliverable 1.3 report", Online: http://www.ontoweb.org