The Semantic Web as a Software Modeling Tool: An Application to Citizen Relationship Management

Borislav Iordanov, Assia Alexandrova, Syed Abbas, Thomas Hilpold, and Phani Upadrasta

Miami-Dade County, Community Information And Outreach Department, Florida, USA
{boris,assia,sabbas,hilpold,phani}@miamidade.gov
http://www.sharegov.org

Abstract. The choice of a modeling language in software engineering is traditionally restricted to the tools and meta-models invented specifically for that purpose. On the other hand, semantic web standards are intended mainly for modeling data, to be consumed or produced by software. However, both spaces share enough commonality to warrant an attempt at a unified solution. In this paper, we describe our experience using Web Ontology Language (OWL) as the language for Model-Driven Development (MDD). We argue that there are benefits of using OWL to formally describe both data and software within an integrated modeling approach by showcasing an e-Government platform that we have built for citizen relationship management. We describe the platform architecture, development process and model enactment. In addition, we explain some of the limitations of OWL as an MDD formalism as well as the shortcomings of current tools and suggest practical ways to overcome them.

Keywords: semantic web, owl, model-driven development, e-government, live system, executable models

1 Introduction

1.1 Motivation

The promise of model-driven development (MDD) is justifiably attractive for any organization supporting its constantly evolving business processes with software that must adapt as quickly and in a cost effective way. In adopting an MDD approach one would expect business changes to be absorbed more easily, with much fewer code changes. E-Government aims at replacing a heavy bureaucracy and manual paperwork with lean software for improved efficiency of service, accountability and transparency. There is a modern trend towards government openness and open data ([3], [4]) which relies heavily on semantic web (SW) technologies such as RDF ([8]) and OWL ([9]). Based on eGovernment application development experience at our local government organization, we have come to the

realization that there is a fundamental similarity between MDD technologies on one side and SW standards and tools on the other. In both cases, a high-level language is used to create domain models. The difference is in how those models are being put to use in software and it comes mainly from the intended applications of MDD, and respectively the SW. In MDD the domain models are either used to generate code in some general purpose programming language or they are dynamically interpreted, while in SW the domain models are used to describe structured data in a formal machine processable way and are usually interpreted as visual presentations for human consumption, data analysis, or mapping to other data formats. In the case of MDD, program logic is sometimes covered by models, for instance via workflow DSLs, but in the majority of cases models are simply used to describe the structure of data, which is what SW does. We have therefore decided to apply SW standards, more specifically OWL, to an MDD process. Since other attemps in using OWL for MDD (such as [1]) were deemed unsuitable and no documented best practices could be found, we have had to establish one ab initio.

1.2 Structure of the paper

First, we describe our problem domain, *Citizen Relationship Management* and the overall requirements for the project. Then we outline the platform architecture and main technological choices. Next, we describe our modeling approach: what is being modeled, how models are interpreted. Finally, we discuss some implementation details.

2 The Problem

The business user of the software described herein is a municipal call center which has been using a legacy case management system for over 10 years. Due to technical constraints, and limitations with configuration of workflows, permissions and business rules, the call center specialists and the fieldworkers using the system (e.g. Animal Services officers, Public Works field personnel) have had to establish numerous workarounds in their work process, and essentially adjust the way they do business based on the configuration options available in the legacy software. As the call center operation grew, however, the system could not match the emerging business requirements without extensive customization. Changes to workflows, notifications, and data fields required an extended amount of time to complete. Additionally, the user interface did not have the necessary flexibility to accommodate emerging business rules, and call center staff required extensive training on the multiple steps needed to complete essential tasks.

The underlying case management model of the system did not match the more dynamic business model of the call center. It was necessary to develop a citizen relationship management system with an architecture that more dynamically reflects changes to the business model, and helps the call center operate more efficiently when providing information and services to constituents.

2.1 Domain

Citizen Relationship Management (CiRM) is based on the CRM (Customer Relationship Management) concept, except that there are profit-oriented activities ([6]). A CiRM system is built to support services to citizens in a wide variety of areas, from requesting basic information, to paying bills, to picking up garbage, to the filling of potholes. A key aspect is reporting non-emergency problems, such as abandoned vehicles or stray animals on the street. These types of non-emergency issues are increasingly handled by city or county-wide call centers in the US, accessed through the 311 phone number. A CiRM is therefore more *case-centric*, with emphasis places on incidents, problem situations and discrete services rather than *customer-centric*, where the focus is on customers, their accounts and their preferences. In CiRM, the focal point of a business process is always a service case which once completed, is archived and accessed only for audit or reporting purposes.

2.2 General Requirements

A key requirement we have had to fullfill is the porting of current case models from the legacy vendor system to our new CiRM system. Another requirement is the integration with applications by other agencies. Frequently there is overlap in functionality with such departmental systems, where both serve similar case management purposes. Enterprise requirements, however, dictate that the systems must be synchronized, and that data and transaction mapping needs to be performed between the two. Yet another critical requirement, particularly important for eGovernment is traceability/accountability. It is necessary to maintain detailed case histories both to satisfy public records laws, but also to improve customer service and conduct service trends analysis. An additional requirement was the ability to adjust case models at runtime-this is one of the crucial reasons for an MDD approach was adopted.

3 Platform Architecture

3.1 Overview

Due to lack of semantic web-based MDD tools, in the course of the system's development effort we created a generic platform where most of the important back-end components are not related to the CiRM domain per se. This project's motivation in adopting MDD is to a large degree for the software to be model-driven *at runtime*, in addition to the usual advantages of having an explicit model and writing less code. Consequently, we write software that interpets our OWL models rather generating code from them. This makes live modifications possible which has tremendous benefits. The architecture outlined here pertains to the platform with application-specific artifacts being plugged into the various architectural components. A detailed architecture was developed following the

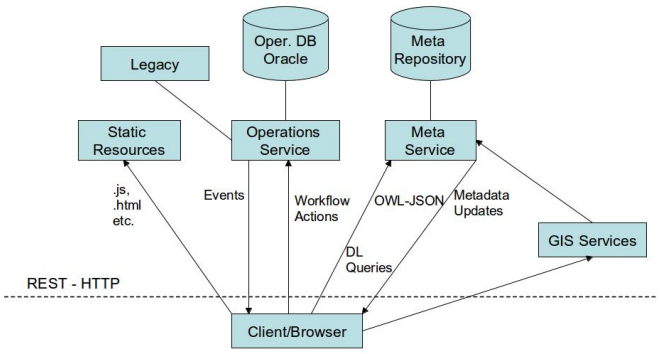

Fig. 1. Architecture Outline

TOGAF process ([10]) with all relevant documents produced over a period of several months. A basic outline is depicted in Figure 1.

According to the architecture, the divide between data and metadata leads to two sets of services: operational services, handling operations data (e.g. actual service cases) and meta services that handle purely metadata related functions such streaming models, generating blueprints, generating UI components. The *meta services* are the core the of architecture as this is were all models, including application domain models and software component models, reside. Models (i.e. ontologies) are persisted in a data store referred to as the *Metadata Repository* or *metabase* for short. The *operations services* on the other hand manage data in an Oracle RDBMS and handle integration with external/legacy systems. The end-user application is based on the latest web technologies - HTML5, JavaScript and related standards.

3.2 Meta Services

Instead of static code generation, we have opted to build the software around the idea of executable models. Both the problem domain and the solution domain are expressed in a formal model that is accessible and modifiable at runtime. The model is then interpreted at runtime by an execution engine comprised of several collaborating components. It is an OWL graph database with full versioning support at the axiom level[1]. There is one and only one metadata repository available for querying. Both server-side and client-side components consume metadata to perform their functions.

[1] To be described in a future publication

The meta services employ OWL description logic reasoning to answer queries using the convenient Manchester DL syntax. Any metadata is returned in JSON format. We employ an ad hoc JSON representation of OWL (there is no standard JSON serialization) that we have defined in due course. Our representation doesn't follow the axiomatic view of OWL. Rather it is more object centric (rather than "fact centric") - we serialize an individual and then all of its properties recursively.

The responsibility of the meta services is to manage all aspects of the model. Many administrative tasks amount to changing certain configurations or tuning a business process or case model, or to adjusting access authorization rules. All those activities are entirely within the realm of metadata management. Thus many modifications that traditionally would require restart of servers or even recompilation and redeployment are entirely handled at runtime. This is where we have found the true benefit of the MDD approach.

3.3 Operation Services

The operation services implement the management of business data and the enactment of business process workflows. Just like the meta services, some of the operation services are platform (i.e. domain-independent) while others are really specific to the application. What defines a service as being operations vs. meta oriented is the fact that it is manipulating concrete business data. The volume of data is much higher, and the data itself is repetitive following structural patterns (i.e. records) in the case of operations services. Alternatively, entities that configure application behavior in some way are not operational data. This, combined with our organization's investment in RDBMs and RDBMs-based reporting technology, led us to chose an Oracle cluster as the backing store. We note, however, that conceptually the line between operational data and metadata becomes fuzzy. In fact, the relational database holds OWL data just like the metabase - some of it in the form of triples, other under a classical relational schema via an ORM-like mapping between OWL and SQL. As with the meta services, historical information is preserved - every time an entity is modified, its current version is time stamped and archived. The services are implemented in Java as a REST API with the application-specific ones isolated under a separate relative path. The API endpoints are referred as action points or workflow steps within the model which is how the model embeds business process behaviors.

3.4 Client-Side Components

Much of the application hand-coding was done using browser-based technologies - JavaScript, HTML/CSS. Because all server-side APIs are stateless and given the capabilities of modern browsers, the bulk of the logic of the application is implemented in JavaScript. We consider JavaScript to be a highly expressive, dynamic language that complements well our late bound use of the models. In fact, JavaScript code can be easily re-generated from a model at runtime, as an

administrative task, without the need for compilation or re-deployment. Granted, we lose the benefits of statics checks. However, our development and deployment processes incorporate tests and the ability to rollback any deployment in case of discovered errors, until they are fixed. Serializing OWL ontologies, or a part thereof, in JSON already provides the structural portion of a JavaScript dynamic object, which is then complemented with additional functions.

Besides regular business logic, client-side components naturally implement UI interaction. Synthesis of UI interfaces is based on an association between UI components and certain high-level abstractions in the domain model. For instance items that are a list, are displayed with a table viewer. Things that are a string are edited with an input box. The model execution engine is capable of doing something sensible in terms of UI, given a business object. It will display a form to edit the object or show a list of its properties to view it, but the layout may not be appealing. To accomodate specifics, HTML templates are used. Such templates are stored in the model together with other software artifacts. Editing a UI template then becomes an administrative task rather than a development task and business users can be empowered to perform it.

4 Modeling with OWL

The flagship modeling language in the MDD world is arguably UML. However several factors, the least of which was a taste for adventure, tilted our decision towards OWL. One major factor were the inference capabilities of OWL. Others are simplicity and universality: while Description Logic in its full power can be difficult to master, working with a handful of concepts like *individuals*, *classes* and *properties* is something well within the grasp of business analysts. Yet, it must be noted that OWL is not an object-oriented modeling tool, unlike the core of UML, it is in fact a logical language with logical semantics and reasoning capabilities not available in mainstream software modeling tools. Hence OWL is much less about *data structures* and much more about factual information. As such, it is perhaps better suited to modeling real-world domains than UML, but less well-tailored to modeling software itself. One key feature missing from OWL 2.0 are meta classes - it is impossible to describe ("talk about") a class. That is, one cannot assign a property to a class like one would to an instance (or an individual in OWL lingo). Also, it is possible to constrain the domain and range of a property, but one cannot make a property be a member of a class. For example, to specify that a property is required, one can use a logical statement saying that class C is a subclass of the class of individuals that all have that property.

Lack of meta classes poses difficulties in particular when modeling the solution domain since the software is described in terms of component types, how they relate to each other, what can be done with them etc. In other words, when modeling the solution domain we frequently have to create descriptions at the meta level as well. In OWL this limitation is overcome by relying on the fact that classes and individuals are disjoint logically, and therefore one is allowed to

use the same name for both a class and an individual. So to talk about a class `X`, we simply declare an individual with the same name `X`. The reasoner knows nothing about the connection between the class `X` and the instance `X`, but this is not an issue as the platform execution engine does recognize that connection. This technique is called *punning* ([5]). It makes the creations of models more verbose due to such duplicate declarations.

In deciding on the OWL variant to use, we were not presented with much of a dilemma since the variant promoted and exlcusively supported by the most popular tool (Protege, [12]) is OWL 2.0, formerly known as OWL DL (Description Logic). The standard API for working with OWL in Java is the OWLAPI ([13]) which is also exclusively based on OWL 2.0. It must be noted that OWL and its ecosystem are developed with the expectation that ontologies will be manipulated by end-user software in a pervavise way, unlike UML which is mainly for engineering tools. That is yet another deciding factor for our choice of OWL as the modeling language.

The model serves the dual purpose of formalizing knowledge about the organization, its function, the environment etc. one one hand, and as the basis for software on the other. Also, as noted above, OWL is used for both data and metadata which yields a very smooth transition from one to the other and makes it easier to create a more dynamic, live system. Business rules can be expressed in SWRL where rules that are exclusive to the model are left to an OWL reasoner to interpret while rules that apply to business objects are run through our own ad hoc inferencing. There are several aspects/dimensions to the model and for development purposes we employ different namespaces and different ontologies (i.e. modules), but the end result at runtime is a single big ontology with all the knowledge the application requires.

4.1 Upper Ontologies

A common practice in the semantic web world when modeling a concrete domain is to find (or define) some very general conceptualization-an *upper ontology* and use it as a starting point. Such conceptualizations have been published and standardized ([7]), but every organization is free to do its own metaphysics. Because at our organization we already had developed such an upper ontology for another project (a semantic search application, where OWL was used for knowledge representation), that ontology was adopted. The top level classes are shown in Figure 2 (a).

There are few, if any, concrete software implications of this upper categorization of the world. It mainly serves as an abstraction aid to a modeler/developer. It also facilitates understanding, documentation, providing opportunities for integration with other in-house software that shares the same upper ontology. For example, now we are in a better position to integrate our online knowledge resources (semantic search) with the government services (CiRM) modeled in their full detail. And this is yet another aspect that sets our approach appart from conventional MDD. Only the relevant abstractions layers can be used as needed, without complete code generation from the whole model.

Fig. 2. Ontology Snapshot

4.2 Domain Model

The domain model is an ontology that describes the problem domain in a software implementation neutral way. Among the aspects of the problem domain is the complete structure of the government organization, with useful information about each department such as phone numbers, office hours of various service points, the list of services it provides. Note that this information is a generally useful and searchable semantic knowledge base, ready to be published online. It is also part of our CiRM domain model used as runtime metadata.

However, the predominant type of business entity in the CiRM world is the service case. A service case is created based on a problem reported by a citizen. Different types of problems require the collection of different information, the engagement of different types of actors and a unique workflow. A given type of problem thus has its own *case model*. Much of the domain modeling revolves around creating and maintaing models of the various types of cases.

Each type of case is an OWL class, but also a (punned) OWL individual so that metadata about that class can be stated. The case type individual has properties describing (among others):

1. Questions that need to be answered to assess the situation (e.g. "approximately how far is the pothole from the sidewalk?")
2. The location of the case (e.g. street address or GIS xy coordinates)
3. The workflow for handling such a case until it is marked as *closed*

All case models also share a common set of attributes such as the location of the case, the date/time when it was opened, who opened it and current status. However because there's no notion of inheritence or subsumption between OWL individuals, and because case type metadata is associated with a punned

individual, this commonality has to be handled in a special way by the execution engine. On the other hand, since data in the CiRM platform is represented as OWL ontologies too, the connection between the case model and the case occurrences is immediate and natural. There is no need to translate from one meta model to another (e.g. UML to Java), hence no mismatch is possible, no unnatural representations in the target language warranted.

4.3 Software Model

Software artifacts are modeled in OWL by treating the application software as a domain to be described like any other domain. Model elements range from top-level entities like `Software_Application` to simple name-value `ConfigParam` entities. Since the runtime representation of the domain model remains in OWL (no separate Java object model needed), and since we do not do code generation for back-end components, only those that must be dynamically found in some way are modeled, like SOAP and REST services or Java implementation classes that must be linked depending on the context. Nevertheless, some parts of our system are driven in an entirely generic way by the model. Figure 2 (b) depicts a sample from the software model portion of the ontology.

We mentioned above the synthesis of user interfaces. To achieve this, we have created a set of UI components as a client-side JavaScript library and we have described them in our ontology. Just like case models lead to JavaScript business objects, descriptions of the UI components are first JSON-serialized and then augmented behaviorly as browser-based UI components. A UI component can be as simple as an HTML template rendered contextually from some data, e.g. top-level UI components like a whole page. Or, it can be something with much richer functionality. For instance, a familiar type of component is the *data table* component for interacting with data in tabular format, with the ability to sort by column or filter out certain rows. We can create an OWL individual description of a data table with a particular configuration, associate it with an operational data query and plug it in, say, an HTML template. In other words, the software model contains concrete instantiations of software components, bound to the domain model and ready to be assembled for end use in addition to abstract descriptions of components types. This is possible because of our universal usage of OWL.

Another illustrative example of the benefits of the integration of domain and solution models within a single ontology repository is access control: notions such as business actions and access policies are part of the solution domain (as they pertain to application behavior) while a case type is part of the problem domain. But we can directly associate a case type with a set of access policies for the various actions available without ever leaving the world of our OWL repository. We can use Description Logic queries to find out what the access policies are. Furthermore, we can use SWRL `if-then` rules to automatically create access policies based on some properties of the resource being protected.

In general, the inferencing capabilities of OWL have proven to be a very powerful tool, but there are several practical limitations in addition to the lack of meta-modeling that we have had to deal with.

4.4 Problems with OWL

While the experience of using OWL as an MDD foundation has been overwelcomingly positive, it was not without roadblocks, mostly resulting from the relative immaturity of SW technologies:

- The main obstacle is the lack of solid, high performance reasoner[2] implementation. There is currently no reasoner that supports all possible inferences.
- Reasoners are not designed to work in a multi-threaded concurrent environment.
- No reasoner supports any sort of contextualized inferencing where a query is performed within the context of a set of extra assumptions/axioms. This would be valuable in reasoning with the operational data entities which, compared to the large meta ontology, are just small sets of axioms that could be assumed just for the context of a given reasoning task.
- Punning provides a workable solution for the lack of meta classes, but in order to express inheritence and other meta properties we would have to develop or adopt an ad hoc framework on top of OWL to recover the lost expressiveness of OWL Full (which has meta classes).
- OWL lacks some basic data structures such as arrays and lists which are difficult to express ([2]).
- Lack of MDD tooling means that some dependencies are harder to track. When an ontology entity is referred directly in code, those references can easily become invalid and undetected. Therefore, we have learned to keep such cases to a minimum. We simply consider this as part of the general drawback of dynamic languages vs. static compilation.
- The fact the OWL is fist and foremost a mathematical logic language that just shares some of the notions behind object-oriented programming (without its constructs or semantics) has led occasionally to unexpected inference results or to overly verbose models. In particular, consequences of the Open World Assumption and Non-Unique Name Assumption challenged the team.

The list is not exhaustive, but it covers the most unexpected problems faced in the course of the project. The technical issues related to reasoning over the models were the biggest hurdle. They were avoided either through aggressive caching or by hand-coding ad hoc inference procedures within the meta services.

5 Model Change Management and Operational Data - Two Implementation Highlights

In this section we present a few details about the implementation and our development process relevant to MDD.

[2] This is how OWL inference engines are called.

5.1 Model Change Management

Following our guiding vision of a model-driven live system where the software is modified at runtime, we needed a reliable change management process and tools. Nearly all business aspects are represented in the model. There are virtually no configuration files, except for a few bootstraping parameters like the location of the metabase. Therefore, a lot of software changes amount to meta repository updates and a crucial aspect of the architecture is the ability to manage those updates just like source code updates within a version control system (VCS). The lack of a native VCS is a frequent problem with modeling tools since file-based versioning is too coarse grained and leads to merging problems due to the usually non-deterministic model serialization. But versioning is crucial to the agile development process we have put in place. Since models essentially compress information, changes are potentially high impact, hence the importance of the ability to go back in time .

As part of our platform development effort, we created a distributed versioning system similar to GIT, but for OWL and that is at the level of the language itself, rather then the textual representation. The units being tracked and versioned are the logical axioms of OWL. The implementation relies on a hypergraph database ([11]) which acts both as the VCS and the meta repository. A software model update is enacted as a push of an ontology changeset which triggers clearing of caches and updates of other runtime structures within the meta services. Rolling back an update triggers the exact same set of events to adjust the runtime state.

Model updates in this setup are akin to component deployments in a traditional architecture. We use Protege to work with the model via a plugin integrated to our infrastructure. The idealized model development process looks very much like a standard programming process:

1. Make model changes on local machine.
2. Push changeset to local development environment.
3. Test locally, then push same changes to test environment.
4. Potentially multiple team members push to test environment - changesets get automatically merged.
5. Check OWL consistency with reasoner, run application-specific test suite in the test environment.
6. Push from merged changeset from test to production.
7. Rollback last changeset from production in case of problems.

In cases where business users need to work with the model, but find it difficult to learn Protege and OWL, a simplified web-based UI was developed. However, model modifications through that UI go through the same change management process via the same VCS. A similar process is put in place for the static web resources. It would be easy to store those resources inside the model as well, but we have not done so due to lack of tool support. Finally, note that only updates to the Java-based core components, i.e. the server-side of the model execution engine, force an interruption of service, but such updates are much rarer.

5.2 Operational Data as Ontologies

We refer to top level entities in our operational data as *business objects*. Those are the enterprise entities that one finds in any enterprise framework and the things that we persist in our relational database. Even though they are stored in an RDBMs, our runtime system manipulates them as OWL ontologies. Each business object is represented as a small ontology following a few naming conventions enforced by the execution engine, such as its IRI[3] format and the presence of a top-level individual representing the entity. We refer to such ontologies as *business ontologies* or *BOs* for short. The typical BO type is of course the *ServiceCase*. Two notable aspects of our relational storage are: (1) the ability to store BOs in a generic way as sets of axioms or store them in a more efficient way by mapping a given BO type to an SQL table; and (2) the auto-versioning of all BOs, using the customary `valid_from`/`valid_to` timestamp mechanism. That is, a modification of a business entity creates a new version of that entity instead of overwriting the existing data. Previous data can be retrieved for auditing purposes.

As with metadata ontologies, BOs are manipulated moslty through their JSON representation which is object-like and very natural to programmers. As noted in section 3.4, JSON serialization is akin to code generation at runtime - when functions are merged into the JSON representation, we have a complete JavaScript object entity, dynamically synthesized from metadata and operational data, in the true spirit of MDD. To create a brand new entity, we use metadata information to construct a prototypical blueprint instead of existing operational data. In particular, a case model is also used as a prototype to construct a new case. This is more in tune with the object-based nature of JavaScript, as opposed to class-based nature of Java and other static languages traditionally the target of MDD.

Finally, note that the perennial problem of model evolution that entails changes to the structure of operational data is solved by matching versioned data with the appropriate versioned metadata. For instance, when a property is added or removed from a BO type, only newly created data will have the correct blueprint. Because of the auditing requirement, updating all existing data is not an option, regardless of whether it is feasible or not. So whenever the latest metadata is incompatible with some older BO (e.g. after a property removal), the meta repository has to be queried for the corresponding older version of the BO type with which to interpret the operational data.

6 Conclusion

Some of the advantages of using OWL for MDD are the simplicity and universality of OWL. When metadata and data, domain and solution all share the same underlying formalism, many design problems are minimized or eliminated. When administrative tasks amount to adjustments of a live application model,

[3] International Resource Identifier

many development tasks are dispensed with. On the other hand, we have experienced difficulties with the level of maturity of SW supporting tools, given the unorthodox use we made of them. We are hopeful that our practical experience has contributed to bridging the largely historical gap between the classical AI techniques of knowledge representation and the modern sofware engineering approach of model-driven development.

References

1. Parreiras, Fernando S.: Semantic Web and Model-Driven Engineering, ISBN: 978-1-1180-0417-3
2. Nick Drummond, Alan L. Rector, Robert Stevens, Georgina Moulton, Matthew Horridge, Hai Wang, Julian Seidenberg: Putting OWL in Order: Patterns for Sequences in OWL. OWLED 2006
3. Bertot, J. C., Jaeger, P. T., Grimes, J. M. (2010). Using ICTs to create a culture of transparency: E-government and social media as openness and anticorruption tools for societies. Government Information Quarterly, 27(3), 264271.
4. Janssen, K. (2011). The influence of the PSI directive on open government data: An overview of recent developments. Government Information Quarterly, 28(4), 446456
5. B. Motik: On the Properties of Metamodeling in OWL, ISWC 2005, Galway, Ireland, 2005. PDF http://www.cs.ox.ac.uk/people/boris.motik/publications/motik05metamodeling.pdf
6. A. Schellong (2008) Citizen Relationship Management, Brussels: Peter Lang Publishing.
7. I. Niles and A. Pease. Towards a standard upper ontology. In Proceedings of the International Conference on Formal Ontology in Information Systems (FOIS), pages 29, 2001.
8. Eric Miller and Frank Manola. RDF Primer. W3C Recommendation, 2004
9. W3C OWL Working Group. OWL 2 Web Ontology Language Document Overview. W3C, 2009
10. Josey A., Harrison, R., Homan, P., Rouse, M., van Sante, T., Turner M., van der Merwe, P., 2011. TOGAF Version 9.1 - A Pocket Guide. 1st ed. Amersfoort: van Haren Publishing.
11. Borislav Iordanov. HyperGraphDB: A Generalized Graph Database, In Proceedings of the 2010 international conference on Web-age information management, 2010. PDF http://www.hypergraphdb.org/docs/hypergraphdb.pdf
12. http://protege.stanford.edu/
13. http://owlapi.sourceforge.net/