AN ONTOLOGY DRIVEN APPROACH TO SOFTWARE SYSTEMS COMPOSITION

Hlomani Hlomani and Deborah Stacey
Computing and Information Science, University of Guelph, Guelph, Ontario, Canada

Keywords: Ontologies, Software systems composition, Semantic web.

Abstract: This paper discusses a proof of concept prototype system driven by knowledge embodied in a set of Ontologies; an algorithm Ontology and an execution timeline Ontology. The main idea behind Ontology Driven Compositional System (ODCS) is allowing domain experts to compose a system by choosing the system components and the desired interactions between these components in a way suitable to their problem. This differs from systems that ship with predefined data sets and algorithms that are preset for a specific purpose which may not be suitable for certain dynamic domains that require highly adaptive, and easily modifiable systems.

1 INTRODUCTION

Ontologies, have been gaining interest and acceptance in computational audiences: formal Ontologies are a form of software, thus software development methodologies can be adapted to serve Ontology development.[1] *As well, Ontology development can be adapted to serve software development methodologies.*

The premise of this paper is that no single solution exists for all problems at hand as some domains are faced with very dynamic problems dictated by many parameters and environmental settings. This implies the need for systems that can be rapidly developed and changed to fit a given task. The changes may be in altering the constituent components (e.g. algorithms) or in altering the performance and functionality of these algorithms. The problem though is that with current system development approaches, modifying the system would require a substantial amount of technical proficiency and time while in most disciplines (e.g. syndromic surveillance, also know as disease detection, monitoring, and real-time situational awareness) it may be desirable to have the very users of the system (e.g. epidemiologists) dictate, among other things, the algorithms suitable for a problem, the appropriate data sources, and the arrangement and flow of data between the constituent algorithms. These users may be technically naïve and, therefore, it may be desirable for these users (domain experts) to not concern themselves with the programming aspect of composing the system. We therefore propose a prototype system that has at its core several Ontologies that provide rich descriptions of the components of a system thereby allowing for the autonomous creation and modification of systems which are problem or mission oriented by the very domain experts who need to use them.

[1]International Conference on Knowledge Engineering and Ontology Development website: http://www.keod.ic3k.org/.

2 BACKGROUND

2.1 The Semantic Web and Ontologies

The semantic web is defined by Shadbolt, Hall and Berners-Lee as a web of data as opposed to the current web which is a web of documents (Shadbolt et al., 2006). It is envisioned to extend the current web by giving web content semantic meaning for the betterment of cooperation between computers and humans (Dong, 2004). Central to the semantic web approach is the use of Ontologies which play a pivotal role not only in the advancement of the semantic web but in knowledge sharing and management at large. These are formal computer stored specifications of concepts in a domain and the relationships that holds between those concepts to provide for a common platform for information exchange (Tjoa et al., 2005; Horridge et al., 2007). Zhang (Zhang, 2007) asserts that an Ontology is a rich expression of semantic relations.

Hlomani H. and Stacey D. (2009).
AN ONTOLOGY DRIVEN APPROACH TO SOFTWARE SYSTEMS COMPOSITION.
In *Proceedings of the International Conference on Knowledge Engineering and Ontology Development*, pages 254-260
DOI: 10.5220/0002304502540260

2.1.1 General Software Development Architecture

An Ontology driven approach to software systems development is similar to other software development paradigms in that there has to be a guiding architecture that specifies the components within the intended systems and how these components relate to each other. For the semantic web a fitting architecture was presented by Knublauch (Knublauch, 2004). He differentiates between two layers of a semantic web application. First, there is a *semantic web layer* that hard-codes knowledge about a particular domain which is then used to model the behavior of the application. Secondly there is an *internal layer* which is composed of the reasoning mechanisms and the application's control logic that controls the application's functionality and interacts with the user via an interface. The prototype system discussed in this paper takes advantage of this general architecture but however provides extended definitions of certain concepts particularly in the application logic. This approach will be discussed in Section 3.

2.2 Other Approaches to Systems Composition

2.2.1 Web Services

Web services are Service Oriented Computing (SOC) implementations that provide a standardized way of presenting some functionality in the form of integrated web-based applications using XML, SOAP, WSDL, UDDI and other internet protocols. They are distributed applications which can be discovered, bound to and interactively accessed in an autonomous manner (Charfi and Mezini, 2007). They have also been defined in the literature as networked capabilities with openly accessible interfaces for other machines to discover and invoke in real time (Blake and Nowlan, 2008; Milanovic and Malek, 2004). Web Services are network, technology and platform independent (Papazoglou, 2003) making them highly interoperable. They are also loosely coupled making the idea of reusability very attainable.

Web Services fit into this discussion of system composition in that a functional system can be developed from putting together already existing web services in what is termed as service composition. Service composition is a cost effective approach which allows a developer to build a fully fledged application by using already existing components in the form of web services. Although there may be some benefit in invoking a single web service, combining several already existing services and ordering them to best suit your requirements may bring added value (Charfi and Mezini, 2007; Milanovic and Malek, 2004). Developers therefore use service composition to rapidly develop applications that meet users needs, meet an organization's goal, or provide a new service function (Milanovic and Malek, 2004; Feenstra et al., 2007).

While service composition has advantages in theory, the adoption and use of discovery protocols and the lack of development of libraries of freely available and discoverable services has left this approach mostly unrealized. Since web services are by definition distributed and thus require distributed resources, this restricts their applicability in some domains where privacy and security are critical. It also necessitates that the user be constantly connected to the Internet. While the compositional system being proposed in this paper does not necessarily use web services, the utilization of distributed programs and resources is a logical extension of our system definitions and thus our system is a superset of the domain of all services/programs including web services.

3 ONTOLOGY DRIVEN COMPOSITIONAL SYSTEM (ODCS)

A prototype system called ODCS has been developed that demonstrates the power and suitability of using Ontologies as the main driver for a compositional system both for its development and post development functionality. By compositional system we mean a system that allows its components to be put together in a systematic manner to achieve a common goal or to derive yet another functional application. This was done by building on current research in the use of semantic webs for application development. These include design patterns/architectures, tools and technologies. As mentioned earlier the architecture discussed in (Knublauch, 2004) was adopted for the development of this system (Figure 1).

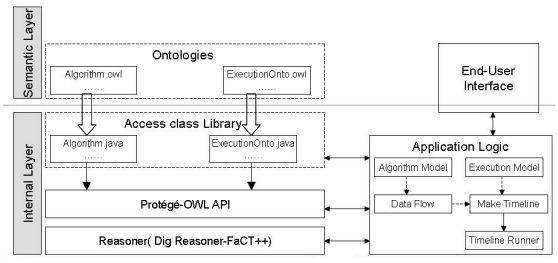

Figure 1: ODCS Architecture.

3.1 Ontologies

The use of Ontologies to provide domain specific knowledge base in order to facilitate for communication and knowledge sharing between people and various computer agents is arguably the pillar on which the semantic web hinges. We however take it beyond just the facilitation of knowledge sharing. We use this common knowledge to drive the development of a compositional system by using the Ontologies to provide an extensible description of the components of a system and how they can be put together. To achieve this we have created Ontologies to describe algorithms that provide the needed functionality including architectural elements and the order of execution to describe the order in which the composed algorithms are going to be run. This section discusses these Ontologies.

3.1.1 Algorithm Ontology

This Ontology was designed to provide an extensible standard description of algorithms in the context of the elements or aspects of the algorithm that one would need to know about the algorithm in order to use the algorithm. The use of a common description of algorithms would allow for easier addition and removal of algorithms from a system or application as this will be done at the semantic level rather than at the programming level therefore reducing dependence on skilled personnel. The development of an Ontology for algorithms, however, proved to be more complex than initially perceived. This can be attributed to the existence of diverse views and perceptions on classification of algorithms. Unlike in other domains where agreed upon taxonomies and classifications already exist, the same cannot be said about algorithms. Algorithms vary in terms of their purpose, input and output requirements, and the parameters needed to modify their behavior. Algorithms that use a similar problem solving approach can also be group together to form a hierarchy. We cannot, therefore claim absolute correctness as different levels of abstractions and perceptions exists. However, it seems more reasonable to classify algorithms according to the reason they were designed (purpose). The focus of this Ontology however is not to provide a classification of algorithms but rather we focus on providing a generic description of the algorithms in order to facilitate automation of the algorithms' usage and provide a plug and play kind of functionality. Such elements as the algorithm's operating environment, algorithm type, input, output, and how it is used are described in the Ontology.

The classes in the Ontology describe what is needed to run an instance of an algorithm defined by the subclasses of the `Algorithm` class. In this Ontology everything is a subclass of `Thing` including `Algorithm` which can be seen as the overarching concept. The algorithm concept is the very thing that all the other classes in the Ontology describe and relate to. We differentiate between two main classes of algorithms: *Computational Algorithms* and *Architectural Elements*. We perceive computational algorithms to be the very algorithms that provide domain specific or general purpose functionality. Architectural elements are those pieces of code whose operation provide support to the computational elements and help in creating and maintaining the structure of the system.

An algorithm exists in some sort of file be it in its source code form or as a ready to run executable file. For this purpose we created a `AlgorithmFile` class which aims at describing aspects relating to the actual algorithm file. It defines such properties as the file name, creation date, last updated date etc. It also defines object properties that provide linkage to location and operating environment classes. Each algorithm executable or source code file has a specific environment in which it can operate on. For this purpose we have an `OperationEnvironment` class that describes this environment. This include subclasses to describe the operating system, drivers, and other software needed to run the algorithm.

Subclasses specify specific types of algorithms and define other properties that are relevant to that specific type. Figure 2 depicts this kind of hierarchical structure of the class. If the system has to search for a statistical algorithm the results could be an instance of any of the subclasses of the statistical algorithms e.g. an instance of an `EMAAlgorithm` class since its instance is an instance of `StatisticalAlgorithm` due to inheritance. This inheritance structure allows the class to be extended when a new algorithm type definition is needed.

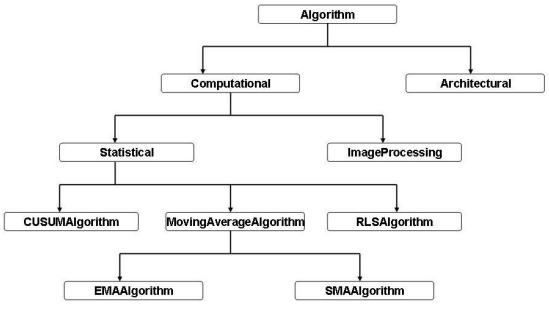

Figure 2: Overview of the Algorithm class hierarchy.

Most algorithms take in some sort of input, therefore for every instance of an algorithm there must be

an indication of whether it receives any input and a precise definition of how the input is received. The `Input` class provides these descriptions. Input to an algorithm may be in the form of data files read by the algorithm, command line parameters passed through the input stream to the algorithm or read from configuration files. These are described by further defining subclasses of the `Input` class. For example, a `Parameters` class is defined and its subclasses are then defined to include the `CommandLine` class that describes the parameters passed through the input stream to the algorithm and `ConfigurationFiles` class that describes parameters read from configuration files. These subclasses can also be further subclassed to provide needed details. This is depicted by Figure 3.

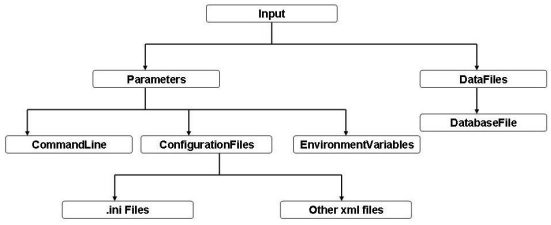

Figure 3: Overview of the Input class hierarchy.

When composing a system by combining algorithms together through their input streams and output stream several incompatibilities may exist some of these include data type mismatches and data stream mismatches. The architectural elements class within the algorithms Ontology therefore serves to provide a generic description of these internal *glue* elements of the system. Data type converters are components of a compositional system that offers data conversion to facilitate compatibility between algorithms. Converters are algorithms in that they are standalone pieces of code that can be fed input and run to produce some output. An example mismatch is that of different data stream types feeding into and out of a pipe. One algorithm may be printing output to the standard stream while the algorithm it connects to expects input from a file. Therefore rather than just declaring the connection to be impossible an architectural element would be placed between these two algorithms to satisfy both algorithms' requirements. In this case the element would be reading data from the standard stream and writing it to a file so that the next algorithm can read the file.

3.1.2 Execution Timeline Ontology

Ultimately the composed system will be run. To facilitate flexibility and extensibility we have developed an Ontology that describes how the events within the composed system will be run. Please refer to section 3.4 for a detailed discussion on the individual elements of a timeline. The basic functionality of this Ontology is to describe the order in which the connected algorithms will be executed. The Ontology defines this order in the form of a `Timeline` class that has a number of intervals. The `Interval` class has a collection of start and end triggers and a number of events. The `Event` class also has begin and end triggers. Specific triggers are defined as subclasses of the `Trigger` class. Triggers could for example be performing the function of checking for preconditions and postcondition of an event or an interval or specifying what the next process (event or interval) to run is. The Ontology also defines exit points within each event, interval and timeline in the form of errors handlers.

3.2 Class Libraries

The system discussed in this paper along with many other semantic web enabled applications is driven by a number of core Ontologies. Their functionality relies on the querying of ontological instances and their property values. This include the use of reasoning engines (in this case FaCt++) to infer some knowledge which was perhaps never explicitly asserted. Two alternative approaches to achieving this knowledge acquisition from Ontologies exists. Generic OWL parsing APIs such as Jena and Protégé-OWL can be used to directly interact with the Ontology. These provide a model in which OWL instances and their properties are saved using generic Java classes (Knublauch, 2004). As the application grows these may be rendered inconvenient since access to OWL objects is done through specifying the objects' names often as strings. This is rather hard to maintain. Another approach which is deemed to be the better of the two and adopted in this work is one in which in addition to using generic APIs, access to the ontological facets is done through custom made Java classes that resemble the structure of the Ontology. This does not only allow seamless access to the Ontology but also provide cleaner object oriented design patterns. This class library consists of definitions of interfaces and classes that implements these interfaces. The interface is a normal Java interface that defines mandatory properties and methods that should be implemented by classes that commits to using this interface. The power of interfaces will be discussed in Section 3.2.2 where such issues as multiple inheritance will be examined.

3.2.1 Generation of Class Libraries

The creation of custom Java classes that map to the Ontology classes is not a trivial process but one can manually create them. However, there exists code generators that can be used to ease this rather intricate process. This Java code generation capability does exist within the Protégé-OWL editor and it was used for the generation of the Access Class Libraries used for the development of this system. Several other code generators exist; examples include OWL2Java and Jastor (Zimmermann, 2009; Szekely, 2009). The primary reason for the use of a Java class code generator was to delegate this task to mechanisms that would do it instantly and hence save considerable amounts of time. However, the code was then manual examined to identify omissions and physically amend faults. These code generators are designed to create a set of Java interfaces and implementation classes from OWL Ontologies such that an instance of a Java class represents an instance of an OWL class within the Ontology. As will be discussed in the section that follows the code generators may not represent all the classes and constraints within the Ontology solely because of the difference between Java and OWL. Manual inspection of the generated code and writing of the omitted concept may still have to be performed.

3.2.2 OWL to Java Mismatches and their Fixes

It would have been ideal to have a one-to-one mapping of the Java classes to the OWL Ontology classes including their properties and asserted restrictions. However due to significant differences inherent within these languages this may not be attainable and therefore alternative ways need to be devised to minimize the impact of these differences. One of the major omissions by the code generator in our case was a someValuesFrom restriction placed on several data and object properties. The purpose of this restriction was to constraint the creation of property values to certain pre-listed classes and class instances. This implies enumerated classes. Enumerated classes are ignored by code generators since they are anonymous classes. This has little impact on our system since our primary focus is not on the creation of Ontology instances but rather their use to drive the development of a compositional system and in the cases where the population of the Ontology is done through an Ontology editor like Protégé this limitation would not be applicable. It would however be quite important for an application that alters the Ontology instances to handle this variation.

We do however recognize solutions and mappings provided in (Kalyanpur et al., 2004). The paper suggests two rather simple solutions. The first solution involves the definition of an enum, struct or a list object that contains possible values for the property. The method that creates the property value will then loop through the enum checking to see if the new value is one of the possible values contained in the enum. The second solution is somewhat similar to the first one but it instead makes use of listeners registered on the restricted property which are invoked every time the property is changed.

Java is a single inheritance object oriented language while OWL is a very rich and highly expressive description logic based language that supports multiple inheritance. Java does not support what may be referred to as multiple implementation inheritance. It however allows a class to implement multiple interfaces. This is a feature that can be exploited to model classes that have two or more super-classes. We have however, managed to get away with not using multiple inheritance in the implementation of our Ontologies so this remains a feature that can be explored should the Ontology evolve and have classes that inherit from several other classes.

3.3 Data Flow

Our idea revolves around pulling already existing algorithms and connecting them into a functional system. We therefore developed a framework to facilitate this connectivity and the flow of data between algorithms. This was achieved by developing a collection of classes that defines a `TaskGraph`, `Task`, `Nodes` and `Connectors`. A `Task` is a representation of a unit that performs some function. For the purpose of this prototype system this can be in the form of an algorithm, a converter or any of the architectural elements. A `Task` is created when the user chooses an algorithm or when an architectural element is invoked to provide for compatibility of the algorithms. A `TaskGraph` on the other hand is a collection of connected tasks. `Nodes` represent the inputs and outputs to an algorithm, converter or an element. They can take the form of file input and output nodes, standard input or output stream nodes, socket nodes and parameter nodes. We also define a `Connector` class that describes methods needed to establish a connection between algorithms. The possibility or absence of a connection being established between selected algorithms depends upon the compatibility of these output and input nodes. The connectivity also depends upon the availability of a *glue* component (either a converter or element) that will be placed in between the algorithms in case the algorithms nodes are incompatible.

3.4 Execution Timeline

Once a `TaskGraph` has been established a walk through the `TaskGraph` will be done to extract and create events thereby creating an execution `Timeline`. This creation and ordering of events is dependent on the type of output and input of the sending and receiving algorithms. To illustrate this consider Figure 4. The figure depicts a `TaskGraph` that

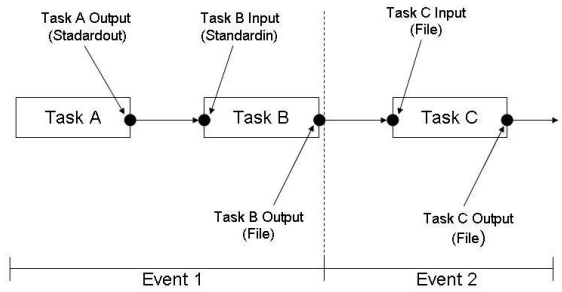

Figure 4: An example `TaskGraph` and the extraction of `Events`.

has three connected `Tasks`. `Task A` pipes its input into `Task B` (this is possible since `A` prints to the output stream and `B` reads from the input stream). `Task B` then writes its output to a file. `Algorithm C` then reads the file and produces output of its own. This implies two `Events`: first `Task A` and `B` are run concurrently piping `A's` output to `B` and the output of `B` is written to a file; secondly `Task C` runs, reads the file produced by `B` and produces output. In this example scenario, the `Timeline` definition would have one `Interval`. The `Interval` would have one start `Trigger` that defines the next process to be run (`Event 1`) and the two `Events`. `Event 1` would have an end `Trigger` that specifies `Event 2` as the next process to run. `Event 2` on the other hand would have a start `Trigger` that checks for the existence of the input file before it could start otherwise the `Event` will terminate with an appropriate error and ultimately the `Timeline` will terminate too. The `Interval` will also have an end `Trigger` that signals that the `Interval` is the last within the given `Timeline` and therefore providing an end point for the `Timeline`.

Figure 4 is an example of a pipes and filters architecture. This architectural style is a data flow architecture that concerns itself primarily with the movement of data between data processing elements (Taylor et al., 2008). To better define a pipes and filters architecture we first define the components. Filters are defined by (Taylor et al., 2008) as preexisting components/programs that consume data from the input stream and produce data through the output stream. Their basic function is to perform arbitrary processing of data that may include transformation of the data and enrichment. Pipes are defined as connectors that interconnect two filters by offering buffering functionality and routing of the first filter's output stream to the input stream of the second filter. It goes without saying then that a pipes and filters architecture is an architectural style that defines several executable programs (filters) that are executed possibly concurrently and employing pipes to route data streams between the programs. This type of architecture allows for the creation of applications even by people without prior software development training. This architectural pattern is trivial to construct from the ODSC Ontologies and can itself be stored in an Ontology that describes architectural design patterns.

4 DISCUSSION

An Ontology driven system offers several advantages. The users of the system has overall control of the system. They have control over what algorithms are included and excluded from their system. They however need not have any technical/programming skills to be able to add or remove an algorithm from their system. They may however need to have an understanding of Ontologies since this will be where most modifications will take place. The whole approach itself allows for possibilities that may have otherwise not been possible following other software development methodologies. A fully functioning system can be rapidly developed by a non-technical user and changed on the fly without the aid of a technical professional. Technical personnel can devote their time on developing algorithms that users may require since new and already existing algorithms can be published onto the Ontology thereby allowing them to be ported and used in the system. The complexity or simplicity of the system is dictated by the user as he/she designs the system by selecting and connecting algorithms into complex or simple combinations that are suited for their purpose. The timeline Ontology and framework also allows for several architectures to be achieved depending on the needs of the application. Events can be run concurrently and branching from one set of events to another can also be achieved allowing for flexibility in the functionality and structure of the application.

The prototype ODCS described in this paper does not yet include a user interface to allow for simple manipulation of the Ontologies by a non-technical domain expert. The interface is by design left as a detachable module since it should be custom designed to fit the abilities, expectations and conceptual frame-

work of the user's domain of expertise. While the interface may change, the underlying algorithms and architectures captured in the Ontologies are available for use by multiple domains. This will allow many application areas to share algorithms developed for different domains. For example, there are many pattern recognition and datamining algorithms developed by the machine intelligence community that are often overlooked by users in domains such as epidemiology, economics, etc. and thus their applications are unnecessarily limited. A system such as ODCS could make a world of specialist algorithms and techniques available to a much wider range of domains and users than is now possible.

5 CONCLUSIONS

In this paper we highlighted the problems of conventionally developed systems with predefined data sets and algorithms that are preset for a particular purpose. We highlighted that these systems may not be suitable for dynamic domains whose environment, parameters and needs change rapidly. We then explored how an Ontology driven approach to software composition can be used to drive the creation of adaptive systems. We specifically discussed ODCS, a prototype system that follows this approach. It is safe then to conclude that an Ontology driven system offers many advantages and is suited for dynamic domains that require systems that can be changed with ease by their users.

REFERENCES

Blake, M. and Nowlan, M. (2008). Taming web services from the wild. *IEEE Internet Computing*, 12(5):62–69.

Charfi, A. and Mezini, M. (2007). Ao4bpel: An aspect-oriented extension to bpel. *World Wide Web*, 10(3):309–344.

Dong, J. (2004). Software modeling techniques and the semantic web. In *Proceedings of the 26th International Conference on Software Engineering*, pages 1160–1163.

Feenstra, R., Janssen, M., and Wagenaar, R. (2007). Evaluating web composition methods: The need for including multi-actor elements. *The Electronic Journal of E-Government*, 15(2):153–164.

Horridge, M., Jupp, S., Moulton, G., Rector, A., Stevens, R., and Wroe, C. (2007). *A practical guide to building owl ontologies using the Protege-OWL Plugin and CO-ODE Tools*, 1.1 edition. Retrieved June 25, 2009, from http://www.co-ode.org/resources/tutorials/ProtegeOWLTutorial-p4.0.pdf.

Kalyanpur, A., Pastor, D. J., Battle, S., and Padget, J. (2004). Automatic mapping of owl ontologies into java. In *16th International Conference on Software Engineering and Knowledge Engineering*, pages 98–103.

Knublauch, H. (2004). Ontology-driven software development in the context of the semantic web: An example scenario with protege-owl. In *International Workshop on the Model-Driven Semantic Web*.

Milanovic, N. and Malek, M. (2004). Current solutions for web service composition. *IEEE Internet Computing*, 8(6):51–59.

Papazoglou, M. (2003). Service oriented computing: Concepts, charecteristics and directions. In *Fourth International Conference on Web Information Systems Engineering*, pages 3–12.

Shadbolt, N., Hall, W., and Berners-Lee, T. (2006). The semantic web revised. *IEEE Intelligent Systems*, 21(3):96–101.

Szekely, B. (2009). *Jastor: Typesafe, Ontology Driven RDF Access from Java*. Retrieved June 25, 2009, from http://jastor.sourceforge.net/.

Taylor, R., Medvidovic, N., and Dashofy, E. (2008). *Software Architecture Foundations, Theory, and Practice*. John Wiley and Sons Inc.

Tjoa, A., Andjomshoaa, A., Shayeganfar, F., and Wagner, R. (2005). Semantic web challenges and new requirements. In *Proceedings of the 16th International Workshop on Databases and Expert Systems Applications*, pages 1160–1163.

Zhang, J. (2007). Ontology and the semantic web. In *Proceedings of the North American Symposium on Knowledge Organization*, pages 9–20.

Zimmermann, M. (2009). *Owl2Java - A Java Code Generator for OWL*. Retrieved June 25, 2009, from http://www.incunabulum.de/projects/it/owl2java/owl2java-a-owl2java-generator.