HELSINKI UNIVERSITY OF TECHNOLOGY
Department of Computer Science and Engineering

Tuomas Korpilahti

ARCHITECTURE FOR DISTRIBUTED DEVELOPMENT OF AN ONTOLOGY LIBRARY

Thesis submitted in partial fulfilment of the requirements for the degree of Master of Science in Technology.

Espoo, March 31, 2004

Supervisor:	Prof. Jorma Tarhio
Instructor:	Prof. Eero Hyvönen

HELSINKI UNIVERSITY OF TECHNOLOGY	ABSTRACT OF THE MASTER'S THESIS

Author:	Tuomas Korpilahti		
Title:	Architecture for Distributed Development of an Ontology Library		
Date:	March 31, 2004	**Pages:**	60
Department:	Department of Computer Science and Engineering		
Professorship:	T-106 Software Systems		
Supervisor:	Prof. Jorma Tarhio		
Instructor:	Prof. Eero Hyvönen		

This thesis analyses a set of ontology change operations that break ontology dependencies, and describes a client-server based ontology library architecture capable of identifying those changes and helping to manage distributed ontology development.

The Semantic Web is the next generation of the Web, where computers are able to understand the contents of the documents they store, and are able to intelligently link and combine the documents based on their content. Ontologies are the core of the Semantic Web. They are knowledge models that define the concepts used to describe the documents and the semantic relations between them. The promise of the Semantic Web requires shared, re-usable ontologies. To ease reuse, ontologies are collected into public ontology libraries.

The main method of re-using an ontology is to include it as a part of another ontology. The second ontology adds application specific knowledge by extending the definitions of the first ontology. This approach allows ontology modularization and distributed development. However, when the first ontology needs to evolve and is modified, the changes may break the second ontology.

The thesis presents a set of ontology changes discussed in the literature. The effects of each change operation to the depending ontologies are analysed, and the changes are prioritized in terms of the severity of their effects and of the frequency of their use.

A client-server based architecture is designed to catch the changes that break ontological dependencies. The system is capable of identifying the problem when the change is made in an ontology editor, and can thus warn the ontology developer on the effects of the change. The architecture framework allows building extensions to develop methods for resolving the conflicts semi-automatically or automatically. For this, the architecture can provide the exact cause of each problem and its effects in all ontologies that depend on the one that is being modified.

The system incorporates a public ontology library where stable versions of ontologies can be published, and a development library where consistent version control is enforced. A managed publishing process is established between the two libraries.

The system is demonstrated with a prototype, which was created to allow the distributed development of an upper level ontology: Yleinen Suomalainen Ontologia (YSO, Standard upper Finnish ontology).

Keywords: ontologies, ontology libraries, distributed ontology development

TEKNILLINEN KORKEAKOULU DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Tuomas Korpilahti		
Työn nimi:	Hajautetun ontologiakirjaston kehittämisarkkitehtuuri		
Päivämäärä:	31.3.2004	**Sivuja:**	60
Osasto:	Tietotekniikan osasto		
Professuuri:	T-106 Ohjelmistojärjestelmät		
Työn valvoja:	Prof. Jorma Tarhio		
Työn ohjaaja:	Prof. Eero Hyvönen		

Tämä työ luokittelee ontologian muutosoperaatiot, jotka rikkovat ontologioiden välisiä riippuvaisuuksia. Työssä kuvataan ontologiakirjastoarkkitehtuuri, joka havaitsee vahingolliset muutokset ja jonka avulla niitä voidaan hallita.

Semanttinen web on tulevaisuutemme älykäs Internet, jossa tietokoneet kykenevät ymmärtämään tallettamansa tiedon semanttiset suhteet. Semanttisten suhteiden perusteella tietoa voidaan liittää yhteen uudella, tehokkaalla tavalla. Ontologiat ovat Semanttisen Webin ydin. Ne mallintavat tiedon taustalla olevien käsitteiden merkitykset ja niiden väliset suhteet. Jotta Semanttinen Web voisi toteutua, tarvitaan yhteisiä, uudelleenkäytettäviä ontologioita tiedon merkitysten kuvailuun. Ontologiakirjastot kokoavat ontologioita yhteen tukeakseen näiden uudelleenkäyttöä ja ontologioiden jakamista.

Tärkein tapa käyttää uudelleen jokin ontologia on sisällyttää se uuteen ontologiaan, joka käyttötapauskohtaisesti tarkentaa ensimmäisen ontologian määritelmiä lisäämällä uusia käsitteitä ja suhteita. Tämä mahdollistaa ontologioiden jakamisen itsenäisiin yksiköihin ja niiden hajautetun kehityksen. Ongelmaksi muodostuvat muutospaineet käsitteiden muuttuessa; alkuperäisen ontologian käsitteitä joudutaan päivittämään, ja muutokset voivat rikkoa siitä riippuvaisia ontologioita.

Tämä työ luokittelee joukon kirjallisuudessa tunnustettuja ontologian muutosoperaatioita. Kunkin operaation seuraukset ontologiariippuvaisuuksille analysoidaan, ja muutosoperaatiot asetetaan tärkeysjärjestykseen vaikutusten laajuuden ja operaation yleisyyden mukaan.

Luokitellut muutosoperaatiot voidaan havaita ontologiakirjastolla, jonka arkkitehtuuri kuvataan. Asiakas-palvelin –pohjainen järjestelmä mahdollistaa virhetilanteiden havaitsemisen heti muutoksentekohetkellä ja tiedon välittämisen virheen syystä ja seurauksista muille ontologioille. Virhetilanteiden käsittelyyn erikoistuneita komponentteja voidaan kehittää järjestelmän tarjoaman rajapinnan avulla.

Arkkitehtuuriin sisältyvät lisäksi julkinen ontologiakirjasto, jossa hyväksyttyjä ontologiaversioita voidaan julkaista, sekä kehitysympäristö, joka takaa yhtenäisen versionhallinnan ontologioille. Ontologian julkaiseminen kehitysympäristöstä julkiseen ontologiakirjastoon tapahtuu hallitun julkaisuprosessin kautta.

Arkkitehtuurista on toteutettu prototyyppi, joka on rakennettu tukemaan Yleisen Suomalaisen Ontologian hajautettua kehitystä Helsingin Yliopistossa.

Avainsanat: ontologiat, ontologiakirjastot, hajautettu kehitys

Acknowledgements

I am grateful to Professor Jorma Tarhio, my supervisor, for his valuable feedback on this thesis.

I would like to express my appreciation to my instructor, Professor Eero Hyvönen, for his guidance during the past few months.

I would also like to thank the people at the Semantic Computing Research Group at the University of Helsinki for the enjoyable discussions, the visionary ideas and down-to-earth criticism that helped me in pulling this all together.

My gratitude goes also to those who helped me to proofread this thesis.

Finally, I would like to thank my family and my fiancée Meri for their everlasting love and support.

Espoo, March 31, 2004

Tuomas Korpilahti

Contents

Glossary of Terms

Affected element	An ontology element that is modified by a change operation, or that contains a reference that is broken by the change
Class	An ontology element representing a categorical concept
IM	Instant Messaging
Index term	A thesaurus term meant for indexing data
Instance	An individual representing a real world item belonging to a certain ontology class
JDBC	Java DataBase Connectivity
MAO	MuseoAlan Ontologia, an upper level museum domain ontology
MASA	MuseoAlan AsiaSanasto, a museum domain thesaurus
Metaclass	A class having an instance that is also a class
MMS	Multimedia Messaging Service
Non-descriptor	A thesaurus term that should not be used to index data. Points to an index term that should be used instead
Ontology	A computer processable model defining the semantics of the concepts and relations of some real world domain
Ontology element	A class, a property, an instance or a metaclass of some ontology
Property	A relation connecting a class or an instance to another class, intance or to a literal value
SMS	Short Message Service
URI	Uniform Resource Identifier
YSA	Yleinen Suomalainen Asiasananansto, a general Finnish thesaurus
YSO	Yleinen Suomalainen Ontologia, a general Finnish upper ontology being developed based on YSA
WWW	World Wide Web

Chapter 1

Introduction

1.1 Motivation

Knowledge modeling with ontologies is one of the key ideas of Semantic Web [1], the next generation of the Web that is currently eliciting much research interest. Ontologies, as defined by Struder and colleagues in [2], are machine comprehensible knowledge models, that explicitly model the concepts and relations of the real world. As Fensel argues in [3], ontologies must be interoperable and they must be publicly shared in order to be able to gain the full benefits of Semantic Web.

According to Gruber's ontology design principles [4], each real world concept is defined only in one ontology, and all systems use the same definition for that concept. This means that ontologies must be reusable in order to let ontology engineers benefit from the increasing number of different domain-specific ontologies. A common way to reuse ontologies is to include them in other ontologies, and then add new concepts and relations. However, when the included ontology is modified, the changes may prove critical for the including ontology and introduce inconsistencies that make the latter unusable. The problem becomes worse when we acknowledge the fact that ontologies are developed in a distributed manner, so that the including and included ontologies are developed in geographically distributed locations by different people. As Stojanovic et al. recognize in [5], Maedche et al. in [6] and Arpírez et al. in [7], a consistent ontology development and maintenance architecture needs to be established in order to manage the evolution of dependent ontologies. Nevertheless, Fensel's note in [8] still holds – methods and tools to support this complex task completely are currently missing.

1.2 Research Problem and Objectives

Ontologies can be reused by including them in other ontologies and then adding new concepts and relations. Severe consistency problems may arise when the included ontology is changed. For example, deleting a concept or a property from an ontology that is included in another ontology causes problems if the same concept or property is referenced in the second ontology. This Master's thesis is to propose how ontology development and evolution could be done in a distributed environment in such a way that whenever a developer performs a change, he is aware of its consequences to the other ontologies that depend on the one he is editing. A specific use case for the thesis is the development of Yleinen Suomalainen Ontologia (YSO, a general Finnish upper ontology).

I will present an ontology library architecture that enables distributed developement of inter-dependent ontologies. For this, I will identify the edit operations that can be harmful and define when they are such. The architecture will provide a mechanism that keeps the ontology developer informed of what consequences his changes will have in ontologies that are dependent on the one he is editing. The goal of the architecture is to create a system that could further be used to develop methodologies and techniques that support the developer in the development process. The methodologies could guide the developer to perform changes in such a way that minimizes the number of conflicts in dependent ontologies. The architecture must therefore enable easy development of extensions for handling the problematic changes.

1.3 Benefits of Study

I identify and analyze the change operations that cause problems with dependent ontologies. We can thus gain further understanding on the problems of distributed development of mutually dependent ontologies.

The architecture can prevent unnecessary problems by warning the developer about the consequences of his changes. As the developer now is notified about hazardous changes, he gets a chance to think for alternative solutions and no longer makes a malicious change by accident or by thinking it would not be harmfull. The number of problems appearing to the reusers of the ontology is therefore smaller.

Catching edit operations that cause problems gives a possibility to guide the developer's actions. It makes possible to propose the developer a set of alternative methods to perform the same operation in such a way that dependency problems are smaller or don't even rise at all. Some suggestions of different methodologies have been made by Stojanovic et al. in [5]. These and other methodologies can be easily developed and tested on the architecture framework.

1.4 Study Methods and Scope

This study is a case study aimed at developing an architecture for ontology development at the University of Helsinki. The specific target was to design a system to help build and maintain a general Finnish upper ontology YSO that is currently under development.

A literature study was conducted to gather and analyze problems in distributed development of dependent ontologies. I interviewed the principal user of the system at the National Library (Kansalliskirjasto) to further gather requirements and to gain insight on the problems of this particular use case. I have also interviewed the key personnel of one large scale ontology building project at the University of Helsinki to use their experience in anticipating the problems that await the YSO developers. Based on the findings, I designed an architecture that allows distributed development of dependent ontologies. The concepts were tested by creating a prototype implementation of the architecture to be used at the University of Helsinki and at the National Library.

The rest of this thesis is structured as follows. First the main challenges that distributed ontology systems face are presented. Then the next Chapter identifies the change operations that may cause conflicts in an ontology dependency chain, and provides a more profound analysis on their effects. After that, we will take a brief look at how those problems are solved today. I then propose an ontology library architecture that recognizes the problematic changes presented and that can be used to help to guide the user to develop better ontologies. To build confidence in the solution, a prototype implemented during the project is presented and evaluated. I will then conlude the thesis with a discussion and a summary of my findings.

Chapter 2

On Ontologies

2.1 Introduction

According to the Computer Dictionary section of HyperDictionary [9], an ontology is *"an explicit formal specification of how to represent the objects, concepts and other entities that are assumed to exist in some area of interest and the relationships that hold among them"*. As the definition suggests, an ontology models different real life concepts and their relationships in a commonly agreed way that can be shared and processed by a machine. Uschold and Jasper [10] further explain the role of an ontology in knowledge exchange by stating that *"An ontology may take a variety of forms, but it will necessarily include a vocabulary of terms and some specification of their meaning. This includes definitions and an indication of how concepts are inter-related which collectively impose a structure on the domain and constrain the possible interpretations of terms."*.

The two statements above underline the aspect that an ontology introduces a structure on a domain of activity. We can readily find important application possibilities for such domain structurization. On one hand, ontologies can be used to index documents by linking the documents to the concepts that are most relevant to the contents of the document. This process is called annotating the documents. On the other hand, given a document, different ontologies can be used to find other documents with closely related information by following the relations between the concepts in the ontologies. The relations define the semantics of each document, that is, how it is related to other documents in the universe. The semantic links carry specific information on what the links mean – we are able to know the context of related things and in what role each document is with regard to others. Having that information in hand allows search engines to infere what information we might be currently interested in by looking at the history of the documents that we have been viewing. Based on that reasoning the search engines can intelligently sort search results, leave out irrelevant results, or even propose us to view some documents that would not be found with a normal search but could be interesting because their semantic links are closely related to the documents we are currently viewing [3].

2.2 Making Semantic Web Happen

In order to deliver the promise of intelligent indexing and information retrieval, the ontologies used to index the documents should be publicly shared. Otherwise, if each document would be indexed according to its private domain ontology no semantic links could be established between the documents. Of course, closed systems could be constructed on top of private ontologies but they would be unable to share information and pass it from one system to another without individually tailored interfaces between each system. Ontology sharing is hence an important issue in Semantic Web. Ontology library systems as described by Ding and Fensel in [8] are a means of sharing ontologies to a wide user base. They are systems that aim to facilitate ontology sharing by gathering a set of ontologies into one place. Ontology reuse is another main function of an ontology library. Existing library implementations have different strategies to support ontology reuse, ranging from a simple approach of copying existing concepts as a basis for a new ontology to advanced ontology inclusion and merging capabilities. Partly related to ontology reuse are the libraries abilities to structurize the library contents – the ontologies, that is – according to some pattern. Some libraries provide ontology structurization by using standard upper level ontologies [11, 12] while others [13, 8] use less advanced means such as directory hierarchies. Library structurization is a stepping stone to ontology management services, which become an important feature of an ontology library as the number of ontologies in the library grows.

Ontology libraries group ontologies together to provide a common access point to a set of ontologies. By promoting a single access to ontologies the libraries allow different parties to acquire the same set of ontologies as a basis for their applications. But as always, each use case is different and is likely to need a customized ontology. Thrust from applications thus pushes towards case-specific ontologies, but the system as a whole will not function if the ontologies are not shared. Would there be a resolution for these conflicting needs? To be of any use, ontologies must be specific enough to support the individual use cases, which may pose even conflicting requirements to some concepts. At the same time, ontologies should be general enough to allow the concepts of each use case to be linked to the concepts of the other use cases. The answer is to create some general level ontologies, and then reuse and modify them to form case specific ontologies. The general level ontologies provide the connecting framework that allows semantic links between different case specific concepts. The details of each particular case can be hidden inside the case specific ontologies. Promoting ontology reuse is indeed another important goal of ontology library systems.

Modeling a domain by defining its concepts and their relations is a non-trivial task; it requires domain expertise and is heavily affected by the intended use case of the ontology. As is the case in any modeling, given the same problem domain two different persons can come up with completely different ontologies. Ontology reuse is thus encouraged to benefit from the work already done in order to speed up ontology development and cut down costs. By re-using the existing ontologies the information annotated according to different application specific ontologies can be put into a wider perspective and combined in innovative ways. A reuse method that is similar to software engineering methods is to design ontologies as compact, independent modules. These modules can then be combined to create more and more complex systems. Modularization of an ontology library is an effective way to support ontology reuse, as Ding and Fensel [8] point out.

Ontology modularization allows another important aspect of ontology library design, that is the use of standard upper level ontologies. A standard upper level ontology is a general ontology that defines general concepts that occur often in the domain ontologies that are used in applications. Its function is to tie together the different bits and pieces of the re-usable ontologies by providing a common framework that can be referenced from the individual, re-usable modules. Applications can then use the upper level ontology to find relations between two semantically distant concepts. In essence, as ontologies structure the world, an upper level ontology is critical to structure the ontology library itself [8]. However, there is something that starts to tear down this powerful and expressive system into pieces even before it has been completed. This demolishing force is called change.

The next sections will take a closer look at one of the principal methods of ontology reuse, including existing ontologies into new ones. They show how it is done and what it means for the concepts and for the ontologies themselves. Then we will consider the challenges the combined ontology structure faces when a change is introduced in the system. A clear example illustrates the problems that are caused by ontology reuse by inclusion. These problems will be further analysed in the next chaper.

2.3 Ontology Re-Use by Inclusion

Ontologies can be reused in other ontologies by including them in each other ontology as a part of the other ontology. The inclusion is done by including all definitions of the original ontology in the other ontology. The developer can then add more definitions to the other ontology to tailor it for its special use case. The new definitions can refer to and reuse the concepts and relations defined in the included ontologies. Inclusion means that the original definitions of the included statements remain in the included ontology. Thus, any time the original definitions are changed, the changes affect all including ontologies. Note that we are not only copying the statements into the new ontology but want to base the new ontology on a set of re-usable modules.

Definition: Ontology A includes ontology B if and only if A includes all of B's statements.

An ontology could be included in another ontology partially, also. However, as Maedche et al. point out in [6], it is much more difficult to prove ontological consistency in the case of partial inclusion as some statements might be missing that would be required by the included statements. Therefore, I have defined ontology inclusion to mean including the entire ontology. Ontology inclusion could be cyclic, too, so that ontology A includes B and B includes A. Cyclic inclusion chains are not considered because changes propagated through the cyclic chain would produce too complex problems. For the rest of this thesis, ontology inclusion is never cyclic.

A need for a change imposes great challenges for ontology inclusion. As the world changes, the reused ontologies must be adapted to the changed world or they become obsolete. But there are other ontologies that are affected by the changes that are made to the reused ontologies, namely, the ontologies that are re-using them. They have become dependent on the ontologies they included for reuse. Changes in the included ontologies may invalidate the dependency relations from the including ontologies to the included ones. Of course, instead of including an ontology we could merely copy its contents to the new ontology. This time the original ontology could be changed freely, and the new ontology would remain consistent because it would have its own definitions of the concepts defined in the original ontology. Eventually this approach would lead to a chaos; each ontology would need to be modified separately. There would be no guarantees that the change would be implemented similarly in all including ontologies. As these ontologies would in turn be reused, we would soon be having a multitude of ontologies containing different versions of the same concepts with no way to tell what they really mean and how they are related to one another. That is why ontology reuse should be done by including ontologies instead of simply copying their contents. As a consequence, when changing a reused ontology there is always a risk that some other ontologies might become broken due to the change.

2.4 Problems from Inclusion

Ontology inclusion is a powerful and widely accepted way of re-using ontologies. It can, however, lead to development problems when an included ontology is changed. The following example clarifies the cause of problems, and introduces one problem type.

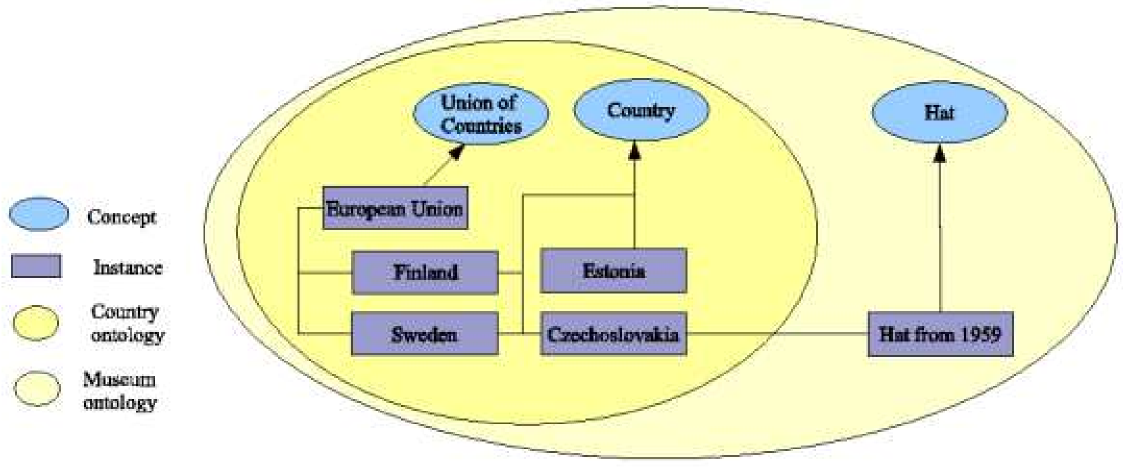


Figure 2.1: Museum ontology including country ontology.

Assume an ontology of countries, which contains concepts such as "Country" and "Union of Countries". As instances, the ontology would have, for example, "Finland", "Sweden", "Estonia" and "European Union". Let’s now build a museum ontology which contains museum artifacts, and connects them to the country in which they were created. To reuse the countries already defined, the museum ontology includes the country ontology. Suppose we have in a museum a hat that was manufactured in Czechoslovakia, and we would like to add it to our ontology. We would first add to the museum ontology a concept "Hat", create an instance of it, and then associate it with the instance "Czechoslovakia". The museum ontology is now dependent on the country ontology, as Figure 2.1 depicts.

As Czechoslovakia no longer exists but was divided into the Czech Republic and Slovakia, the developers of the country ontology remove it from the ontology. The situation is illustrated in Figure 2.2. This change breaks the museum ontology because it contains a reference to an instance that no longer exists. A change made and verified to be safe by one party has thus caused a serious problem to another party.

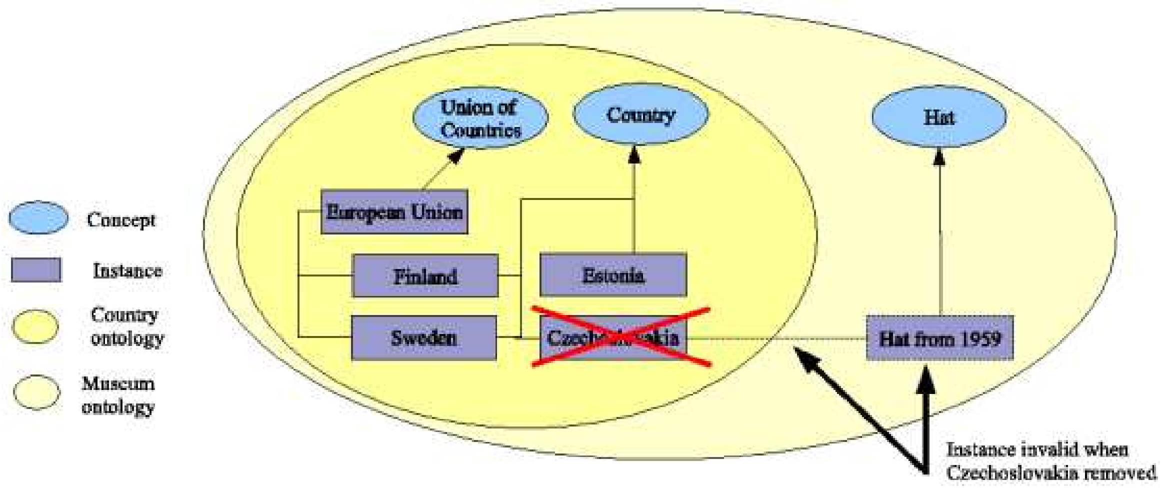


Figure 2.2: Museum and country ontologies after removal of Chechoslovakia.

In the example the effects were seen at the instance level but we run into similar problems at the class level very easily. This is because ontologies are a model of the world, and need to be adjusted to fit current understanding of the world at each point in time. Concepts do become outdated and are made obsolete by other concepts. Sometimes the concept remains but its meaning changes with time. Ontologies thus need to be changed. Until it is possible to detect what kind of effects a change would cause to other ontologies, ontology engineers are unable to evaluate the results of a change, and thus are incapable of comparing several changes that could be performed to fulfill the need for change.

Chapter 3

Problem Analyses

3.1 Introduction

In this Chapter I will focus on the problem of ontology evolution and changing dependent ontologies. Changes introduced by ontology evolution become harmful if they affect other ontologies that depend on the one that is being changed. Ontology dependency boils down to a reference from one ontology element (a metaclass, a class, a property or an instance) to another, which is defined in a different ontology. Because the referenced ontology must contain the complete definition of the element, the reference must be either from an element "lower" in the element hierarchy to an "upper" element (an instantiation or a subconcept relationship) or a direct reference where the referenced element is the value of a property in another ontology. I have thus focused my analyses of a change operation to the children of the changed element – its subclasses and instances – and to the direct references pointing to the element.

Noy and Klein [14] have identified ontology change operations and analyzed them with respect to the instance-data preservation dimension. They define 22 change operations, which can be grouped to the following categories: creating and deleting things, moving a class (a property) up or down the class (property) hierarchy, adding and removing superclasses, connecting and detaching class and property hierarchies (adding a property to a class / removing a property from a class), re-classifying a class as an instance or vice-versa, altering a restriction for a property, declaring two classes disjoint, moving properties to another class, and merging and splitting classes. For these cases, Noy and Klein provide valuable information by describing what data would be lost at the instances in the changed ontology and further are there means to avoid the data loss. However, a closer inspection of the change operations they have identified reveals that some of them can be composed from the other change operations by performing the other operations in a certain sequence. As a consequence, in the analyses presented by Noy and Klein the same consequences are presented several times for different composed operations. In my opinion, this suggests that a more expressive solution would be to define a set of atomic change operations that perform fundamental changes to the ontology structure, and then analyse the more complex operations by looking at the combined effects of the atomic operations that can be used to compose them.

In the following section I present a set of atomic operations and analyse their effects on the ontology that is changed. I show how they can be combined to form the more complex operations suggested by Noy and Klein. In doing so I extend the analyses of Noy and Klein with the class and property hierarchy aspects, i.e. how the class and property models are affected by the change operations. I analyse the operations from the viewpoint of inter-dependent ontologies, i.e. what are the effects of a change with respect to the instance-data perservation dimension and with respect to the class and property model consistency when the change affects an element that is referenced from another ontology. Having completed the analyses of change operations, I move the focus of the thesis towards the use case by briefly describing the problems and experiences obtained by interviewing the future principal user of my system and the key personnel of a similar project at the University of Helsinki. Based on the theoretical analyses and on the practical problems, I prioritize the need of support for the different change operations. A summary of the results concludes this Chapter.

3.2 Change Analyses

Some of the change operations presented in the analyses of Noy and Klein can be decomposed into a more primitive operations. These atomic operations can be performed in some sequence to produce the effects of the compound operation. For example, an operation of merging a set of classes into one class can be decomposed into adding all the properties of one class to another, then changing all the instances of the first class to be instances of the second class, moving all property values pointing to the first class to point to the second class, and then deleting the first class from the ontology. The same process can be repeated several times to merge more classes. The next section suggests a set of atomic operations that could be used to compose the compound change operations. One might argue that some atomic change operations could be again decomposed into other operations, for example, adding a property to a class is in fact a special case of widening a restriction for a property. However, those operations are very fundamental in nature to build ontologies and therefore deserve a more detailed analyses.

3.2.1 Atomic Changes

Creating Things Creating something that does not exist before is definitely an atomic change. Creating a new class, property or instance adds a new URI to the ontology. By definition the URI cannot have existed before in the ontology where it is being created. However, it could be already used in another ontology. If there is an ontology that includes the changed ontology and already contains an element with the same URI, this results in a naming conflict. A Semantic Web practice is that any two things with equal URIs are considered to be the same thing. If the types of the two things are different – for example, "http://foo#bar" is a class in an ontology I and a property in an ontology L, and L includes I – it is not clear what is the meaning of the concept in the including ontology L. A conflict is thus introduced into the including ontology L.

Deleting Things Deleting a class, property or an instance from an ontology removes the URI and all references to it from the ontology. When deleting a class C, we must decide what to do with its subclasses and instances. They can either be deleted or set to become the subclasses and instances of the superclass of C or of the root class as Maedche et al. illustrate in [15]. The first strategy simply increases the number of deleted elements. It deletes all instances of C, all of C's subclasses and all of their instances. It is a strategy that causes the maximal amount of data loss. Even though a delete operation removes all references to the deleted things in the changed ontology, it does not modify the including ontologies. They remain as they were and become broken if they reference the deleted elements or any of their subelements or instances. In the latter approach the instances of the class C have a less specific type and lose the values of the propertied defined in C. The subclasses lose the properties defined in C. All properties lose C from their range and domain. In the ontologies that include the modified ontology instances still contain values for the properties that were defined in C, and direct subclasses of C keep declaring C as their superclass. As C no longer exists, all elements referencing to it become broken. The change is a data-loss change at both class and instance levels, and may break the class model in the referencing ontologies.

Deleting a property P removes the property from all classes, and the value of P for all instances is lost; again a data-loss change. For the subproperties of P the change equals the removal of a superclass: According to the selected strategy they are either deleted or they become the subproperties of P's superproperty, losing the domain and range inherited from P. The effects to the property hierarchy are identical to the effects to the class hierarchy explained above. In addition to the broken property hierarchy, there may be classes in the including ontologies that have P associated to them. These classes become broken. Deleting an instance I invalidates in including ontologies those instances that have I as the value of some of their properties. The class model is not affected.

Adding a Property to a Class When a property P is added to a class C, the instance data remains intact. The subclasses of C inherit the new property. Even if a subclass $SubC$ already has the property P, normally no problems arise. Inheritance is used to attach P to $SubC$ and the definitions between P and $SubC$ refine the relationship. However, as Noy and Klein point out in [14], if the property restrictions inherited from $SuperC$ are incompatible with the local restrictions for the same property in $SubC$, then the property values may become invalid and the operation would be a data-loss operation. This can happen in the including ontologies, also, but otherwise they are not affected. As the property P is added to the class C, C is added to the domain of P and of its subproperties. This widens the group of classes that are associated with the subproperties of P, and thus causes no problems.

Removing a Property from a Class Removing a property P from a class C removes it from the subclasses of C, too. Instances of C (and of its subclasses) lose their values for the property P. The class C is removed from the domain of all subproperties of P, and the instances of C and of the subclasses of C lose all values for those properties. If the range of P is a class or an instance R, removing P from C removes a semantic relation between C and the range class or instance R. The situation becomes very unclear here, because the key idea behind re-using ontologies is to reuse and enrich the semantic information stored in them. When this information is altered, the meaning of the changed ontology changes. If the meaning is changed enough, the entire context of the changed ontology may change. It should no longer be used in its previous context but in some other context. As a consequence, the semantics of the change in the including ontologies are unknown. At worst, the change can break the semantics of ontologies including the changed ontology rendering them useless. In addition, the including ontologies continue to reference to P from the instances of C and of its subclasses. Those references are broken. The change loses data on instance level, may break the semantics of the class model, and may introduce invalid references to referencing ontologies.

Adding a Superclass or a Superproperty Adding a new subclass-superclass link between a subclass $SubC$ and a superclass $SuperC$ adds to $SubC$ new properties inherited from $SuperC$. In most cases, inheriting new properties is equivalent to adding properties, which was shown to be safe in most cases. However, the warning said about adding properties that already exist in the subclasses should be beared in mind. Another effect of adding the subclass-superclass link is that the identity of $SubC$ (an all of its subclasses) is changed as it inherits the identity of $SuperC$ (it can be treated as $SuperC$). The change does not restrict the current use of $SubC$, so it is considered a safe change. The property hierarchy and instances are not affected by the change.

If a new subproperty-superproperty link is added between a subproperty $SubP$ and a superproperty $SuperP$, the same applies to the property hierarchy. $SubP$ inherits the domain and range of $SuperP$, which essentially means widening the restrictions for $SubP$. $SubP$ can now be used in place of $SuperP$. Possible problems may arise if there are property restrictions in the two properties that are not compatible with each other. For example, if one of the properties is defined to be transitive or symmetric and the other is not, there may be some values for the properties that break the transitivity or symmetricity restriction. These values would become invalid. Because one must be able to use all subproperties in the place of their superproperties, all subproperties of $SubP$ – defined in the changed ontology or in the included ontologies - comply with the restrictions posed by $SubP$. Therefore, a validity check can be done in the changed ontology by checking if there would be a conflict between the property restrictions of $SubP$ and $SuperP$. If a conflict is not found, the change is safe. Otherwise, the change poses the same problems in all including ontologies as it does in the changed ontology.

Removing a Superclass or a Superproperty When a subclass-superclass link between a subclass $SubC$ and a superclass $SuperC$ is removed, $SubC$ no longer has the properties it inherited from $SuperC$. We have already looked at the effects while talking about removing a property from a class. In addition to that, the identity of $SubC$ changes as it can no longer be treated as $SuperC$. The change makes invalid all property values belonging to a property that has defined $SuperC$ of an instance of $SuperC$ as its range and has $SubC$ or an instance of $SubC$ as its value. These properties may be defined in other ontologies as well.

Removing a subproperty-superproperty relationship between a subproperty $SubP$ and a superproperty $SuperP$ causes the same changes in the property hierarchy. In addition, as $SubP$ no longer inherits $SuperP$, the domain and the range of $SubP$ are decreased. Decreasing the domain of $SubP$ essentially removes $SubP$ from the classes of $SuperP$'s domain. Decreasing the range causes a problem if $SubP$ has at some instance a value that no longer belongs to the range of $SubP$. Thus, removing a superclass causes data loss on instance level and may break the class model. These problems arise in the including ontologies, also. The change does not cause any broken references.

Re-classifying an Instance as a Class As an instance I is re-classified as a class, its type changes. The number of instances in the ontology is decreased and the number of classes increased by one. There is no change of URIs. The change invalidates property values that have I as the value for properties that have an instance as their value type. Essentially the change may cause a property value to break a property restriction. Such property values and restrictions may be in the changed ontology or in the including ontologies.

Even though re-classifying an instance as a class does not necessarily cause data loss, re-classifying an element as another type of element fundamentally changes the semantics of the re-classified element. It is thus questionable if other ontologies can continue to reference that element without an experienced ontology engineer validating first that the concept is still relevant to the referencing ontology and reference context. Human intervention should be considered each time before the changes are automatically propagated to the including ontologies.

Re-classifying a Class as an Instance An inverse operation to re-classifying an instance as a class is re-classifying a class as an instance. Its effects are not comparable, however. If a class C is re-classified as an instance, all instances of C become become instances of the superclasses of C. This means that they become less specifically typed and lose the properties defined in C. The subclasses of C become the subclasses of C's superclasses. The identity of C itself is then changed. The re-classification invalidates property values that have C as their value and belong to a property that has a class as its value type. The same problem appears in the including ontologies. In addition, if the including ontologies define subclasses for C the class hierarchy is broken for those classes. This in turn breaks the instances of those classes and any properties that have C, the subclasses or their instances in the range. Finally, as was discussed above the semantics of C change considerably and the context in which C is reused in the including ontologies should be revised by an experienced ontology developer.

Declaring Classes C_1 and C_2 Disjoint Declaring classes C_1 and C_2 disjoint means that they have no common instances nor common subclasses. Thus, the change invalidates instances that belong to both C_1 and C_2, and all common subclasses. If an including ontology defines a common subclass $SubC$ for C_1 and C_2, it and all of its instances become invalid. A more severe effect is that the semantics of the class model have changed. Again, reusers of the changed ontology should revise how they have reused C_1 and C_2 to ensure that their new meaning is relevant in the context of the including ontology's use case. One could assume that if there exists common subclasses or instances in the including ontology then the reuse context of the changed ontology is quite different from the one originally considered by the changed ontology developers.

Defining a Property as Transitive or Symmetric When a property P is defined as transitive or symmetric, all property values that violated the transitivity or symmetry become invalid. This can happen either directly in the values of the property or in the values of some subproperty of P. The symmetric relationship is likely to pose more problems, as it would require that the domain and the range of P (and of all of its subclasses) are identical, and that each instance that has a value R for P is the value of P in R. Such a property network can be non-complete simply because of human factors – the ontology developer may not have remembered or simply bothered to add the property values everywhere, or has simply made a mistake while adding the values. This problem applies in the changed ontology as well as in the including ontologies, and each ontology must be checked separately.

Widening a Restriction for a Property As we have already seen, properties can have restrictions, for example the number of allowed values, the number of required values, domain of the property, range or the property etc. Widening a restriction is a generalization of a change operation that means making the restriction less strict, for example increasing the number of allowed values, adding a class to the range or replacing a range class with its superclass. The effect is that the slot can be used more freely, and all current associations are preserved. Including ontologies that use the property include the wider restriction and the existing ontological relations remain intact. The change does not pose restrictions the subproperties of the changed property. The change is thus safe.

Narrowing a Restriction for a Property Narrowing a restriction of a property is the opposite of widening it, e.g. increasing the number of required values, removing a class from its range or replacing it with a subclass etc. Existing property values that violate the narrower restriction become invalid. The subproperties of the changed property may introduce such values, also. As each subproperty chould be used in place of the changed property, their values should conform to the restrictions posed by the changed property. Because a restriction was narrowed, the values of the subproperties may not fulfil the new, stricter restriction and a conflict is arosed. The violations can occur in the current ontology as well as in including ontologies.

The change operations described above form the set of atomic changes, that is changes that can be used to compose all other type of changes that one is able to perform on ontologies. In the next section we shall take a look at the composite changes and how their effects can be analysed.

3.2.2 Composite Changes

A composite change can be composed from a number of atomic changes by performing the atomic changes in some order. Often there exists some freedom of choice on which order the atomic changes can be performed but the result of the operation can also depend on the execution order. It is obvious that the effect of the composite change cannot be greater than the combined effects of the individual changes composing it. However, an important observation is that in some cases the effect can be smaller. For example, when a class is moved to another branch in the class tree it first loses the properties it inherited from the old superclasses and then inherits the properties from the new superclasses. Some of the lost properties may be reinherited in the second phase; this happens if the properties belong to the new superclasses. In such cases, the values of those properties need not be lost – the system would thus need to remove and add only a subset of the properties that the atomic changes would require. Therefore, there is an upper limit to the effects of a composite change. The effects cannot be greater than the effects of each atomic change performed sequentially on the ontology. The composite change approach allows us to minimize the effects of different composite changes by breaking them into atomic changes and then eliminating redundant changes from the change sequence. This is equal to calculating the delta, i.e. the difference between the state of the ontology before and after the composite change. The analyses presented in the previous section allows us to focus our search of difference on a subset of the ontology elements. In the following paragraphs we will investigate different composite changes and see how their effects can be minimized.

Moving a Property Up the Class Hierarchy Moving a property P up the class hierarchy means that P is moved from a subclass $SubC$ to a superclass $SuperC$. In other words, P's domain class $SubC$ is replaced with its superclass $SuperC$. The change is equal to removing $SubC$ from the domain of P and then adding $SuperC$ to the domain of P. As we consider the change sequence from $SubC$ point of view we notice that delta of its state before and after the composite change is an empty set, that is, from $SubC$ point of view nothing needs to be changed. The class $SubC$ still inherits the property P, and no data need be lost. However, $SuperC$ might have other subclasses and some of those subclasses or their subclasses might already contain P. In this case the property restrictions need to be checked for consistency, as was notified while discussing adding a property. Such subclasses could be defined in the including ontologies, also. Moving a property up the class hierarchy is thus a relatively safe operation, provided that the property is not used elsewhere in the subtree.

Moving a Property Down the Class Hierarchy When a property P is moved down the class hierarchy, it is moved from a superclass $SuperC$ to its subclass $SubC$. Effectively, the domain class $SuperC$ is replaced by its subclass $SubC$. As with moving a property up the class hierarchy, in this case also the change can be decomposed into successive remove property and add property operations. The property P is removed from $SuperC$ and then added to $SubC$. After the change the class $SuperC$ no longer has the property, and all instances of $SuperC$ lose their value for P. The subtree starting from $SubC$ perceives no change, and is not affected by the change. This combined effects are thus equal to removing a property but ruling one subtree out from the set of affected elements. The effects were discussed earlier as we looked at removing a property from a class, but in this case we have managed to limit the number of affected elements by being able to state that the concept tree starting from $SubC$ is not affected by the change.

Changing the Superproperty of a Property to a Property Higher in the Property Hierarchy Changing the superproperty of a property P to a property higher in the property hierarchy means that the superproperty relationship of the property P is moved to point to a superproperty of its current superproperty $SuperP$. The operation can be decomposed into sequentially removing the subproperty-superproperty link between P and $SuperP$, and then adding a subproperty-superproperty link between P and the superproperty of $SuperP$. As a result, P becomes the brother of its ancient superproperty $SuperP$ and no longer inherits the domain and range of $SuperP$. The lost domain classes lose P from them, and their instances lose all values for P. This was discussed while considering the effects of removing a property from a class. As it was seen in that case, the same problems apply for all subproperties of P. Some of these subproperties can be defined in the including ontologies. This time the effects are limited to the domain and range of $SuperP$ – P still inherits the domain and range of the superproperty of $SuperP$, and all of its superproperties.

Changing the Superproperty of a Property To a Property Lower In the Property Hierarchy When the superproperty of a property P is changed to a property lower in the property hierarchy, the superproperty relationship of property P is moved to point to a subproperty $SubP$ of P's current subproperties. The composite operation can be decomposed into changing the subproperty-superproperty link between $SubP$ and P to be between $SubP$ and $SuperP$, removing the subproperty-superproperty link between P and $SuperP$, and then adding a subproperty-superproperty link between P and $SubP$. The delta of the composite operation shows that $SubP$ no longer inherits P, and P inherits $SubP$. From the point of view of $SubP$, the change is equal to moving it up the property hierarachy, which was discussed above. From the point of view of P, new classes may have been added to its domain and range definitions. The change breaks the property hierarchy if P does not fulfill all value restrictions defined in $SubP$. This is because a subproperty must fulfill all value restrictions of its superproperty.

Moving a Property from a Class C_1 to a Referenced Class C_2 A property P is moved from a class C_1 to a class C_2 referenced by C_1. The operation can be performed by adding P to the class C_2, copying the values for P at the instances of C_1 to the instances of C_2, and then removing P from C_1. In [14] Noy and Klein suggest that no data is lost if the values for P at instances of C_1 are moved to instances of C_2. Moving the values as was described can only be done if there exists one instance of C_2 for each value of P at the instances of C_1. If several instances of C_1 reference the same instance of C_2 and have different values for property P, more instances of C_2 must be created or data is lost. Creating new instances could affect the semantics of the changed ontology, there again that is exactly what moving a property to a referenced class does; it moves a semantic relation from one concept to another. That is why the ontology reuse context should be revised after the change. Also, even if the change can be made to retain all information in the instance level in the local ontology, the instances of C_1 and of its subclasses in including ontologies still contain references to P. For them, the change equals removing the property P from C_1. This can cause problems and it has been discussed before.

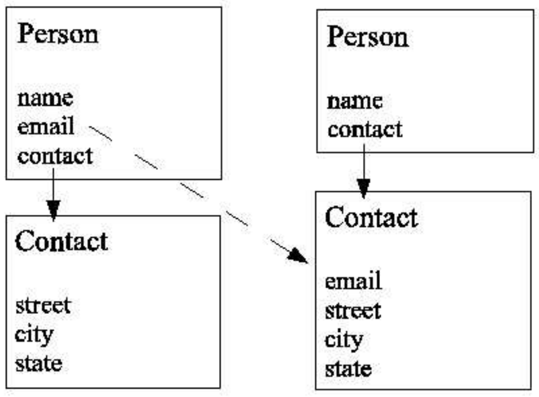


Figure 3.1: Moving a property P from a class C_1 to a referenced class C_2 [14].

Moving a Property from a Class C_1 to a New Class Moving a property P from a class C_1 to a new class N means creating a new class N, adding a reference from C_1 to N, and then moving the property P from C_1 to N. This time new instances of the new class N are created so that the data can be transferred without loss. However, rest of the problems of the previous section still remain. Such a structural change cannot be automatically carried out in the including ontologies but their structure is broken. This means that if C_1 or its subclasses are subclasses or instantiated in the including ontologies, they are broken by the change.

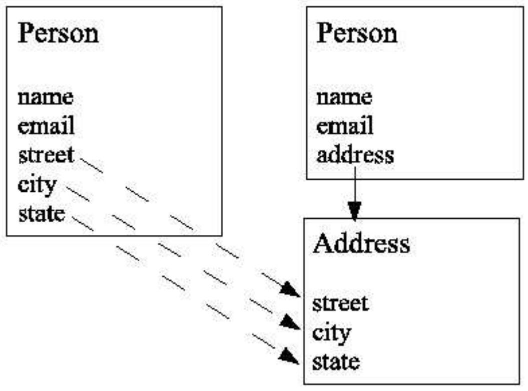


Figure 3.2: Moving a property P from a class C_1 to a new class [14].

Changing the Superclass of a Class to a Class Higher in Class Hierarchy Changing the superclass of a class C to a class higher in the class hierarchy means that the subclass-superclass relation between a class C and its superclass $SuperC$ is replaced by a relation between C and a superclass of $SuperC$. Therefore, C becomes the brother of $SuperC$ and no longer inherits $SuperC$. This is equivalent to sequentially removing the subclass-superclass relation between C and its superclass $SuperC$ and adding a new subclass-superclass relation between C and a superclass of $SuperC$. The effects were explained while we were looking at removing a superclass, but are limited to the properties inherited from $SuperC$ and to the use of C as $SuperC$. It is therefore enough to check for problems with regard to one ancient superclass, only. Unfortunately, the effects cannot be verified by looking at the changed ontology because the same problems can be raised in the including ontologies.

Changing the Superclass of a Class to a Class Lower in Class Hierarchy When the superclass of a class C is changed to a class lower in the class hierarchy, the subclass-superclass relationship between a class C and $SuperC$ is moved to a class lower in the class hierarchy; i.e. to a class $SubC$, a subclass of C. Effectively, the subclass-superclass relationship between $SubC$ and C is changed to be a relationship between $SubC$ and $SuperC$. For C, after the operation it has possibly inherited additional properties from $SubC$ but no data is lost. For $SubC$, which used to be a subclass of C, this case is equivalent to changing the superclass to a class higher in the class hierarchy. This case was discussed above. So, this time we are looking at the same operation from the C's point of view. For C there are no problems as long as the property restrictions match.

Moving a Class in the Class Hierarchy Moving a class C is to another location in the class hierarchy is a very common ontology change operation. However, it can be easily decomposed into a sequence of superclass removals and additions. The consequences of the move operation are essentially defined by the difference in the set of superclasses C has before and after the operation. Let S_o denote the set of superclasses of C before the operation and S_n the set of superclasses after the operation. By definition, S_n cannot be equal to S_o because that would mean that the class was not moved at all. If S_n contains S_o, the operation is safe. In such case only problems can emerge from the property restrictions of the properties in the new superclasses that are not contained in S_o. If S_n does not contain S_o, the operation becomes a superclass removal. C loses all superclasses that are in S_o but not in S_n, and with them C loses all properties inherited from those classes. Some of the properties may be reinherited from the new superclasses in S_n. However, after the operation C can no longer be used in place of its old superclasses, which can invalidate properties and property values as was explained when analysing the removal of a superclass. This may break the ontology structure of the including ontologies, and most certainly causes data loss. The move operation can thus be either a safe operation or a data loss operation that breaks the ontology structure of the including ontologies.

Merging Classes When a set of classes are merged the superclasses, subclasses, and properties of the merged class are the union of the superclasses, subclasses, and properties of the original classes. This can be done by creating a new class, adding it as a subclass of all superclasses of one class to be merged, moving all properties of the class to be merged to the new class, changing all instances of the class to be merged to be the instances of the new class, changing all references to the class to be merged to point to the new class (a reference to a class means that the class is in the range of some property, or is a value for some property at some instance), and then deleting the class to be merged. The process is repeated for each class to be merged. Instance data can be preserved if property values are moved to the instances of the merged class. However, if the original classes contained conflicting property restrictions for same classes, these values cannot be moved to the new class. As the classes are merged their original URIs are removed as well as the URIs of their instances. Including ontologies are broken, if they reference those URIs. If the classes to be merged are subclasses or instantiated in the including ontologies, the structure of the including ontologies is broken for those parts. If the including ontologies define properties that have in their domain or range any of the classes that were merged, those properties together with their subproperties become broken. Merging classes thus breaks all including ontologies that refer the removed URIs in any way.

Splitting a Class A class can be splitted into several classes by splitting the properties into the new classes. The properties of the new classes then specify which instances belong to which class. All property values at the instances can be moved but it might be necessary to split instances, too, if they contain properties from two new classes. While doing so the original URIs of the instances would be removed. In any case, the URI of the original class must be removed. If the including ontologies reference the removed URIs, they become broken as was explained while analysing the deletion of concepts.

3.3 Experiences from Building a Museum Ontology

My research group had previous experience on building a large upper level ontology in another Semantic Web project. I interviewed the project manager, the domain expert and the ontology engineer of the project to gain more insight on the problems that could stand in the way of YSO developers.

Prior to my study, the research group had carried out a research project to create a semantically linked web site from the collections of three Finnish museums. A museum domain ontology MuseoAlan Ontologia (MAO) was created from the museum domain thesaurus MuseoAlan AsiaSanasto (MASA) [16] to classify the domain. Currently MAO contains 6500 classes and 10 properties. Individual ontologies were created for each museum to annotate the collections of the museums. Those ontologies include MAO and define instance data of the collection objects only. The number of instances in each ontology varies from 1200 to 2100, totalling 4900 instances for the three ontologies.

The development of MAO corresponds much to YSO development. YSO is about to become a shared, national upper level ontology. As MAO, it is based on a thesaurus (Yleinen Suomalainen Asiasanasto, YSA [17]). YSO will be referenced from many ontologies, exactly as MAO is. A difference between YSO and MAO is the size of the ontology. YSA contains approximately 14000 index terms and 3000 non-descriptors whereas MASA contained "only" 6000 terms. The development records of MAO are thus a good source of information for estimating the problems that YSO development is likely to face in its first phases.

During the museum project it had become clear that most problems would arise when a URI that existed in MAO was changed or removed. This one change could in turn invalidate hundreds of instances in the annotation ontologies, and it was discovered difficult to find out exacly where the invalid instance data lied. Another observation was that during the project the class hierarchy of MAO was heavily changed and reconstructed. This posed problems in the including ontologies because they contained instances that had property values, which became invalid as the MAO classes to which they pointed were no longer in the range of the properties. Also, the application development that was running in parallel with the ontology development had suffered a great deal from the instability of the class hierarcy, since the changes could invalidate the inference rules written in Prolog that described how the ontology was to be used in the application.

To summarize, the results of the interviews of the key personnel in a similar real-world project suggested that the problem area would be twofold. On one hand, the experience from the museum project leveraged problems related to the removal of a URI from an included ontology. On the other hand, the modification of the class hierarachy had been done quite often, which had led not only to ontology consistency problems but above all to problems in application development. Even though application development is not the precise subject of this thesis, it is a crucial aspect if Semantic Web is about to emerge - if there are no applications that can use ontologies, there is no Semantic Web, and thus no need for ontologies. Also, heavy restructuration of an included ontology is likely to affect the way in which it should be reused by the including ontologies, and is therefore an important concern for high quality ontology development environments.

3.4 Problems from Practice

The thesaurus Yleinen Suomalainen Asiasanasto (YSA) [17] is maintained by the National Library. When YSO is finished, the administrators of YSA will take on YSO maintenance responsibilities according to the project plan. I interviewed the main adinistrator to find out what kind of changes are most common in the maintenance phase. The maintenance process of YSA is quite rigid. All change requests requires a throughout disquisition on the use of the term in question. The disquisition can involve several domain experts. After the disquisition, the change request is handled in a board of directors and experts. If the request is approved, the change is made to YSA. Otherwise the change request is archived for later use.

Mostly modifications are adding new terms. Delete operations occur virtually never. The administrator recalled two or three term deletions during the past ten years. The relations between the existing terms are modified more often, as the semantics of existing terms change. The introduction of new terms or the changed semantics of existing ones may cause merging and splitting terms in the thesaurus. A major problem was conceived to be expressing terms with multiple semantical meanings. For example, "model" can be a profession of a person performing in a fashion show, or a description of some system.

The architecture for a system to be used in YSO development needs to support a controlled process to publish changes, yet it must feature an environment where it is easy to try different modeling solutions. Most operations during the maintenance phase are adding new concepts that often make existing concepts obsolete. Merging and splitting concepts can occur sometimes.

3.5 Prioritization of Changes

The theoretical analyses as well as the experiences from a previous upper level ontology project suggest that the most critical issue of ontology dependence problems is the modification or removal of a URI. Deleting a class, a property or an instance loses data both in the changed ontology and breaks the including ontologies. Changing a URI breaks the including ontolgies.

Changing the class or property hierarchy by removing a superclass or a superproperty, by moving a class in the hierarchy or by moving the superclass of a class or the superproperty of a property to an element higher in the hierarchy is the next important thing. It changes the identity of the classes (or properties) and the direct type of its the instances, which can severely break the including ontologies. The changed direct type of instances can lead to invalid property values in including ontologies. Removing a property from a class can introduce a large loss of information, and is therefore ranked next. Moving the property down the class hierarchy produces the same results for the classes from which it is removed.

The experiences from MAO development lift up narrowing a restriction for a property to be the next most important change. It can cause serious data loss but primarily introduces into the including ontologies conflicts that are difficult to find. Narrowing a restriction is closely followed by the need of support for merging and splitting classes, as those operations are likely to be one of the main causes of problems during YSO maintenance phase. They are estimated to be common operations during YSO development, also, as the ontology is iteratively re-structured.

Moving a property from a class to a referenced class or to a new class are ranked next. They are generic ontology modification operations that are likely to prove valuable during the early construction phases of YSO, when the ontology structure is changing rapidly. Declaring two classes disjoint is a quite rare operation in practical ontology work; the need for it did not arise during MAO construction, ever. However, it contains a risk of data loss, and should be monitored. Defining a property transitive or symmetric occurs more often but has more limited consequences, and is therefore considered less important.

When new classes, properties and instances are created the new URI may introduce a naming conflict. Also, if the same URI did not appear twice in the ontology library, even in two disjoint ontologies, the library would be more consistent, which is one of Gruber's ontology design principles [4]. Checking the including ontologies, or even the entire ontology library for similar names when a new URI is created is a very handy practical aid to ontology development. However, because the risks involved with duplicate URIs are very small, it is not considered a priority issue.

The next group of changes includes adding a property to a class, adding a superclass or a superproperty, and moving a property up the class hierarchy. Under some particular cases they can introduce problems in the including ontologies, but this happens very rarely. The operations themselves are performed rather often, and gain thus a higher priority than re-classifying a class as an instance. The need for re-classifying never occurred during MAO development and both re-classifying operations are considered rather rarely used. Since re-classifying a class as an instance may cause data loss, it is prioritized higher than re-classifying an instance as a class, which may only cause some property values become invalid.

The lowest priority was given to the completely safe operations of widening a restriction for a property, changing the superproperty of a property to a property lower in the property hierarchy, and changing the superclass of a class to a class lower in the class hierarchy. When inspected from the point of view of the class or property that is being modified, these operations do not pose any problems to the ontologies. However, one should bear in mind that the last two operations can be viewed also from the point of view of the old subclass or subproperty. From that point of view, the operations equal the opposite operations of moving a superclass or a superproperty to a class or a property higher in the hierarchy. These are both high priority operations and potentially cause problems in the changed ontology as well as in the including ontologies.

3.6 Summary

In this Chapter the different ontology change operations were presented and analysed, and experiences from practical ontology work were introduced. The main problems of current working methods were briefly presented to evaluate the future needs of the system users. The experiences, problems and analyses results were put in the context of the thesis use case; the results were prioritized for YSO development and maintenance. The results are summarized in Tables 3.1, 3.2 and 3.3, which group the results into three groups. Table A.1 in the appendices groups the results in one table.

Table 3.1 presents the change operations that cause most problems in distributed development of dependent ontologies. Any ontology library system aiming to support distributed ontology development is strongly adviced provide means to detect and handle these operations. Table 3.2 illustrates a set of ontology change operations that should be considered in a practically oriented ontology development environment in addition to the operations presented in Table 3.1. Their priorization is highly influenced by the YSO case context. Revising the priorities may increase performance or usability when a system is used in other type of ontology construction.

Table 3.3 lists the changes that are relatively safe. Ontology library systems aiming to improve the user experience when the user is using the system may implement support for some of these changes. For example, functionality to find similar names from the ontologies of the ontology library when the user creates a new concept would be handy indeed. However, the problems caused by the operations are very limited, or

Table 3.1: *Most critical edit operations.*

Priority	*Operation*	*Comment on effect*
High	Deleting a class, a property, or an instance	Data loss, breaks referencing ontologies
High	Changing a URI	Equals to delete in referencing ontologies
High	Removing a superclass or a superproperty	Identity of subelements changes. Breaks referencing ontologies
High	Moving a class in the class hierarchy	May be safe, or may equal removing of several superclasses
High	Changing the superclass or the superproperty to a class or a property higher in the hierarchy	Equals removing some superclasses or superproperties
Medium-High	Removing a property from a class	Loss of data. Breaks referencing ontologies
Medium-High	Moving a property down the class hierarchy	Equals removing a property for some classes

the operations are used so rarely that they are not very interesting from keeping the referencing ontologies consistent point of view.

Table 3.2: *Changes that should be supported by an ontology engineering architecture.*

Priority	*Operation*	*Comment on effect*
Medium	Narrowing a restriction for a slot	Data loss. Introduces hard to find conflicts into referencing ontologies
Medium	Merging classes	Breaks referencing ontologies. Common operation in YSO development and maintenance
Medium	Splitting a class	Breaks referencing ontologies. Common operation in YSO development and maintenance
Medium-Low	Moving a property to a referenced class	Breaks referencing ontologies. Useful operation in ontology restructuring
Medium-Low	Moving a property to a new class	Breaks referencing ontologies. Useful operation in ontology restructuring
Low	Declaring two classes disjoint	Rare operation. Breaks the semantics of referencing ontologies
Low	Defining a property transitive or symmetric	More common than declaring classes disjoint, but less dangerous. Breaks the semantics of referencing ontologies

Table 3.3: *Safe changes.*

Priority	*Operation*	*Comment on effect*
Very Low	Adding a new URI: a class, a property or an instance	Common operation. Problems occur very rarely, limited consequences
Very Low	Adding a property to a class	Very rarely conflicts if the same property exists in a subclass and property restrictions conflict
Very Low	Adding a superclass or a superproperty	Safe, may introduce same problems as adding a property. Very safe operation
Very Low	Moving a property up the class hierarchy	May cause same problems as adding a property if the class has other subclasses which already have the property
Very Low	Re-classifying a class as an instance	Very rare operation. Loses data. Breaks referencing ontologies
Very Low	Re-classifying an instance as a class	Very rare operation. Breaks referencing ontologies
Safe	Changing the superclass or the superproperty to one higher in the hierarchy	No problems for the class or property in question
Safe	Widening a restriction for a property	Completely safe change

Chapter 4

Existing Solutions

4.1 Introduction

The problems caused by ontology reuse by inclusion have been acknowledged, and different solutions exist to tackle the problem. In this Chapter I will present some existing systems and methods, and briefly explain their focus in regard with the conflicts raising from changing an included ontology.

4.2 KAON Engineering Server

KAON ontology development environment [6] has been developed in the University of Karlsruhe in Germany. In KAON architecture (extended with Engineering Server and JBoss) the ontologies can reside on different hosts and include other ontologies. Included ontologies are duplicated, and changes are allowed only to the original ontology. The changes are then propagated to the duplicate ontologies, and the user is asked if he would like to accept the proposed changes or keep the older version of the included ontology. This functionality is explained by Maedche, Stojanovic and colleagues in [6], [18] and [5].

KAON focuses on fluent propagation of changes between the remote ontology copies. It provides tools to help the user to understand the impacts of the proposed changes and lets the user decide if the changes are to be executed. However, it does not provide means to help the user, who made the original change, to understand what problems his change will cause for others. This is because KAON targets a more distributed, larger ontology library system than what is needed for YSO development.

4.3 Protégé

Protégé 2000 [19] is a widely used ontology editor developed at the Stanford University. It offers an intuitive graphical user interface, and its support for ontology inclusion is one of the best in the field. Protégé can store ontologies in files and in relational databases, but ontology inclusion is supported in file based ontologies, only. Information about each ontology is stored in a project file, and class and instance data are separated to different files. The project file defines the location of the class and instance files, and which ontologies are included. Ontology inclusion is done by loading the projects in order. Protégé maintains ontological consistency by preventing users to modify the included ontologies. All changes must therefore be made to the original ontology. Changes appear in the including ontologies as soon as the including projects are reloaded. Protégé does not provide any means to see what effects the change has in the including ontologies. Developers can thus easily break the including ontologies as was explained in section 2.4.

4.4 OilEd

OilEd is an editor for ontology languages OIL [20] and DAML+OIL [21]. It is developed at the University of Manchester and its functionality is further explained by Bechhofer et al. in [22]. OilEd does not support ontology inclusion in the sense that was defined in section 2.3. For OilEd, ontology inclusion means merging copies of the included definitions into the model to be edited. After the merge operation, the information on the origin of a definition is lost. As was discussed earlier, this approach is safe from dependency problems, because there are no dependencies between ontologies. At the same time it introduces severe consistency problems as the ontologies need to evolve. Because there is no information on where each concept definition has come from, each ontology needs to be modified independently of other ontologies. One modification must be done several times to different ontologies. There are no guarantees that the changes would be same, much less that the different ontologies reusing the same ontology would remain inter-operable. This causes serious problems in application development.

4.5 Ontolingua

Ontolingua [11] is a collaborative ontology development tool that supports even cyclical ontology inclusion. As was mentioned in section 2.3, cyclical ontology dependency chains can introduce very complex problems when the effects of the changes should be checked or propagated to the including ontologies. Alas, despite of its powerful inclusion mechanism Ontolingua does not provide support for changing the original ontology and propagating the changes to the including ontologies.

4.6 WebODE

WebODE [7] is a web based ontology editor initially developed for editing OIL ontologies. It is based on METHONTOLOGY, a methodology for ontology development first introduced in [23] and then refined in [24]. WebODE uses a database to store ontologies. It utilises a concept of user groups to establish access control to ontologies, and has synchronization mechanisms to prevent errors from concurrent access. In [7] the authors state that it supports collaborative ontology editing at the knowledge level, but do not explain more precisely what this means. WebODE introduces an interesting ontology reuse feature that it calls the instance sets. As WebODE does the separation of class and instance data, it allows the user to easily create different sets of instance data to be used individually with one class model. Each instance set can represent a different use case of the ontology. For example, given an enterprise storehouse ontology, different instance sets can represent the stores of different companies.

Ontology inclusion support in WebODE is far less advanced than the instance set approach would suggest. The system allows the developer to import only classes from other ontologies, and demands that each class is imported individually. Properties or instances cannot be imported. Even though WebODE implements support for some interesting features such as the instances sets, its inclusion mechanism is somewhat limited. According to my understanding, the system does not provide means to propagate ontology changes to dependent ontologies in a well-controlled manner.

4.7 OntoEdit

OntoEdit is an ontology engineering environment, which concentrates on tasks involved with ontology requirements specification, refinement and evaluation. It combines methodology-based ontology development with collaboration and inferencing, as explained in [25]. Due to its focus on the early phases of the ontology life cycle, it does not offer any support for ontology evolution. However, it has been acknowledged for example in [8] and in [6] that ontology evolution should be considered a very important aspect in the ontology life cycle.

Chapter 5

Proposed Solution

5.1 ONKI Overview

In the previous chapters the role of ontologies in Semantic Web and their modular reuse was discussed. It was found that the modular reuse may introduce consistency problems when the reused ontology is modified. Different change operations were examined and their results were analysed with regard to ontological consistency in dependent ontologies. We reviewed some lessons learned from practice, and the change operations were prioritized based on their occurence frequency and on the seriousness of their effects. We briefly looked the different stands the existing systems take on these issues. In this Chapter an architecture is presented for a system that could be used in distributed ontology development and maintenance. The system's primary goal is to provide a framework for YSO development. While focusing on this particular use case, the system architecture is also aimed at supporting upper level ontology creation and maintenance work and providing a work bench for researchers to try out different ontology engineering methodologies. The system incorporates ontology library and distributed ontology development fuctionalities and integrates a set of supporting acitivities into the same tool.

I propose a client-server based ontology library architecture (ONKI, ONtologiaKIrjasto) for distributed development of an ontology library [8]. The architecture is designed to enforce disciplined ontology development and release processes with user access control. The need for managed ontology development is recognized by Gruber in [4] and Farquhar in [11]. ONKI contains a centralized storage with version control for ontologies under development. The server incorporates an ontology dependency model, which keeps track of the dependencies between different ontologies. During ontology development it is used to guide developers if they attemp to do changes that would break other ontologies than the one they are editing. End users and application developers access the ontologies via a public ontology library, where developers can publish their ontologies as they reach a stable and mature enough state. If a developer decides to commit a change that breaks other ontologies, a notification is automatically sent to the developers of the broken ontologies.

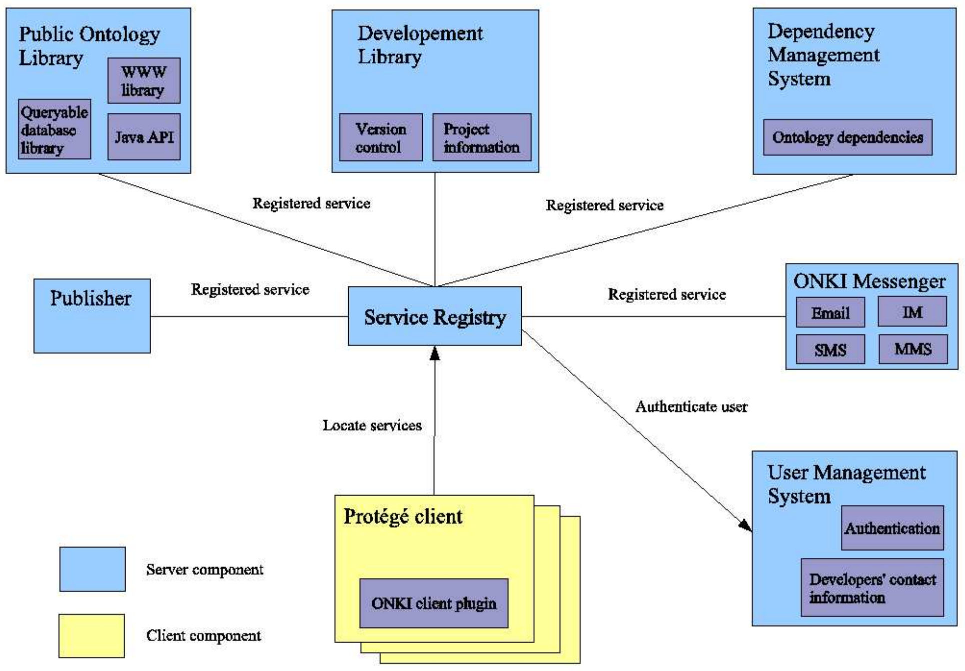


Figure 5.1: Conceptual ONKI architecture.

The ONKI architecture incorporates an ontology editor, which checks the safety of operations from the server before executing them. Ontology editing is performed locally at the developer's workstation, and the results are committed back to the central repository. When a user wants to develop an ontology A, he first creates a new ontology. He then checks out from the version control system any ontologies he wishes to include (B and C). The user starts developing ontology A locally, and regularly commits the changes to the server. At commit time the system updates the ontology dependency model to contain the relations that point from ontology A to ontologies B and C. If the developers of B or C now try to make a change that would affect ontology A, their clients warn them about the consequences of the change and give them a change to proceed with the change or to cancel it. This ensures that if a change in the upper ontology is needed, its developers know what it will mean for those ontologies that reuse the upper ontology. The developers can therefore try different alternatives to come up with a change that is backwards compatible or affects as few including ontologies as possible. No-one can anymore accidentally break another ontology by changing an included ontology without warning.

5.2 ONKI Server

A client-server architecture is seen important in collaborative editing and versioning, as among others Ding and Fensel acknowledge in [8]. The key purpose of the client-server architecture is to offer a common access point where the edits performed by different users can be checked in order to keep the edited ontologies in a consistent state. ONKI Server provides the ontology developers a common access point to ontologies in the ontology library. It offers a persistent storage of ontologies with version

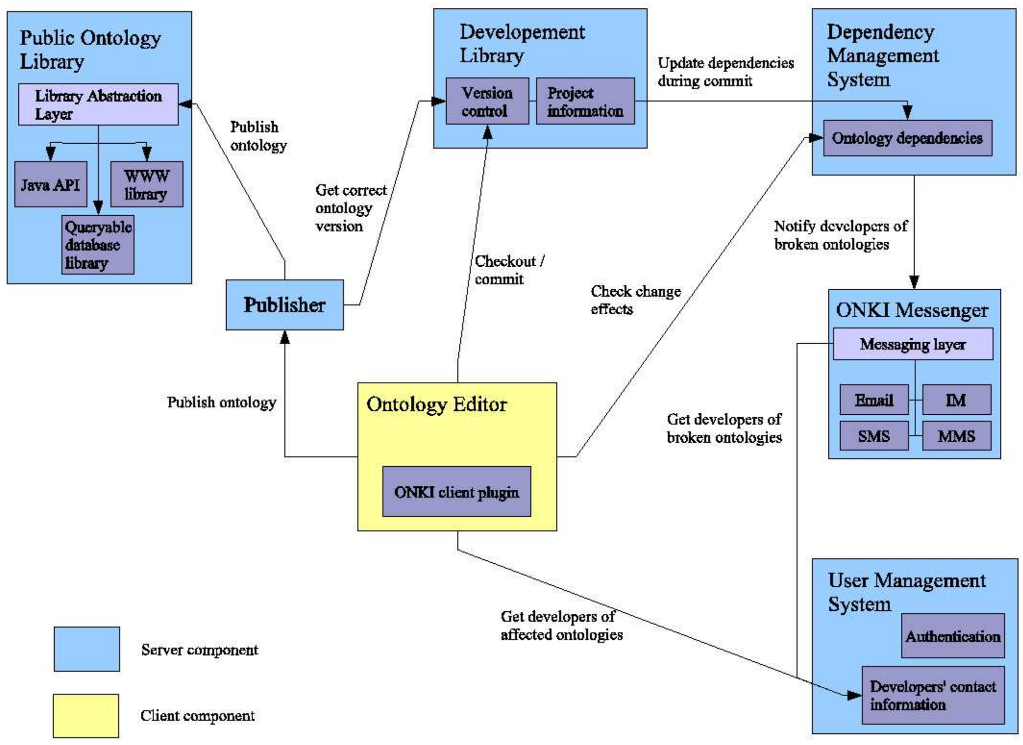


Figure 5.2: Detailed ONKI Server architecture.

control, user management and authentication, real-time querying of dependencies between ontologies and a public ontology library to publish finished ontologies.

The core of the server is the Service Registry and the authentication module of the User Management System. Other components connect to the system by registering themselves as services in the Service Registry. All services can then be accessed via the Service Registry. The process is illustrated in Figure 5.3. Figure 5.2 omits the processes of authentication and locating a service and shows the server architecture in more detail. It displays the functional connections that the Service Registry creates on request between different system components.

The server components form a system that enforces a disciplined process for ontology development. The developers develop ontologies in the Development Library, which integrates version control and modular design of ontologies. Most other solutions lack the versioning support, as Ding and Fensel point out in [8]. The Dependency Management System and ONKI Messenger help to coordinate collaborative development of inter-dependent ontologies so that developers do not introduce errors in including ontologies while modifying their own ontology. The ONKI Client checks all edit operations from the Dependency Management System, and warns the user when he attemps a change that would break other ontologies. If the user decides to proceed with the change and commits it to the Development Library, ONKI Messenger automatically sends notifications to the developers of those ontologies that have become broken by the change. Those developers then have the change to bring their ontologies up to date with the new version of the reused ontology, or to decide to remain compatible with the previously released version. As ontologies come ready for public access, they are published in the Public Ontology Library. The Publisher enforces a consistent ontology publishing policy with official and developer releases of ontologies. The functionality of each of the components is presented in more detail in the next sections.

5.2.1 Service Registry

The Service Registry is the heart of the ONKI architecture. As the components of the ONKI server could be distributed across the network, the Service Registry provides the system information on where each component is located. It thus provides an access point for services that the server components implement. All server components join the ONKI server architecture by registering themselves as services in the Service Registry. They can then be accessed by requesting a connection to a specific service from the Service Registry.

The Service Registry enforces communication security by authorizing each service request from the User Management System. As a user edits the ontology, his ONKI client requests services from the Service Registry. If the users has the right to use those services, the Service Registry establishes a secure communcation channel between the client and the service component. The channel is then forwarded to the client to be used for communication.

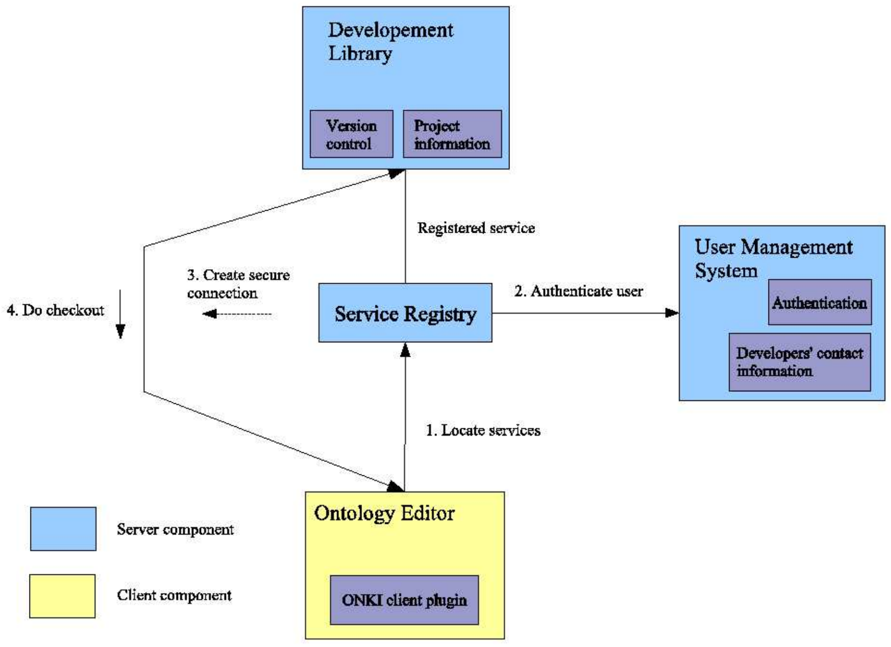


Figure 5.3: Locating a service.

5.2.2 User Management System

The User Management System offers authentication services to the ONKI system. As a user requests the Service Registry for a service, the registry queries the User Management System to check if that user has the right to access the specified service. If the access is granted, a secure communication channel is created between the client and the service. Otherwise, the client's request fails.

Another function of the User Management System is to contain an ontology-based module that stores the information on developers registered to develop the ontologies of the Development Library. The user ontology has a username, real name, organization and contact information for each user. The contact information can include any number of email addresses, phone or fax numbers or postal addresses. The information in collected as a user registers himsef as a developer of certain ontologies. The registration is handled by the User Management System. Having registered, the users can then receive automatic notifications on any changes that affect the ontologies they develop. The automatic notification service is provided by the ONKI Messenger and is described later in more detail.

5.2.3 Development Library

The Development Library provides a centralized storage for ontologies under development. Its main purpose is to integrate version control to the ontology development process, which is a prequisite for serious ontology development. Consistent ontology versioning and version control is widely considered as a core requirement for an ontology library system (see [26], [8], [12]).

The version control system in ONKI is based on CVS[1] [27], a popular and freely available open source version control software. The system keeps track of the versions of class and instance models, as well as of a file containing separate project information. A project binds a class and an instance model together to form an ontology; users may create several separate projects based on the same class or instance data to create differenct ontologies. Klein et al. [26] suggest separating the identity of ontologies from the indentity of the files in which the data resides. An ontology library modularization [8] can be based on this separation. The purpose of the modularization and the identity separation is to promote ontology reuse. The design allows ONKI system to support the use of class sets as well as instance sets as proposed in [7]. Analogously to an instance set, a class set is a class and property model that can be used to structurize certain data represented by an instance set. Several class sets can be created for the same instance set to view the data from different perspectives, or to use the data for different purposes.

5.2.4 Dependency Management System

The Dependency Management System is the core value-adding component of the ONKI architecture. Its purpose is to support collaborative editing and ontology reuse, which are one of the most important functions of an ontology library system [8, 11]. All dependencies between the ontologies in the Development Library are stored in the Dependency Management System. The underlying data model is an ontology that

[1]http://www.cvshome.org

stores information on which elements in which ontologies reference which other elements in which other ontologies. It also contains weight factors for the referecing elements. The weight factors are class tree weight and instance tree weight. The class and instance tree weights are used to estimate the magnitude of the effects to the referencing ontology if a particular element would be broken. This way the developer can see an estimation of the severity of the problems the change would cause to other ontologies, and can use this value to compare different change operations. The class tree weight represents the number of classes in the ontology that will become broken if the element is broken. Correspondingly, the instance tree weight represents the number of instances that refer to the element and would hence be broken with the element. Properties can be included in those Figures by thinking of them as classes and instances. A definition for a property is considered a class and hence increases the class tree weight. The attachment of a property to a class is considered an instance and this time the instance tree weight is increased.

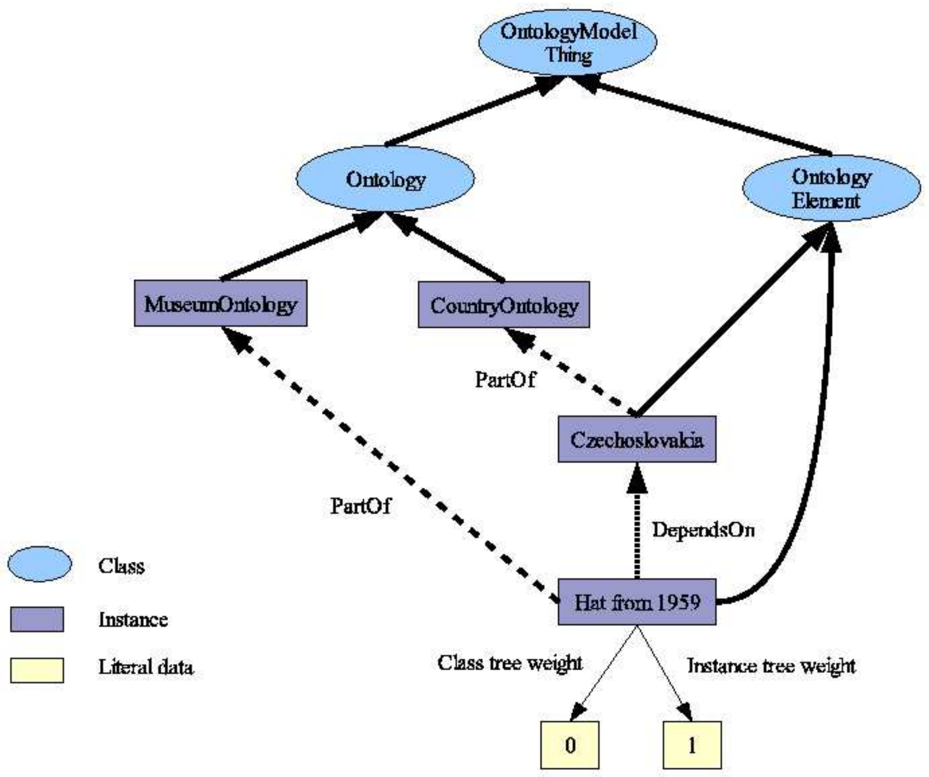


Figure 5.4: An example of the Dependency Ontology.

Figure 5.4 illustrates the Dependency Ontology. It shows the relations that the system would need to solve the case of the Czechoslovakian hat in the museum from section 2.4. The class hierarchy beginning from the element "Hat from 1959" contains one instance and no classes; its class tree weight in the Dependency Ontology is zero, and its instance tree weight is one. When the ontology developer is about to make a change to an ontology, the client software determines which elements in that ontology would be affected by the change. The client then queries the Dependency Management System to see what other elements in other ontologies would be affected by the change. From the information represented in Figure 5.4 the Dependency Management System would reason that if the instance "Czechoslovakia" would be removed from the country ontology, the element "Hat from 1959" in the museum ontology would become broken. The problem would affect one instance in the museum ontology, that is, the element "Hat from 1959". The developers of country and museum ontologies would know that no other classes or instances would be broken.

Of course, the same results could be obtained if the Development Library supported real-time querying of the ontologies it stores. However, individual ontologies can become quite large, counting several thousands or even millions of elements. Querying several ontologies of such size would soon tear down any ontology library implementation. That is why the ontology dependency checking functionality and the Dependency Ontology have been separated into a dedicated server component. The Dependency Management System needs to be able to rapidly answer questions of type "What ontologies (and how) depend on a given ontology?". It would often be too expensive to query the entire contents of the Development Library.

5.2.5 ONKI Messenger

When an ontology developer has reused another ontology by including it into the one he is developing, he needs to know if someone changes the included ontology so that the changes would affect his own ontology. This is where ONKI Messenger service steps in. When an ontology is committed back to the Development Library, its dependencies are updated to the Dependency Management System. If the existing references to the ontology are broken due to the commit, ONKI Messenger uses the ontology's change log to automatically generate notifications [11] to the registered developers of the ontologies that have become broken. The developer that initiated the commit may add clarifications, explications or instructions to the messages to help other developers understand the change and how the committed ontology should be reused by the broken ontologies.

ONKI Messenger abstracts the message transmission channel so that messages can be sent to whatever device necessary. The message can for example be an email, an instant message, an MMS message or an SMS message. This allows ONKI integration to support mission critical ontology development. For example, an ontology could be used to filter the news feed of an international new agency. The system filtering the news could also perform content analyses on the news articles by using various data mining techniques [28]. The results of the data mining could then be automatically converted into ontology edit operations that expand and restructure the ontology. For example, the system could automatically add new classes and relations between different ontology elements, or split and merge classes. As the core ontology would propably be reused in various case specific ontologies to produce more customized services such as a sports news service, a domestic political news service and an economic news service, the system would need to ensure that these value added services would keep on running as the core ontology is enriched.

5.2.6 Public Ontology Library

As ontologies have been developed, a need emerges to publish them for the wide audience. ONKI Server supports this task with the Public Ontology Library and the Publisher modules. The Public Ontology Library abstracts the concept of a publicly accessible ontology library. For developers, the library has a service to add a new ontology or a new version of an ontology to the library and set a status for the ontology version. Ontology development can continue in the Development Library while application developers can use the published, stable releases to develop applications and services based on the ontologies. Pre-release ontology versions can be published to support application development and they are clearly distinguished from the official ontology versions.

For the library end users, the Public Ontology Library architecture features a extensible set of library implementations into which the new ontology is added. The library can be accessed through several interfaces. A WWW-site where end users and application developers can download entire ontologies is probably one of the principal access methods. This is the standard way most ontology libraries work. WWW-based access is easy to implement and readily available to everyone with today's telecommunication infrastructure. No wonder that Ding and Fensel raise it as one important feature of an ontology library in [8].

Another way to access the Public Ontology Library would be via a queryable database with a WWW-based user interface. Advanced users and application developers could query for a certain concept in a certain ontology. An example of this kind of access is Verkkosanasto VESA, http://vesa.lib.helsinki.fi/, which provides a query interface to the thesauri Yleinen Suomalainen Asiasanasto YSA (General Finnish Thesaurus), Allmän tesaurus på svenska ALLÄRS (Swedish version of YSA), and Musiikin asiasanasto MUSA/CILLA (Thesaurus of Music).

The two approaches presented above both aim to satisfy the ontological needs of a human user. But the library user profile should not be limited to mere humans. The software that is being developed these days incorporates more and more often automatic upgrading capabilities. As the Internet has become widespread and the end user network speeds are growing due to increased number of broad band subscriptions, the mere nature of software is facing a change. It is becoming more like a service fulfilling the end user's need than a stand-alone product. As ontologies are used to add intelligence to the software applications, an ontology library must be capable of providing programmatic access to the ontologies in order to become a truly useful library platform. Different software components can access the Public Ontology Library through a Java API to automatically download ontologies as they need them to interact with other software components or to perform automatical updating of the ontology when a new version is published.

5.2.7 Publisher

To bridge the gap between the Development Library and the Public Ontology Library one needs a dedicated ontology publishing tool. The Publisher contains functionality to publishes a specified version of an ontology from the Development Library to the Public Ontology Library. It enforces a consistent ontology publishing discipline with official and developer releases of ontologies. The release thinking is a step towards thinking ontology management as product management, a shift needed to create an integrated environment that supports efficient ontology use. Official releases are releases that have been tested with the official releases of the ontologies that they include, and are guaranteed to be stable. They form the basis for ontology-based application development. Software developers link their systems and knowledge bases with officially released ontologies. These ideas have been suggested by Farquhar et al. [11] and Gruber [4], but the stability guarantee given by a commitment to an official release is needed for the promises of ontology use to become reality.

Developer releases can come out more often that official releases. An ontology developer may wish to give the application developers a preview of his ontology, even though it is not yet complete. The developer release does not guarantee anything; any concept or relation might be changed or removed any time. The purpose of these releases is to speed up application development. Software developers can start creating software of the new version of the ontology already before the official release is ready.

5.3 ONKI Client

The client system consists of an ontology editor, of a knowledge base storing the ontology to be edited, and of a dependency client. There are quite a few ontology editors available on the market, and any one of them could be selected. The only restriction is that the editor must provide the change identification layer of the dependency client some way to intercept edit operations before they are written to the ontology knowledge base. Figure 5.5 illustrates the client architecture and information flow when the user makes an edit operation at the editor. Let us next look what happens in the dependency checking mechanism lying between the editor and the knowledge base.

5.3.1 Change Identification Layer

At the surface of the dependency client lies the Change Identification Layer. When the ontology developer edits an ontology, he uses an ontology editor to perform a series of edit operations on the knowledge base containing the ontology. The edit operations must be caught before they enter the knowledge base or the system is not capable of preventing the user from breaking dependent ontologies. The Change Identification Layer serves as an interface to the ONKI client. Different implementations can be created to attach the ONKI system to different 3rd party ontology editors. The duty of the Change Identification Layer is to recognize any edit or modification attempted by the editor, intercept it, and forward it up the ONKI client component chain. As an edit is detected, the Change Identification Layer first identifies and classifies it according to the classification presented in Chapter 3. The edit is decomposed into individual change operations and then forwarded to the filtering layer that validates its safety.

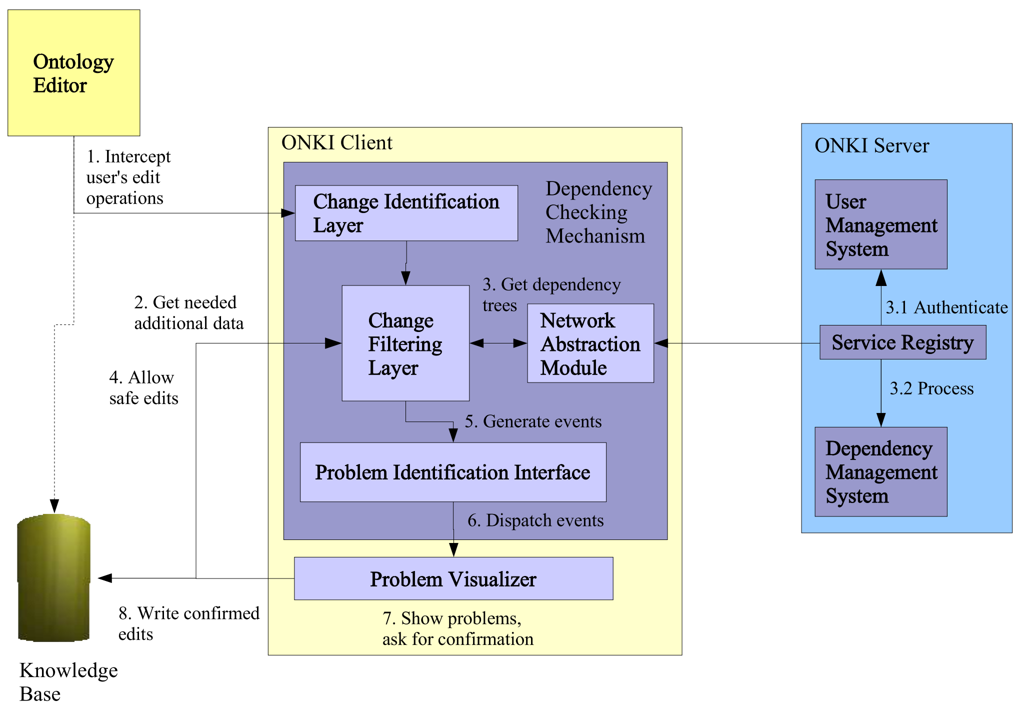


Figure 5.5: ONKI Client components.

5.3.2 Change Filtering Layer

Change Filtering Layer receives individual edits from the Change Identification Layer. It calculates the elements that are affected by the edit as explained in Chapter 3. As we remember from the change analyses, the edits are formed from a sequence of change operations that are used to perform the edit. The system can use these change operations to compute the delta between the knowledge base state before and after the edit. The result is exactly the elements that are affected by the edit. Having gathered all affected elements, the layer contacts the Dependency Management System on the server and asks which elements have dependent elements in other ontologies. The server replies with a list of dependency trees. At the root of the dependency tree is the affected element from the local ontology. Its child nodes identify the elements that depend on the root element and are defined in other ontologies, and the ontology in which those elements are defined.

If the returned set of dependency trees is not empty, the edit will break other ontologies. In this case the layer creates a dependency event that identifies the original edit operation as the cause of the problem, and attaches the dependency trees to the event to specify the consequences of the operation. The created dependency event is forwarded to the Problem Identification Interface. In the case when the server reply is empty, no other ontologies are affected by the change. Dependency processing then ends silently.

5.3.3 Problem Identification Interface

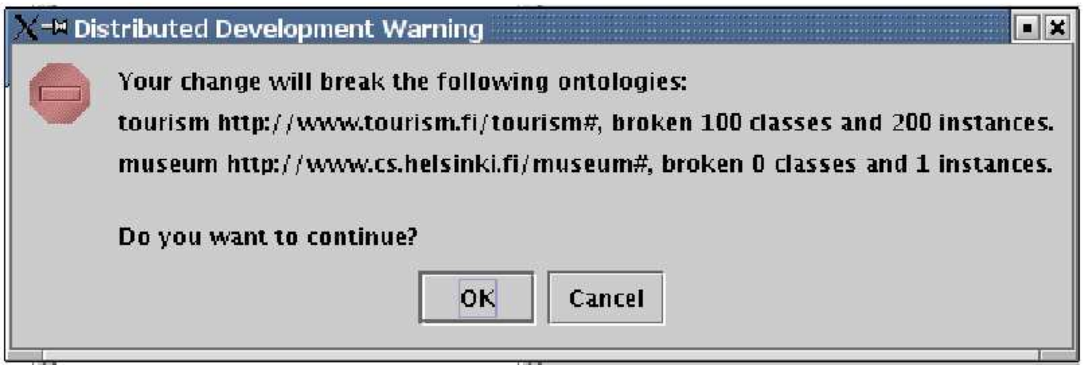
The Problem Identification Interface is where external software can detect dependency problems. The interface has an event dispatching mechanism that distributes dependency events on the identified dependency problems (for example, "removing of class http://www.cs.helsinki.fi/testProject#foo breaks its subclass http://www.test.fi/testProject2#bar in ontology http://www.cs.helsinki.fi/testOntology"). Problem handling components or user interface software can listen to these events and try to solve the problems automatically, or present the consequences to the user and ask for confirmation.

A second event dispatching mechanism is included to distribute events about the underlying dependency model itself (such as "model processing","model saved" etc.). This is provided to let user interface software get information on how dependency checking is progressing, and use this to customize the user interface accordingly. This is important because the processing involves calls over a network. The network may be slow, and the user should be informed that something really is happening and the software has not crashed.

3rd party software components can implement one, the other, or both of the event listener interfaces, and register themselves to listen to the events. Then they receive events when the user attemps to perform a change that would cause problems to other ontologies. They also recieve information on the state and execution progress of the dependency plugin.

5.3.4 Problem Visualizer

Problem Visualizer is a simple component built on top of the problem identification layer. It listens to dependency events, shows them in appropriate detail to the user and asks the user to confirm or cancel the change that causes problems. If the user proceeds, the change is made to the underlying knowledge base. Otherwise, entire change is cancelled. Figure 5.6 shows a draft of the Problem Visualizer output. The example shows a rather simplistic visualization approach, and from the Figure one can readily see why the approach should be kept simple.


Distributed Development Warning

Your change will break the following ontologies:
tourism http://www.tourism.fi/tourism#, broken 100 classes and 200 instances.
museum http://www.cs.helsinki.fi/museum#, broken 0 classes and 1 instances.

Do you want to continue?

Figure 5.6: Warning displayed by the Problem Visualizer.

The amount of affected items in the ontologies can be quite large in some cases as the tourism ontology case shows. The visualizer needs hence to summarize the information it receives from the Dependency Management System. A new button could be added to the output dialog for retrieving further details. The button could extend the information presented to the user by showing all problem details such as which elements in other ontologies are affected, how they are affected, who of the developers are developing the affected ontologies, and how many users the affected ontologies have. The needed information could be retrieved from the Dependency Management System, the User Management System and from the Public Ontology Library.

Chapter 6

Implementation

6.1 Introduction

To demonstrate the architecture described in the previous section, I have built a prototype implementation of it. I chose to use Protégé 2.0[1] [19] as the ontology editor. Protégé is one of the most popular ontology editors in the world. It is an open source product written in Java[2] [29] and has an intuitive graphical user interface, which makes it easy to learn. Protégé originates from the Stanford University, where it is actively developed further. It has a plugin architecture that allows extending the original product in various ways. The ONKI Client was implemented as a plugin to Protégé.

The ONKI Server prototype features the Service Registry, the Dependency Management System, the Development Library, the Publisher and the Public Ontology Library. The implementation was written mostly in Java. The Publisher was prototyped as a Unix Shell Script. The Development Library is based on CVS [27] and the knowledge base of the Dependency Management System on Jena 2.0 [30]. CVS is the Concurrent Versions System, the dominant open-source network-transparent version control system.[3] It is freely available under the GNU Public License. Jena is a Java framework for building Semantic Web applications. It provides a programmatic environment for RDF, RDFS and OWL, including a rule-based inference engine. Jena is open source and grown out of work with the HP Labs Semantic Web Programme.[4]

6.2 Monitored Change Operations

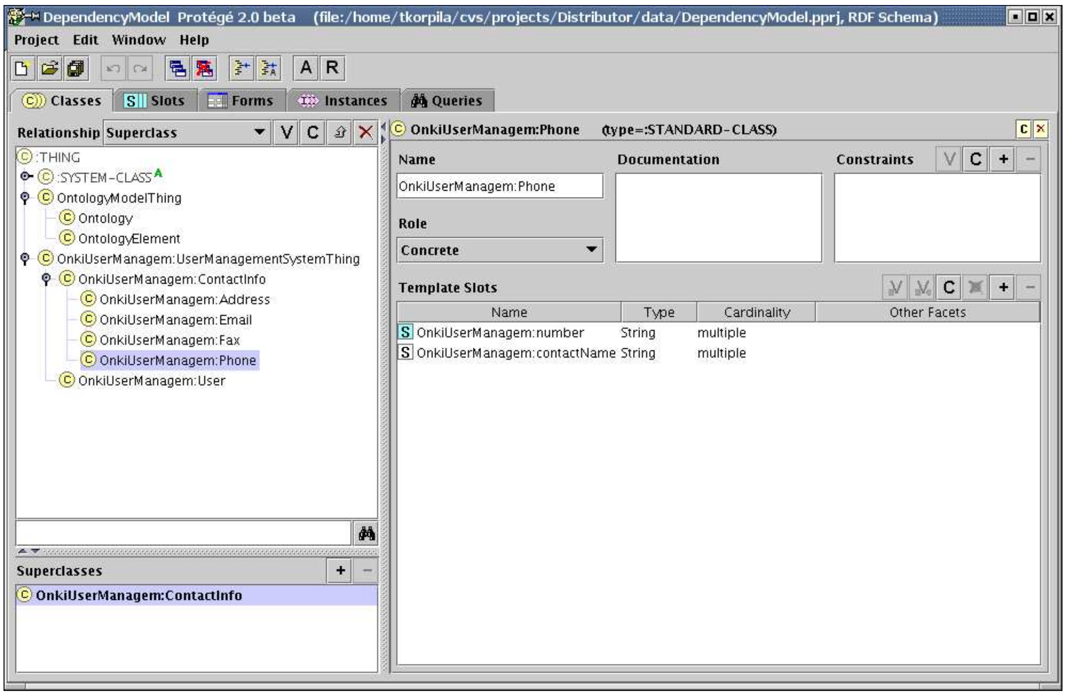
Before deciding which change operations the ONKI prototype system should monitor, one must first consider a couple of issues. ONKI will be used to develop YSO from material that is based on the YSA thesaurus. On one hand, YSO is an upper level

[1]http://protege.stanford.edu

[2]http://java.sun.com

[3]http://www.cvshome.org

[4]http://jena.sourceforge.net



OnkiUserManagem:Phone (type=:STANDARD-CLASS)

Class hierarchy:
- :THING
  - :SYSTEM-CLASS
  - OntologyModelThing
    - Ontology
    - OntologyElement
  - OnkiUserManagem:UserManagementSystemThing
    - OnkiUserManagem:ContactInfo
      - OnkiUserManagem:Address
      - OnkiUserManagem:Email
      - OnkiUserManagem:Fax
      - OnkiUserManagem:Phone
    - OnkiUserManagem:User

Superclasses: OnkiUserManagem:ContactInfo

Name: OnkiUserManagem:Phone
Role: Concrete

Name	Type	Cardinality	Other Facets
OnkiUserManagem:number	String	multiple	
OnkiUserManagem:contactName	String	multiple	

Figure 6.1: Protégé 2.0 ontology editor.

ontology, and thus is not likely to contain many instances compared to the number of classes. On the other hand, the Protégé editor offers mainly views that are structured around the class hierarchy and represent a class as the most central ontology building block. Also, the experiences obtained from the construction of MAO indicate that focusing on classes would benefit the ontology engineer the most.

The change operations that the system would monitor were selected based on the priorization done in section 3.5. The most critical change operations listed in Table 3.1 were all implemented, except the support for monitoring for moving a property down the class hierarchy, which is not supported by Protégé as a single operation. Support for monitoring instances for loss of the direct type was added, too. This happens as a side effect of a superclass removal, and can be done as an individual operation in the Protégé editor, too.

From the second set of changes (the "should-have" features) listed in Table 3.2 Protégé supports directly only narrowing a property restriction and defining a property symmetric. From the experiences of MAO development I decided to implement a functionality to check for a change in the range of a property. This can cause serious data loss and can easily introduce into the including ontologies conflicts that are difficult to find. Other types of property restrictions were not considered in the prototype. Defining a property symmetric was estimated to occur so rarely that it would not be a priority issue in the prototype.

The features presented in Table 3.3 were not implemented. On the first hand Protégé does not support the re-classifying changes as one operation. On the other hand the rest of the changes pose a minimal risk of conflicts to the ontology dependency structure, and were therfore left out from the prototype. Table 6.1 summarizes the monitored operations.

Table 6.1: *Monitored change operations.*

Operation	*Comment on effect*
Deleting a thing (a class, a property or an instance)	Data loss, broken references
Changing a URI	Data loss, broken references
Removing a property from a class	Data loss, broken references
Removing a direct superclass from a class	Data loss, change of semantics in an including model
Moving a class in the class hierarchy	May equal removal of several superclasses
Removing instance's direct type	Data loss, conflicts in an including model
Changing the range of a property	Data loss, conflicts in an including model

6.3 Stored Dependency Relations

To be able to use automatic inferencing to decide if the above mentioned change operations are harmful or not, the Dependency Management System needs to store information on the dependency relations between the dependent ontologies. When populating the Dependency Ontology, the system iterates the classes, properties and instances of the ontology L that is open in the editor.

For a class C defined in an ontology L, the system checks if it has a superclass $SuperC$ that is defined in an included ontology I. If $SuperC$ exists, the Dependency Management System marks that C in ontology L depends on $SuperC$ in ontology I. Similarly, if C has a property P that is defined in the included ontology I, dependence from C to P is stored.

In addition to the class hierarchy dependencies, the property hierarchy may also be reused in an including ontology. Therefore the dependencies in the property hierarchy must be stored. If a property P has a superproperty $SuperP$ that is defined in an included ontology, a dependency from P to $SuperP$ is stored. This is sufficient to deal with the property hierarchy dependencies, but other types of dependencies can be created between different classes and properties. A relation between two classes is created by creating a property that has one class in its domain and another one in its range. The range of a property is a set that defines what type of values the property may have. The range may include classes and instances; they create a dependency from the property P to the range class or instance R, which must be stored. The domain of the property was already handled while inspecting the class C.

Classes may have class level properties. Class level properties are properties that are attached to a metaclass M of the class C. C is thus an instance of M. Originally, the ontology hierarchy is an N layer model. A class M can be instantiated, an instance C is created and its type is M. The instance C can be further instantiated by creating an instance I that has C as its type. Now I is an instance, C is a class (because it has an instance, I) and an instance (of M), and M is a metaclass because it is a class, which has an instance that is also a class (C). Properties attached to M can have values in

Table 6.2: *Dependency relations stored in the Dependency Management System.*

Element	*Relation*
Class	Is a subclass of an included class
Class	Has an included property
Class	The value of some class level property is a reference to an included class or instance (Class is an instance of a metaclass. Class level properties are the properties of the metaclass)
Property	Is a subproperty of an included property
Property	Has an included class or instance in the range
Instance	Is an instance of an included class
Instance	The value of some property is a reference to an included class or instance

C, and properties attached to C can have values at I. Properties of M are class level properties; they have values that are attached to a class (C). Roughly speaking, a class level property is a property that can be assigned a value at some class. If the value of such a property in class C is a reference to an included class or instance V, the dependency from C to V is stored to the Dependency Management System.

For an instance I, the system checks if it is an instance of an included class C, in which case the dependency from I to C is recorded. I can have values for its properties. If the value of a property is a reference to a class or an instance V and V is defined in an included ontology, a dependency from I to V is stored. Table 6.2 summarizes the dependency relations that are stored in the Dependency Management System.

6.4 Server Implementation

The server core implementation includes the Service Registry, a network abstraction module and a threading model. The User Management System is not implemented, and thus in the prototype the Service Registry does not offer authentication services. The network abstraction module implements a protocol to transfer service requests, parameters and request results over the network. The threading model has two threads for each client; one thread processes the request and another monitors the network connection and sends progress reports back to the client.

The Dependency Management System is based on Jena 2.0, which is used to store the Dependency Ontology. The system is integrated with the Service Registry as a service. The Development Library is prototyped with a CVS repository. The versioning metadata is automatically provided by the Revision Ontology [31] through CVS interoperability. The Development Library prototype is not currently integrated to the service architecture. The integration could be done by using JCVS[5] [32].

[5]http://www.jcvs.org

The Public Ontology Library and the Publisher are prototyped with a Perl script that is run in a directory where the ontologies have been checked out from the Development Library. The script tags with a given product tag the version currently in the directory for each file. It then creates a subdirectory in a public directory on a WWW server, names the subdirectory with the product tag, and copies all tagged files to that directory. Finally, it publishes the new ontology to the WWW server users by seting the read permissions of the directory and the files.

The prototype implementation was done quickly in order to create a proof of concept for the architecture and test the most interesting system features. Even though all components are not integrated to the Service Registry, the server prototype can be used to test different ontology development processes in a distributed setting. The ONKI Server prototype supports the development, publishing and maintenance phases of the ontology life cycle. Distributed ontology development and maintenance was considered a priority during the prototype implementation.

6.5 Client Implementation

The ONKI Client is based on Protégé. It offers a powerful plugin architecture that allows one to create graphical plugins to perform different tasks related to ontology editing. Tab plugins add a tab in the user interface. The developer can add to the tab whatever functionality he feels necessary. The ONKI Client was implemented as a tab plugin. Other type of plugins can be implemented, too. Slot plugins can be created to display the value of a certain property. For example, if the property value is an URL address, a slot plugin can display the value as a hyperlink and start the default web browser if the user clicks the link. Backend plugins allow developers to create different ontology serializations and knowledge base implementations. Different backend plungins have been implemented, for example an RDF(S) plugin, an OWL plugin, and database backend plugins. They allow users to save their ontology in a different format, effectively translating an ontology from one language to another, or to store the ontology in a database to support large ontologies. At the Protégé's web page (http://protege.stanford.edu/plugins/) exists a library of plugins freely available for download.

Despite its powerful plugin architecture, Protégé API lacks documentation and implementations to some functions. The implementation of the ONKI Client plugin was not easy and the RDF(S) backend plugin source code had to be modified. The client plugin checks the consequences of a change operation after it has been performed on the knowledge base. This is not a serious limitation, because as the user is warned he can use the undo functionality of Protégé or revert to a previous save of the ontology. The information flow is represented in Figure 6.2. The changes that break other ontologies are logged, and the log is saved at the same time as the ontology. Therefore, the log contains only the saved modifications that have broken other ontologies. The log details could be used to automatically specify which elements in other ontologies are broken, and why.

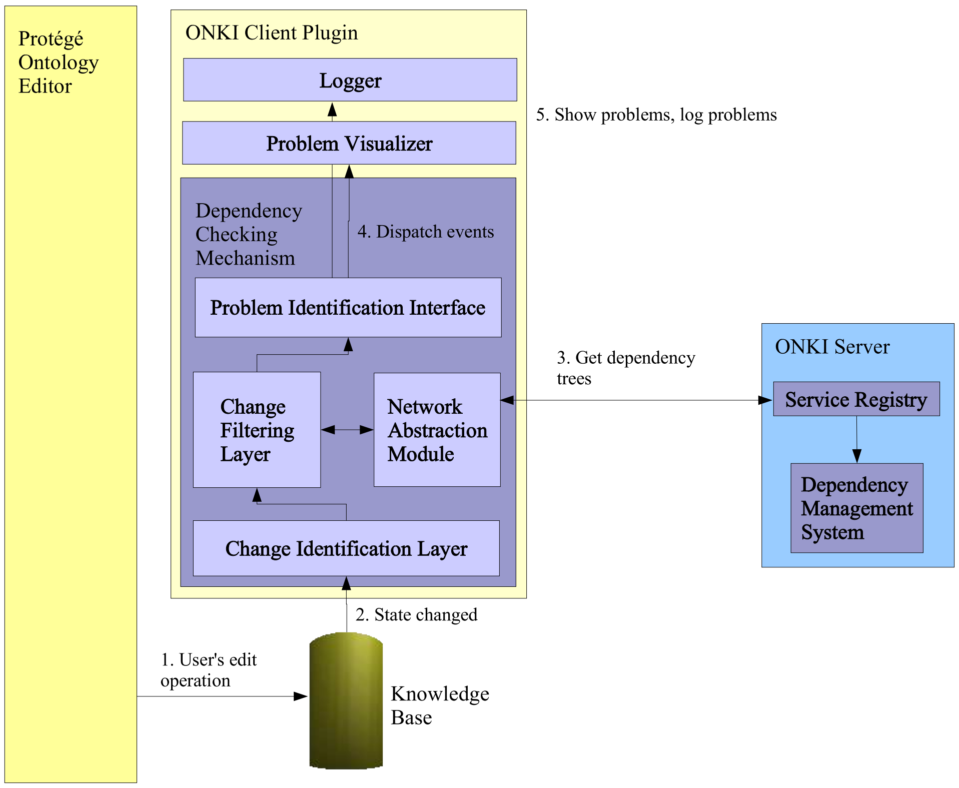


Figure 6.2: ONKI Client protoype.

6.6 Performance Tests

YSO development has not yet started, and it could not be used to evaluate the ONKI architecture. To get some performance data, I set up a development environment for the ontologies created in the museum project, and used it to analyse the performance of the architecture. The primary goal of the performance tests was to demonstrate the architecture in action and show that it is capable of correctly recognizing problem situations. The secondary goal was to evaluate the scalability of the architecture by identifying possible bottlenecks in the prototype. The last goal of the tests was to provide evidence that a usable system can be created from the architecture by showing that the delays caused by the remote dependency checks are not too long to hinder editor usability.

The test data from the museum project totals 9 ontologies. It is formed by 4 upper level ontologies, by 2 referencing ontologies that include the same top level ontology, and by 3 bottom level ontologies that include all the upper level ontologies. The data is presented in Figure 6.3. The top level ontologies are standalone ontologies that do not include other ontologies. The museum domain ontology MAO contains 6757 classes and 11 properties. It does not define any instances. The actor ontology introduces different organizations and individuals that have some role with respect to the items in the museum collections. This ontology has 14 classes, 6 relations and 1715 instances. The locations ontology defines different geographic locations. It has 21 classes, 9 properties and 862 instances. The collections ontology stores information about the different collections in different museums. It defines 10 classes, 23 properties and 123

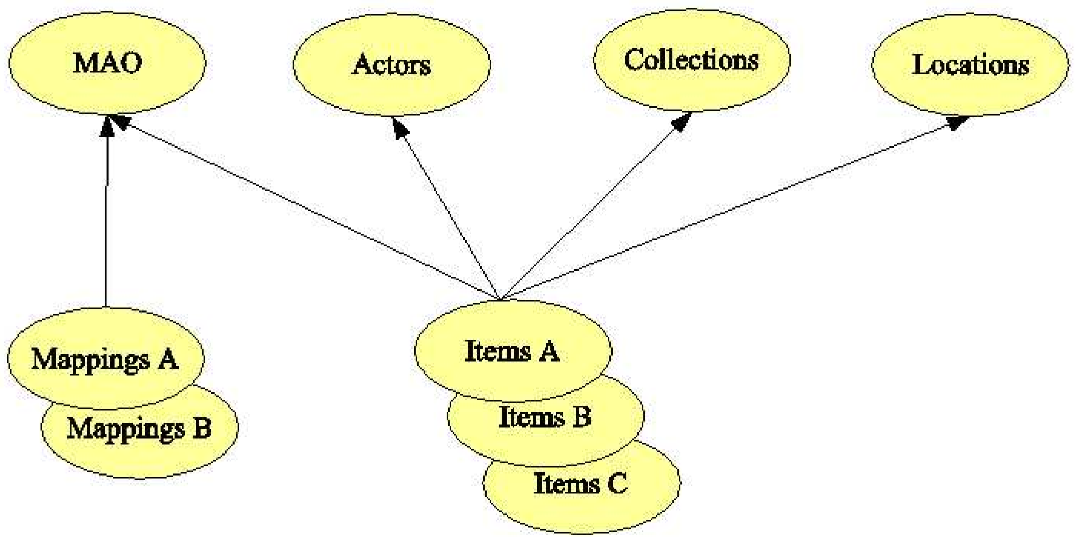


Figure 6.3: ONKI prototype test data.

instances.

The museums had indexed their collection items using a terminology specific for each museum. To add the collection items of a museum into the system, a mapping ontology was created to connect the terms used in the museum indexing to the concepts used in MAO. Similar mapping ontologies had been created for other top level ontologies, too, but they were not available. The mapping ontology for museum A defined 31 classes, 6 properties and 2584 instances. The ontology for museum B had 34 classes, 6 properties and 2473 instances. The mapping ontology for museum C was not available.

To annotate the items of the museum collections, a bottom level ontology had been created for each museum. Each item ontology included all top level ontologies. One instance in the item ontology corresponded to an object in a museum collection. The information about the item was converted to references to other elements in the ontology and in the included ontologies. The resulting item ontologies defined 1 class and 38 properties each. The item ontology for museum A had 1192 instances, the ontology for museum B 1592 instances, and the ontology for museum C had 2110 instances.

The tests started by first loading the ontological dependencies into the Dependency Management System. Ontology development was then simulated by modifying the upper level ontologies to test if the prototype could recognize the problematic edits and how long it would need to process the client requests. During the tests, the ONKI Server was running on machine with a Pentium 4 processor running at 1.8 GHz. 200 MB of memory were allocated to the Java virtual machine. The ONKI client was running as a Protégé plugin on a similar machine.

Each of the bottom level ontologies was loaded into Protégé, and the ONKI client was used to calculate the dependency relations and store them to the Dependency Management System. The same was done for the two MAO mapping ontologies. MAO was then opened with Protégé, and the ONKI system was tested by removing classes and properties, removing properties from classes, and by moving classes to other branches in the class hierarchy. The ONKI Client was modified to recorded the time that it needed to check the effects of each change operation from the server.

6.7 Test Results

The time needed to calculate the dependencies from the mapping ontologies to MAO and then update them to the server was found adequate. The mapping ontology for museum A defined 2584 instances, which created 2918 dependency relations from the ontology to MAO. Updating the dependencies to the server took 4.9 seconds. The mapping ontology for museum B had 2473 instances, and resulted in 1835 dependency relations. Processing took 4.4 seconds.

The item ontologies defined solely new instances that had an included class as their direct type. The instances possibly referenced each other or other included instances. The item ontology for museum A defined 1192 instances, and generated 11956 dependency relations. Updating the server dependency model took 2.6 minutes. The item ontology for museum B added 1592 instances, which led to 17398 dependency relations. The server needed 4.7 minutes to add the relations to the dependency model. The item ontology for museum C defined 2110 instances, and generated 20270 dependency relations. The server dependency model was updated in 6.3 minutes.

After these initial setup operations the Dependency Ontology contained 12979 instances and 50955 dependency relations. Elements of MAO ontology were referenced by 31149 relations, elements of the actor ontology by 8949 relations, elements of the location ontology by 5996 relations, elements or the collection ontology by 4861 relations. The number of references to one element varied between one and 2110. The system was now setup and ready to undergo the functionality testing needed to find out if it performance was adequate for real-life ontology development.

Ontology development was simulated by performing a series of change operations to the largest upper level ontology stored in the system. When MAO was opened and modified, a warning was displayed for all modifications that caused a referencing ontology to break. Checking any individual change took less than 500 milliseconds, even if an entire top level branch was deleted from MAO. The test results are presented in Table 6.3. Columns C, P and I indicate the number of classes, properties and instances the ontology defines. Column RE shows the number of referenced elements in the upper level ontologies, and the number of referencing elements in the bottom level ontologies. Column R shows the number of dependency relations towards the upper level ontologies, and the number of dependency relations from the bottom level ontologies. For the Dependency Ontology, R shows the number of dependency relations stored in the model. Column R/RE shows the average number of dependency relations per referenced element in the top level ontologies, and the average number of dependency relations per referencing element in the bottom level ontologies. The last line in the table shows the size of the Dependency Ontology in the server. The values R and R/RE are not applicable to the Dependency Ontology.

Table 6.3: *Performance test results. C=Classes, P=Properties, I=Instances, RE=Referenced/referencing Elements, R=dependency Relations, R/RE=Relations per Element.*

Ontology	*C*	*P*	*I*	*RE*	*R*	*R/RE*
MAO	6757	11	0	2101	31149	14
Actors	14	6	1715	1368	8949	6
Locations	21	9	862	350	5996	17
Collections	10	23	123	15	4861	324
Mapping A	31	6	2584	2563	2918	1
Mapping B	34	6	2473	1680	1835	1
Items A	1	38	1192	1192	11956	9
Items B	1	38	1592	1591	17398	10
Items C	1	38	2110	2110	20270	10
Dependency Ontology	10	17	12979	-	50955	-

6.8 Results Explained

When the ONKI prototype was tested against the material gathered from a project similar to the one in which the prototype will be used, it proved to be capable of recognizing the problems that editing an included ontology may cause. Even though the initial storing of dependency relations to the system took a rather long time to process, checking the effects of an individual edit was done in less than 500 milliseconds. This response time kept the editor usable at all times. ONKI performance in detecting hazardous changes was thus found quite acceptable. With the test data described in the previous section, the prototype system showed potential to help the ontology developer keep the ontological dependencies consistent while storing a considerable amount of dependency relations (51000 relations) in the Dependency Management System.

The scalability of the ONKI architecture is greatly affected by three factors: the number of dependency relations the system must handle at any client request, the number of dependency relations in the Dependency Ontology, and the ontology engine used inside the Dependency Management System to store that ontology. The prototype implementation used Jena 2.0. A major bottleneck was found in the prototype, where adding new dependency relations took much too long to process. The bottleneck was caused by the sequential search and add operations to the Jena API during a Dependency Ontology update. A rewrite of the update method could improve the performance but the problem is likely to be solved better by replacing the file based Jena ontology with a database backed one.

The number of dependency relations in the Dependency Management System is a function of the number of ontologies stored in the Development Library and of the level of dependencies between the ontologies. The number of relations the system must handle at any client request is itself a function of the level of dependencies between the ontologies. These Figures depend on how the ontologies are reused, and vary heavily with the specific use case of the ontology. From the Table 6.3 we can see that the bottom level ontologies used in the test had a relatively high level of dependency. The amount of referenced elements compared to the total amount of elements in the ontology varied between 10 and 80 percent (MAO 31%, Actors 79%, Locations 39%, and Collections 10%). These Figures were caused by the way the ontologies were reused and linked in the MAO project that served as the test data. Such high levels of dependency suggest problems to ontology maintenance and consistency as the ontologies evolve. They also surface a need for serious process and product management practices in ontology development, publishing and maintenance in order to prevent chaos in the ontology library. Having said that, however, one must remember that the results were caused by the data that came from one project only. The sample data itself might cause bias in the results. Nevertheless, if similar results would be obtained from a more detailed dependency analyses of other ontologies, it would be an important issue for ontology reuse.

The system was found to function as expected with the given test data. If the system works in practice as this set of tests suggest, then the ONKI architecture can be considered as a possible solution to the problem of distributed ontology development. The ONKI prototype successfully identifies the changes that cause problems in distributed ontology development. In the prototype implementation updating the Jena ontology inside the prototype of the Dependency Management System is very slow, but otherwise the prototype performance is high enough to be used in real ontology development. Scalability issues are likely to come up in the function the performance limitations of the ontology engine used in the Dependency Management System.

Chapter 7

Discussion and Conclusions

A set of ontology change operations was presented and analysed in Chapter 3. Existing solutions were briefly described in Chapter 4. Chapter 5 presented an architecture that allowed catching these change operations and evaluating their effects on dependent ontologies. A prototype was implemented as described in Chapter 6 and it was found usable for distributed development of an upper level ontology. The thesis thus reached the goals that were set in Chapter 1. This Chapter will briefly summarize the change analyses, the designed ONKI architecture and the prototype implementation. It will then draw conclusions from the results, discuss their significance, and propose ideas for future work.

7.1 Changing an Included Ontology

Changing an included ontology may cause conflicts in the including ontologies that reference the changed concepts. An edit operation that is performed by the user may be decomposed into a set of atomic changes that limit the effects of the operation. The edit decomposition can be used to calculate a change of the state of the ontology before and after the edit operation to more precisely identify a set of concepts that may be affected by an edit operation. Ontology edit operations themselves can be divided into three categories based on their severity: Removing or changing a URI, removing a superclass or a superproperty or changing it to a one higher in the class or property hierarchy, removing a property from a class and moving a property down the class hierarchy form the first category of changes. These edits modify the class and the property hierarchies and cause invalid references in the including ontologies. They occur frequently in ontology editing and have severe consequences to the including ontologies, causing data loss and large inconsistencies. A distributed ontology development environment needs to protect the developers of inter-dependent ontologies against these changes.

The second category is formed by narrowing a restriction for a slot, merging or splitting classes, moving a property to a referenced class or to a new class, and declaring two classes disjoint and defining a property transitive or symmetric. The effects of these operations are more limited than the ones of the first category, or they occur less frequently. Distributed ontology development environments should help the users with these changes, if they are considered to be common in the specific type of ontologies the users are creating. At least they should be identified by the system.

The last category contains the edits that either virtually never cause problems or are performed very rarely. These operations are adding a new URI, adding a property to a class, adding a superclass or a superproperty, moving a property up the class hierarchy, re-classifying a class as an instance or vice versa, changing a superclass or a superproperty to a class higher in the class or property hierarchy, and widening a restriction for a slot. Ontology editors can fairly safely ignore these kinds of changes when considering conflicts between dependent ontologies. They may be supported for other reasons, for example user convenience; listing similar URIs from the ontology library when the user creates a new concept would improve the editor usability.

7.2 ONKI – an Architecture for Distributed Development of an Ontology Library

The ONKI architecture was designed to be used in distributed development of dependent ontologies. It supports effective ontology management and distributed development in the development and maintenance phases of the ontology life cycle. The safety of edit operations can be checked and the problematic changes caught with a client-server based architecture. The ONKI server is based on a service architecture, where all service requests are first sent to the Service Registry. Using the User Management System, it authenticates the request and creates a secure communication channel between the requester and the service. The most important service is the Dependency Management System, which keeps track of the dependencies between the ontologies in the system. Clients can check the effects of a change operation from there. The Dependency Management System is connected to the Development Library that stores the ontologies under development. The library enforces consistent version control on the ontologies. If an ontology is committed back to the development library and it causes conflicts to some other ontologies that depend on it, the ONKI Messenger automatically notifies the developers of the affected ontologies. It uses the User Management System to find the contact information, and provides different messaging channels such as email or MMS.

A Public Ontology Library contains the ontology versions that have been published. It is designed to be an interface to a multitude of library implementations ranging from a searchable database backed library to a simple WWW site to a programmatically accessible library. Publishing an ontology from the Development Library to the Public Ontology Library is done with the Publisher, which allows a managed deployment process to be established.

The ONKI client consists of a client module, which is inserted in an ontology editor. The Change Identification Layer in the client module interrupts all change operations before they are performed to the editor knowledge base. The Change Filtering Layer determines, which local ontology elements would be affected by the change, and checks from the server what effects those modifications would have in other ontologies. All changes that would cause problems are forwarded to the Problem Identification Interface, which distributes notifications about the problems to all registered listeners. Listener applications can then attempt to solve the problems or even prevent the conflicting changes from being written to the knowledge base. A sample listener application would just show the problems in detail and confirm the problematic edit operations from the user, allowing or discarding them based on his input.

7.3 ONKI Prototype

A prototype of the ONKI architecture was implemented to be used in the distribution of the development of an upper level ontology YSO. The server prototype is based on Jena 2.0 and CVS. The client prototype works on top of Protégé 2.0 ontology editor. The prototype identifies the problematic changes and warns the ontology developer about them. A detailed log is also created, and it can be used to find the cause and consequences of any breaking change. Due to missing method implementations in the Protégé API the client integration required changing the source code of Protégé's RDF(S) plugin. However, all problems were overcome.

The prototype was tested by simulating the development of MAO, a museum domain upper level ontology. The prototype showed good performance in identifying the problematic changes, but updating the Dependency Management System's dependency ontology turned out to be quite slow. The server performance could be increased by using a database backend instead of a file based system. Jena supports relational databases MySQL, PostgreSQL and Oracle through a JDBC connection. Also, modifying the client implementation to send just the modified dependencies based on the log collected would dramatically cut down the number of dependencies the server must process during an update.

7.4 Conclusions

Ontology reuse can be based on the modularization of ontologies and on the inclusion of those modules in other ontologies to form larger entities. A client-server based architecture can be used to implement an ontology library system that allows distributed development of ontologies and helps the developers in developing and maintaining the inter-dependent ontologies. The client-server architecture need only be a logical architecture. In reality, the server architecture can be distributed to balance server load and increase scalability. To do this, a centralized registry is needed where different services can be located. This can be done transparent to the user. It is essential that the ontology developers are kept up to date on the effects of their edits because, in the case of inter-dependent ontologies, a seemingly small change in one ontology can cause massive effects in several other ontologies.

All changes do not cause problems to the referencing ontologies. Change operations can be classified according to the severity of their effects, and the classification can be used as a prioritization for ontology development environment implementations. Different composite changes can be created by grouping individual change operations. The upper boundary for their effects is formed by the effects of the individual changes that form them. However, their effects can be smaller because some of the composing changes can re-establish conditions that were broken by other composing changes. This information can be used to decide at the client end if an operation is safe, despite the fact that it is composed from unsafe operations. This can be used to increase ontology editor performance.

The dependency level between two ontologies varies heavily based on how the ontologies are reused. In the test case the ontology dependencies from the lower level ontologies to MAO and other upper level ontologies proved to be quite strong. Thirty to eighty percent of the upper level ontology concepts were referenced, and the average number of references per referenced element was from 6 to 17 (324 in one case). The average number of references from a bottom level ontology element was 10. In my opinion, such levels of dependency suggest problems with the maintainability of ontologies. However, the test data was from one project only and it might be balanced.

7.5 Significance of Results

To promote ontology reuse, it is important to solve the problems of distributed development of inter-dependent ontologies. Serious ontology development requires a set of quality tools to manage, develop and maintain ontologies and the dependencies between them in a disciplined manner. The results of the thesis show that these tools can be based on a client-server architecture. There exist other systems similar to ONKI, such as the KAON Engineering Server. Those systems may be better in some particular ontology development tasks such as allowing a more powerful ontology inclusion mechanism, propagating the changes to the dependent ontologies, merging two ontologies, translating ontologies from one language to another, offering methodological support to ontology development, or supporting ontology versioning. However, the problem of those systems is, on one hand, the non-impugning approach they take on any single change operation and, on the other hand, the rigid barriers some of them impose between different ontology development tasks. Even if a change has massive negative impacts on the including ontologies, it is never considered if the change should be made in another way. The ONKI architecture embraces change in an intelligent manner, guiding the ontology developers towards conscious development decisions by helping them to create backward compatible changes that keep the ontological dependencies consistent. For this to be possible, one must take an integrated approach to ontology library design. The integrated approach is crucial in creating the full set of easy-to-use utilities that are needed to allow the wide-spread use of ontologies. Such systems should be prepared to handle large numbers of dependencies between ontologies, as the ontologies can be reused with different instance sets.

7.6 Future Work

More research on different ways to reuse ontologies should be done in order to find ways to cut the level of dependencies between ontologies. The development records of MAO indicated one type of reuse, where the including ontologies add mainly instance data. This causes heavy dependence between the ontologies, and is against the sound ontology design principles presented by Gruber [4]. However, it is an intuitive way to reuse ontologies while building intelligent applications, and it would be interesting to know if it is the way most ontologies are being reused. This would shake the image that we have today on ontology reuse.

For ONKI to become a full, industry-class distributed ontology development and management environment, the performance problems related to Jena need to be solved. This was discussed in section 7.3 and a proposition was made to guide future development. The User Management System and authentication should be implemented, and the server components could be distributed to support larger user volumes and bigger ontology libraries. ONKI could be extended by implementing a support for some composite changes such as moving a property up the class hierarchy. These operations could be implemented at the ONKI client. It would reduce the amount of inference that must be done in the Dependency Management System and would thus directly increase performance. Clients could cache relevant parts of the dependency model locally, so that network calls could be reduced. This, again, would improve the system performance.

The Development Library, the Publisher and the Public Ontology Library should be integrated as services in the ONKI architecture. The user could then perform all development tasks from his graphical client, which would make the ontology development process more effective. The ONKI Messenger could be implemented, and support for semi-automatic merging of changes from included ontologies could be done according to the principles presented by Maedche et al. in [15].

References

[1] Sean Bechhofer, Leslie Carr, Carole Goble, and Wendy Hall. Conceptual Open Hypermedia = The Semantic Web? In *Proceedings of the WWW2001, Semantic Web Workshop*, Hongkong, 2001.

[2] R. Struder, V.R. Benjamins, and D. Fensel. Knowledge engineering: Principles and methods. *IEEE Transactions on Data and Knowledge Engineering 25(1–2)*, pages 161–197, 1998.

[3] D. Fensel. *Ontologies: Silver Bullet for Knowledge Management and Electronic Commerce*. Springer-Verlag, 2001.

[4] T. R. Gruber. Towards Principles for the Design of Ontologies Used for Knowledge Sharing. In N. Guarino and R. Poli, editors, *Formal Ontology in Conceptual Analysis and Knowledge Representation*, Deventer, The Netherlands, 1993. Kluwer Academic Publishers.

[5] Ljiljana Stojanovic, Alexander Maedche, Boris Motik, and Nenad Stojanovic. User-driven ontology evolution management. In *European Conf. Knowledge Eng. and Management (EKAW 2002)*, pages 285–300. Springer-Verlag, 2002.

[6] A. Maedche, B. Motik, L. Stojanovic, R. Studer, and R. Volz. An infrastructure for searching, reusing and evolving distributed ontologies. In *Proceedings of the twelfth international conference on World Wide Web*, pages 439–448. ACM Press, 2003.

[7] Julio C. Arpírez, Oscar Corcho, Mariano Fernández-López, and Asunción Gómez-Pérez. Webode: a scalable workbench for ontological engineering. In *Proceedings of the international conference on Knowledge capture*, pages 6–13, Victoria, British Columbia, Canada, 2001. ACM Press.

[8] Ying Ding and Dieter Fensel. Ontology library systems: The key to successful ontology re-use. In *The first Semantic web working symposium (SWWS1)*, Stanford, USA, July 29–August 1 2001.

[9] Hyperdictionary home page. http://www.hyperdictionary.com/computing/ontology, accessed March 5, 2004.

[10] M. Uschold and R. Jasper. A framework for understanding and classifying ontology applications. In *Benjamins VR (ed) IJCAI '99 Workshop on Ontology and*

Problem Solving Methods: Lessons Learned and Future Trends. Stockholm, Sweden. CEUR Workshop Proceedings 18:11.1-11.12. Amsterdam, The Netherlands, 1999.

[11] A. Farquhar, R. Fikes, and J. Rice. The Ontolingua server: A tool for collaborative ontology construction. Technical report, Stanford KSL 96-26, 1996.

[12] Jeff Heflin and James A. Hendler. Dynamic ontologies on the web. In *AAAI/IAAI*, pages 443–449, 2000.

[13] J. Domingue. Tadzebao and WebOnto: Discussing, Browsing and Editing on the Web. In *Proceedings of the 11th Knowledge Acquisition for Knowledge-Based Systems Workshop,* Banff, Canada, April 18th–23th 1998.

[14] N. Noy and M. Klein. Ontology evolution: Not the same as schema evolution. *Knowledge and Information Systems*, 5, 2003.

[15] A. Maedche, B. Motik, and L. Stojanovic. Managing multiple and distributed ontologies on the semantic web. *The VLDB Journal*, 12(4):286–302, 2003.

[16] R.L. Leskinen, editor. *Museoalan asiasanasto*. Museovirasto, Helsinki, Finland, 1997.

[17] Yleinen suomalainen asiasanasto. http://vesa.lib.helsinki.fi/ysa/, accessed March 23, 2004.

[18] Maedche, Alexander and Motik, Boris and Stojanovic, Ljiljana and Studer, Rudi and Volz, Raphael. Ontologies for enterprise knowledge management. *IEEE Intelligent Systems*, 18(02):26–33, 2003.

[19] W. Grosso, H. Eriksson, R. Fergerson, J. Gennari, S. Tu, and M. Musen. Knowledge modeling at the millennium – the design and evolution of protege-2000. In *Proceedings of the 12 th International Workshop on Knowledge Acquisition, Modeling and Mangement (KAW'99)*, Banff, Canada, October 1999.

[20] Dieter Fensel, Ian Horrocks, Frank van Harmelen, Stefan Decker, Michael Erdmann, and Michel C. A. Klein. OIL in a nutshell. In *Knowledge Acquisition, Modeling and Management*, pages 1–16, 2000.

[21] I. Horrocks D. L. McGuinness P. F. Patel-Schneider D. Connolly, F. van Harmelen and L. A. Stein. DAML+OIL (March 2001) Reference Description, http://www.w3.org/TR/daml+oil-reference, March 2001, accessed March 1, 2004.

[22] Sean Bechhofer, Ian Horrocks, Carole Goble, and Robert Stevens. OilEd: a reason-able ontology editor for the semantic web. In *Proceedings of KI2001, Joint German/Austrian conference on Artificial Intelligence*, number 2174 in Lecture Notes in Computer Science, pages 396–408, Vienna, September 2001. Springer-Verlag.

[23] A Gomez-Perez, M Fernandez, and A.J. De Vicente. Towards a method to conceptualize domain ontologies. In *ECAI-96 Workshop on Ontological Engineering*, Budapest, Hungary, 1996.

[24] M Fernandez, A Gomez-Perez, and N. Juristo. METHONTOLOGY: From ontological art towards ontological engineering. In *AAAI-97 Spring Symposium on Ontological Engineering*, Stanford University, March 24th–26th 1997.

[25] Y. Sure, M. Erdmann, J. Angele, S. Staab, R. Studer, and D. Wenke. OntoEdit: Collaborative ontology development for the semantic web. In *Proceedings of the first International Semantic Web Conference 2002 (ISWC 2002), June 9-12 2002, Sardinia, Italia*. Springer, LNCS 2342, 2002.

[26] Michel Klein, Dieter Fensel, Atanas Kiryakov, and Damyan Ognyanov. Ontology versioning and change detection on the web. In *13th International Conference on Knowledge Engineering and Knowledge Management (EKAW02)*, Sigüenza, Spain, October 1–4 2002.

[27] Free Software Foundation. Concurrent versions system, http://www.cvshome.org, accessed February 23, 2004.

[28] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2000.

[29] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. GOTOP Information Inc., 5F, No.7, Lane 50, Sec.3 Nan Kang Road Taipei, Taiwan; Unit 1905, Metro Plaza Tower 2, No. 223 Hing Fong Road, Kwai Chung, N.T., Hong Kong, 1996.

[30] Jeremy J. Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne, and Kevin Wilkinson. Jena: Implementing the semantic web recommendations. Technical Report HPL-2003-146, HP Labs, December 24, 2003.

[31] T. Kauppinen. Ontology versioning framework. Master's thesis, University of Helsinki, forthcoming in 2004.

[32] The JCVS home page, http://www.jcvs.org/, accessed February 24, 2004.

Appendix A

Effects of Changes Prioritized

Table A.1: *Effect of changes for classes, properties, instancies and dependency relations.*

Priority	*Operation*	*Comment on effect*
High	Deleting a class, a property, or an instance	Data loss, breaks referencing ontologies
High	Changing a URI	Equals to delete in referencing ontologies
High	Removing a superclass or a superproperty	Identity of subelements changes. Breaks referencing ontologies
High	Moving a class in the class hierarchy	May be safe, or may equal removing of several superclasses
High	Changing the superclass or the superproperty to a class or a property higher in the hierarchy	Equals removing some superclasses or superproperties
Medium-High	Removing a property from a class	Loss of data. Breaks referencing ontologies
Medium-High	Moving a property down the class hierarchy	Equals removing a property for some classes
Medium	Narrowing a restriction for a slot	Data loss. Introduces hard to find conflicts into referencing ontologies
Medium	Merging classes	Breaks referencing ontologies. Common operation in YSO development and maintenance
Medium	Splitting a class	Breaks referencing ontologies. Common operation in YSO development and maintenance
Medium-Low	Moving a property to a referenced class	Breaks referencing ontologies. Usefull operation in ontology restructuring

Table A.1: *Effect of changes for classes, properties, instancies and dependency relations.*

Medium-Low	Moving a property to a new class	Breaks referencing ontologies. Usefull operation in ontology restructuring
Low	Declaring two classes disjoint	Rare operation. Breaks the semantics of referencing ontologies
Low	Defining a property transitive or symmetric	More common than declaring classes disjoint, but less dangerous. Breaks the semantics of referencing ontologies
Very Low	Adding a new URI: a class, a property or an instance	Common operation. Problems occur very rarely, limited consequences
Very Low	Adding a property to a class	Very rarely conflicts if the same property exists in a subclass and property restrictions conflict
Very Low	Adding a superclass or a superproperty	Safe, may introduce same problems as adding a property. Very safe operation
Very Low	Moving a property up the class hierarchy	May cause same problems as adding a property if the class has other subclasses which already have the property
Very Low	Re-classifying a class as an instance	Very rare operation. Loses data. Breaks referencing ontologies
Very Low	Re-classifying an instance as a class	Very rare operation. Breaks referencing ontologies
Safe	Changing the superclass or the superproperty to one higher in the hierarchy	No problems for the class or property in question
Safe	Widening a restriction for a property	Completely safe change