An Architecture for Collaborative Ontology Library Development

Tuomas Korpilahti and Eero Hyvönen

University of Helsinki, Department of Computer Science
P.O. Box 26 (Teollisuuskatu 23)
FIN-00014 UNIVERSITY OF HELSINKI, Finland
Semantic Computing Research Group
http://www.cs.helsinki.fi/group/seco/
tuomas.korpilahti@cs.helsinki.fi, eero.hyvonen@cs.helsinki.fi

Abstract. We consider collaborative ontology library development, where ontologies are re-used by including their resources in other ontologies. When an included ontology is changed the ontology including resources from the modified one may become obsolete. This paper analyzes the effects of ontology change operations to dependent ontologies and metadata. A client-server architecture for co-operative ontology library development is presented that helps to make ontology changes in such a way that the ontology versions remain backwards compatible as much as possible. The system also tells the developers the effects of a change to other dependent ontologies and repositories. In this way accidentally destructive modifications can be avoided and the other developers of the ontology library can find out what modifications are necessary in their own ontologies for interoperability. A prototype of the system on top of the Protégé-2000 ontology editor has been implemented and preliminary tests for using it are presented.

1 Towards Collaborative Ontology Development

Ontologies [1, 2] are machine comprehensible knowledge models that explicitly model real world concepts and relatons. They should be publicly shared and interoperable in order to make possible the full benefits of the Semantic Web [1]. According to Gruber's ontology design principles [3], each real world concept is defined only in one ontology, and all systems use the same definition for that concept. Ontologies should hence be reusable to let ontology engineers benefit from the increasing number of different domain-specific ontologies.

1.1 Ontology Life Cycle

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 according to the evolving prototype model [4], the changes may prove critical for the including ontology and introduce inconsistencies that make the latter obsolete or even unusable. The problem originates largely from the fact that mutually dependent ontologies are developed co-operatively in geographically distributed locations by different people. A co-operative ontology library development and maintenance architecture needs to be established in order to manage the evolution of dependent ontologies [5–7]. Nevertheless, methods and tools to support this complex task completely are missing [8].

Ontology management consists of development, publishing, evolution, and maintenance processes (cf. figure 1). During the *development process* the ontology developer creates a new ontology by (possibly) re-using existing ones. The *publishing process* is used to control public access to the ontology; it ensures that the correct version of the ontology is available in different contexts. For example, the ontology developers may want to work with a different version of the ontology than the application developers or the end-users. After publishing the ontology for the first time, the *maintenance process* begins. Here the ontology is modified due to various needs. For example, the domain modeled or the needs of the ontology users may change, and the ontology must be adopted to the changed use conditions. When ontologies are mutually dependent, i.e., when ontologies include resources from each other, changes need to be propagated from one ontology to another. This is supported by the ontology *evolution process* that includes ontology change and change propagation processes. The maintenance process includes successive evolution and publishing processes, which react to the changed user needs of the ontologies and keep feeding the ontology users with up-to-date versions of the ontology. *Decommission* ends ontology's life.

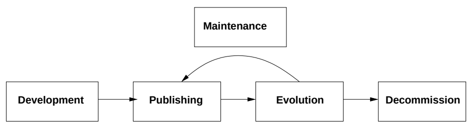

Fig. 1. Ontology life cycle processes.

1.2 State of the Art

There are approaches and systems that aim to solve the problems presented. KAON [6, 9, 5] is an ontology development environment developed at the University of Karlsruhe. In KAON architecture the included ontologies are duplicated. Changes are allowed only to the original ontology and are propagated to the duplicates, causing more changes in the ontologies that are re-using them. The re-users are asked if they would like to accept the proposed changes or keep the older version of the included ontology. KAON propagates the changes between the remote ontology copies fluently. It provides tools to help the re-user to understand the impacts of the proposed changes and lets them decide if the changes are to be executed. However, KAON does not provide means for the user who made the original change to understand what problems his change will cause for others. This means that the developer of a re-used ontology is not capable of getting information for evaluating different modeling decisions. The system is not capable of notifying the developer when he makes a change that breaks the backwards compatibility of his ontology. KAON also lacks the functionality to support ontology publishing process, which is one of the key components of the ontology management process. The system presented later in this paper attacks these problems by providing an integrated tool to support ontology engineering workflow.

Protégé-2000 [10] is a popular ontology editor. It offers an intuitive graphical user interface, and its support for ontology inclusion is one of the best in the field. Protégé-2000 can store ontologies in files and in relational databases, but ontology inclusion is supported only in file-based ontologies. The system maintains ontological consistency by preventing users from modifying 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 loaded. However, Protégé-2000 does not provide means to see what effects the change has in the including ontologies. Developers can thus easily break the including ontologies. No evolution support is available. Publishing and other management related issues are not considered, either. Our system extends Protégé-2000 by managing the ontological dependencies and by adding ontology evolution and publishing support.

Ontolingua [11] is a collaborative ontology development tool that supports even cyclical ontology inclusion. Despite of its powerful inclusion mechanism, it does not provide support for changing the original ontology and then propagating the changes to the including ontologies.

WebODE [7] is a web based ontology editor initially developed for editing OIL ontologies. It uses a database to store ontologies. WebODE utilizes the concept of user groups to establish access control to ontologies, and has synchronization mechanisms to prevent errors from concurrent access. In a wide sense, this could be though of as a way to publish an ontology by allowing public access to it. The mechanism does not support concurrent use of multiple ontology versions for different interest groups, such as ontology re-users, application developers, and end-users. The authors of WebODE state that it supports collaborative ontology editing at the knowledge level, but do not explain more precisely what this means. According to our understanding, WebODE does not provide means to propagate ontology changes to dependent ontologies in a well-controlled manner.

OntoEdit [12] is an ontology engineering environment, which concentrates on tasks related to ontology requirements specification, refinement, and evaluation. It combines methodology-based ontology development with collaboration and inference. Due to its focus on the early phases of the ontology life cycle, it does not offer any support for ontology publishing or evolution.

To contribute to the research above, our paper presents a solution approach that integrates the aspects of consistent ontology development, ontology publishing, and ontology life cycle management into one tool. The tool provides ontology developers a way to evolve any re-used ontologies in a backwards compatible way and stay informed about the relevant changes in ontologies that they are re-using. New ontology versions can be published in a controlled manner. Distributed ontology development is supported, and ontology publishing and maintenance processes can be put in place.

In the following, we first present an analysis of ontology change operations with respect to ontology dependencies, class and property hierarchies and instance data preservation. Based on the analysis, solutions are suggested for the needs of each process presented, and a prototype implementation is described. The paper concludes with a discussion and a summary of our findings.

2 Ontology Change Operations

Ontology inclusion means in this paper (as in Protégé-2000 [10]) that ontologies are explicitly imported into another ontology in a predefined order without duplicating resources. Cyclic dependencies and ontology alignment are not considered. An included ontology O is considered backwards compatible if the ontologies that include a previous version of O can replace the previous version with the new version without conflicts or data loss. Our goal is to create a system that helps the developer to develop backwards compatible ontologies while allowing maximal freedom of ontology modification. The definition of ontology inclusion allows us to change the included ontology if the change does not affect the parts that are re-used in other ontologies. However, sometimes an ontology requires so profound changes that it is not possible to keep it backwards compatible. In this case the system should inform the user what consequences his change would cause, and use this information to help solve the dependency problems if the change is executed.

Noy and Klein [13] identified 22 ontology change operations and analyzed them with respect to the instance-data preservation dimension. They describe what data would be lost at the instances in the changed ontology and would there be 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 operations by performing the other operations in a certain sequence. As a consequence, the same results are presented several times for different composed operations in their analysis. 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 analyze the more complex operations by analyzing the combined effects of the atomic operations that are used to compose them. Our more concise set of 16 atomic operations [14] is based on the change operations of Noy and Klein [13]. In the set we have selected the operations that can be used to compose the other operations. In addition to the instance-data preservation dimension, their effects on the class and property hierarchies are also analyzed, and the change effects on the ontological dependencies are considered when a re-used ontology is changed. The analysis is limited to the dependencies of concepts in RDF(S) context. We do not consider the semantic effects of e.g. changing documentation or concept labels. Table 1 summarizes the atomic change operations and the analysis results fully described in [14]. We demonstrate the idea here by presenting the analysis of the operation Removing a property from a class.

Table 1. Change operations and their effects. C = class hierarchy, P = property hierarchy, I = instance data, DEP = ontology dependencies.

Operation	*Critical for*			
	C	*P*	*I*	*DEP*
Creating a class, a property or an instance				
Deleting a class	X		X	X
Deleting a property		X	X	X
Deleting an instance			X	X
Adding a property to a class				
Removing a property from a class	X		X	X
Adding a superclass				
Removing a superclass	X		X	X
Adding a superproperty				
Removing a superproperty		X	X	X
Re-classifying an instance as a class			X	X
Re-classifying a class as an instance	X		X	X
Declaring two classes disjoint	X		X	X
Defining a property transitive or symmetric		X	X	X
Widening a restriction for a property				
Narrowing a restriction for a property		X	X	X

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 P. The class C is no longer inherited to the domain of P's subproperties, 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 of re-using an ontology is to re-use and enrich the semantic information stored in it. When this information is altered, the context of the ontology changes. As a consequence, the semantics of the change in the including ontologies are unknown. At worst, the change can break the semantics of the ontologies including the changed ontology. In addition, the including ontologies continue to refer to P from the instances of C and of its subclasses. These references are broken. The change loses data on instance level, may break the semantics of the class model, and may introduce invalid references in referring ontologies.

The analysis results of atomic changes can be used to analyze more complex ontology change operations. A complex ontology change operation can be decomposed into atomic operations. The effect of the composite change cannot be greater than the combined effects of the individual changes composing it. However, 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 re-inherited in the second phase. This happens if the new superclasses have those properties. The values of those properties need not be lost. The system would 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 equals to calculating the difference between the state of the ontology before and after the composite change.

3 Ontology Change Process

The ontology developer performs edit operations on the knowledge base. An edit operation is defined as a set of atomic change operations grouped together. The changes can be performed sequentially or in parallel. The mode of execution is determined by the edit's designer. Some changes need to be performed in a specific order to accomplish the desired result. Performing them in another order would produce different post operation state. Sometimes there is freedom in the execution order. For example, when merging several classes into one, the result is the same regardless of which two classes are merged first. In these cases, allowing parallel execution allows knowledge base performance optimizations.

The edit operations that the ontology developer does must be captured and their effects validated before they are executed on the underlying knowledge base. Figure 2 presents a model on how to do this in a distributed environment. A change identification layer is needed between the ontology editor user interface and the knowledge base. Listeners that are registered to this layer receive all user operations and can cancel them before they are performed. The listeners form a chain of validators that the operation must pass in order to be executed. The identification layer acts as a checkpoint through which all edit operations must pass. It allows one to intervene and evaluate the safety of the edits before they are written in the knowledge base. The identification layer must guarantee that the atomic changes are correctly grouped into edits. Without grouping, it is not possible to compute the post operation state. Having intercepted an edit operation, the identification layer forwards it to a change filtering layer. It checks if the edit causes conflicts in the including ontologies. If not, the edit can be safely performed and it is forwarded to the knowledge base, otherwise corrective measures should be taken.

Each change in an edit operation affects a certain number of concepts in the ontology to be changed. The change filtering layer first determines the concepts that will be affected by the edit's changes in the ontology to be changed and

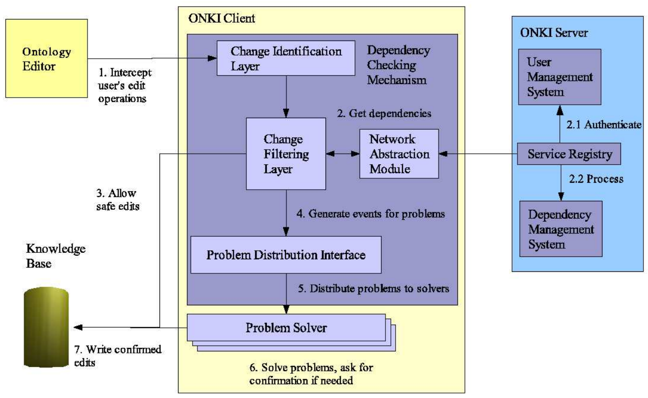
ONKI Client
Ontology Editor
1. Intercept user's edit operations
Change Identification Layer
Dependency Checking Mechanism
2. Get dependencies
Change Filtering Layer
Network Abstraction Module
3. Allow safe edits
4. Generate events for problems
Knowledge Base
Problem Distribution Interface
5. Distribute problems to solvers
Problem Solver
7. Write confirmed edits
6. Solve problems, ask for confirmation if needed
ONKI Server
User Management System
2.1 Authenticate
Service Registry
2.2 Process
Dependency Management System

Fig. 2. Ontology change process.

calculates what the difference in their state would be before and after the edit. The layer checks from the server if the concepts whose state would be changed are referenced in the including ontologies. If they are, the change effects in the including ontologies are calculated starting from the referencing elements. If the effects would cause a conflict or loss of data in the including ontologies, the system has found a problem. The user cannot see these problems by looking only at the ontology he is changing. Identifying them needs information on how the ontology is re-used in the including ontologies. This implies that there must be a common access point to all ontologies and to the dependency information. A server based ontology library is suggested for this purpose. One should note that our approach includes a compromise. It is not possible to guide the development of re-used ontologies without knowledge on what in them is essential for the re-users. Confidential ontologies may not want to publish this information. Complete change logs could be used to support their development.

An edit operation that causes problems is forwarded to a problem dispatching interface. There different solvers can listen to problems and attempt to find solutions to them. Each edit may cause several problems. The problems and their effects are grouped together with the edit to identify the root of the problem. The interface dispatches the edit to a chain of registered problem solvers. Each solver may solve any number of problems associated with the edit, cancel the edit completely or force the edit to be performed. Automatic or semi-automatic problem solving methods could be used. The force operation is meant for allowing the user to perform the edit despite of the effects it causes to other ontologies. If the edit is canceled in the solver, its processing ends. If the edit is forced to be accepted, it first passes all solvers and is then directed to the knowledge base. The edit and the problems it causes are recorded to a log that can be used to identify and help solve the problems in the including ontologies. Unless forced to be accepted, the edit is canceled if all problems are not solved. Finally, the edits that pass all validators in the filtering layer are written to the knowledge base. A notification is generated for a knowledge base modification interface to let the user interface components refresh their view of the knowledge base.

4 Ontology Management Process and Change Propagation

A system required for ontology dependency management is depicted in Figure 3. It is a client-server architecture acknowledged to be the key to successful collaborative editing [8]. The client-server model needs only be a logical model, which allows distributing the knowledge base itself as the ontologies in the library grow larger and the system gains more users.

The server side knowledge base allows wrapping the persistent ontology storage inside a version control mechanism that is considered a very important ontology library service [8]. Consistent versioning is a prerequisite for successful dependency checking. It is imperative to know which version of an included ontology should be used with the referencing ontology. There are two fundamentally different possibilities here. One is to try to keep the including ontology always up to date and use the most recent included ontology version available. This is the situation when a large ontology is divided into smaller modules. Each module must be working with the newest version of the other modules in order for the entire ontology to be consistent and usable. The other possibility for ontology inclusion is to refer to a version older than the most recent one. This case is also very common, as applications and domain specific ontologies may not need to be of the most bleeding edge technology but rather to be guaranteed to work without problems. In this context developers often opt for older versions of software and technology, arguing that it is better tested and its problems are better known. If this is the case, the change filtering layer need not worry about the references coming from such an ontology — as the user is making a change in the included ontology, he is implicitly creating a new version of it. It is thus not necessary to worry about being backwards compatible with ontologies that are not committed to keep up with the development of the modified ontology.

As a new version of an included ontology is committed to the library, the system can automatically send notifications to the developers of the ontologies that include the changed ontology. A flood of notifications should not occur, because both parties want to minimize their number. The developers of re-usable ontologies want to avoid changes having an impact beyond local scope. Re-users want to minimize the number of modifications required in their ontology after a change in an included ontology. A log of problems was created when the developers of the included ontology made modifications that cause problems to other users. Based on this log, the messenger can include in the notifications personalized instructions for each ontology developer on what the new version of the included ontology has broken in their ontology. The messenger should implement an abstraction of the notification carrier – email, short messages, instant messages, or even multimedia messaging could be used. The message

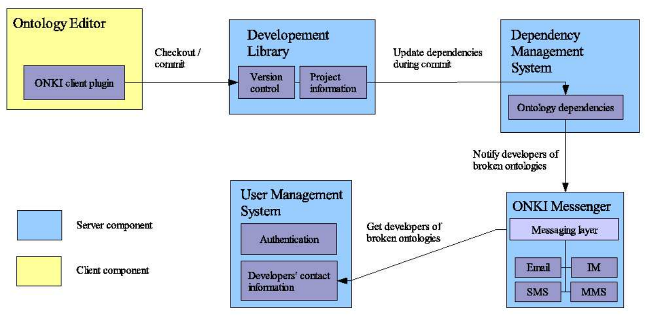

Fig. 3. Ontology management process and change propagation.

carrier could be selected based on the importance of the notification. As the change filtering layer determines the effects of an edit for the including ontologies, it can assign a priority or severity metric for the edit's effects. This metric can be used to determine the message carrier. In order to be able to send the messages, the system needs to register which developers are developing which ontologies, and record contact information for the developers. The same module could be used for user authentication and for providing communication security, which are needed in a network architecture.

When the developer of an including ontology receives the notification that a change in an included ontology has caused his ontology to break, he can use the message to identify the problem and consider the changes required to make his ontology compatible with the new version. If he decides to adapt the ontology to the changes, the message details can be used to help him do the changes semi-automatically. He may also wish to keep the older version, in which case the decision is written in the dependency system. [6]

5 Ontology Publishing Process

The ontologies under development should be kept apart from the published ontologies freely available to the wide audience as illustrated in Figure 4. The reason for this is twofold. On one hand, application development needs the same ontology version to be used widely in applications in order to maximize software interoperability. This requires that a specific ontology version is published as the one to be referenced and re-used, and then promoted to the wide audience. Otherwise, it is highly likely that each application would use a version of the ontology that is different from and possibly incompatible with the versions used by the other applications. Reference ontology versions would undoubtedly arise, but it would take longer as they would need to accumulate a certain mass of interoperable applications to become de facto standards. On the other hand, an ontology, as any software, needs to be updated and is likely to contain bugs. The end users should be protected against the possibility of accidentally upgrading to a beta quality product. Therefore, allowing one to publish an ontology and thereby declare it to be complete, tested, and supported becomes an essential requirement for an ontology library.

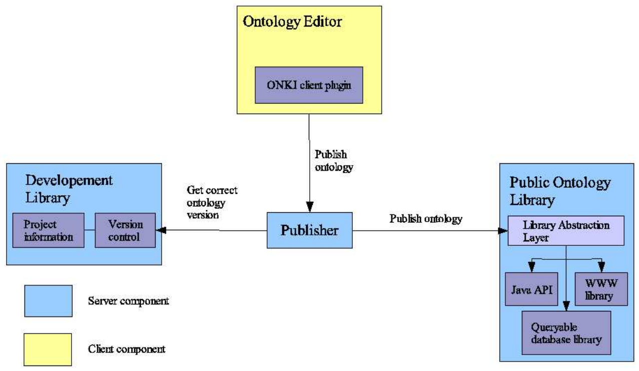

Fig. 4. Ontology publishing process.

Publishing an ontology from the development library to a public ontology library means that a specific ontology version, together with the correct versions of the ontologies it includes, is made publicly available. The ontology library should support this process. The interface to the public ontology library should abstract the different implementations of the library. For example, the ontologies could be browsed via WWW, queried through an interface, and downloaded via different software APIs. When a new version of an ontology is inserted, it should be linked with the previous version. Despite all this, the interface to add an ontology to the library should not be affected.

6 ONKI Prototype

An architecture supporting the processes discussed above is presented in Figure 5. A prototype implementation of the architecture called ONKI [14] has been built at the University of Helsinki as a part of the National Ontology Project in Finland[1]. ONKI uses Protégé 2.0 [10] as the ontology editor. The identification layer, the change filtering layer, and the problem dispatching layer are implemented as a Protégé plug-in. Combined they serve as an editor to demonstrate the ontology change process.

The ONKI server prototype is based on Jena 2.0[2] [15]. The dependency relations from a concept in an including ontology to a concept in an included ontology are stored into a Dependency Ontology in Jena. As the included ontology

1 http://www.cs.helsinki.fi/group/seco/ontologies/
2 http://jena.sourceforge.net

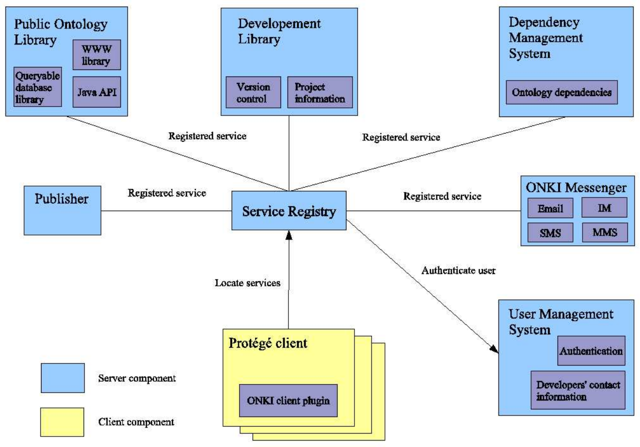

Fig. 5. ONKI architecture.

is changed, the change filtering layer calculates the affected ontology elements in the changed ontology and checks from the server if they are referenced from other ontologies. If they are, a warning message is displayed to the user and the details of the edit operation and of the reference are written into a log file. The centralized knowledge base and the publishing process are prototyped with CVS [16] and with a Perl script that tags a specific version of the ontologies and copies it to a WWW server. The prototype does not yet include the messenger notification service.

The ONKI prototype was tested with ontologies and metadata from the semantic portal MuseumFinland [17, 18]. In the tests, the development was simulated using the prototype. The primary goal was to demonstrate the architecture in action and show that it is capable of correctly recognizing problems.

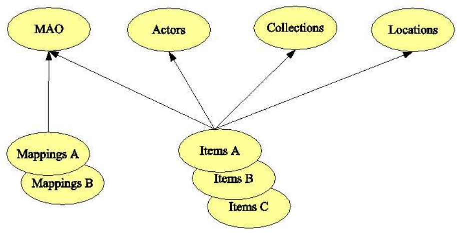

Fig. 6. ONKI prototype test data.

The test data totals 9 ontologies divided into 4 upper level ontologies, 2 referencing ontologies that include the same top level ontology, and 3 bottom level ontologies that include all the upper level ontologies. The sizes and characteristics of the ontologies can be found in Table 2. The inclusion structure of the data is presented in Figure 6.

The museum domain ontology MAO is used to structurize the domain of museum collection items. The Actors ontology introduces different organizations and individuals that have some role with respect to the items in the museum collections. The Locations ontology defines different geographic locations. The Collections ontology stores information about the different collections in different museums. 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. Mapping ontologies for museum A and B were available for the test. Similar mapping ontologies had been created for other top level ontologies, too, but they were 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 ontological dependencies were first loaded into the Dependency Management System's Dependency Ontology. The mapping ontologies had been used as a bridge between MAO terminology and the corresponding museum's local terminology, so they were heavily connected to MAO. 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 and thus were also heavily dependent on the included ontologies.

After the initial setup operations the Dependency Ontology contained 12979 instances and 50955 dependency relations. The test setup is presented in Table 2. 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 RE and R/RE are not applicable to the Dependency Ontology.

Ontology development was simulated by performing a series of edit operations to the largest upper level ontology. When MAO was opened and modified, a warning was displayed for all modifications that caused a referencing ontology

Table 2. *Performance test setup. 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	-

to break. Checking any individual edit took less than 500 milliseconds, even if an entire top level branch was deleted from MAO. The number of dependency relations in the Dependency Ontology and the number of relations the system must handle at any client request are functions of the number of ontologies stored in the Development Library and 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. In this case the bottom level ontologies had a relatively high level of dependency. The amount of referenced elements compared to the total amount of elements in the ontology varied from ten to eighty percent.

These figures were caused by the way the ontologies were re-used in the museum 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 underline the need for serious process and product management practices in ontology development in order to prevent chaos in the ontology library. However, the results were caused by the data that came from one project only. The sample data might cause bias in the results. Nevertheless, if similar results would be obtained from a more detailed dependency analysis of other ontologies, it would be an important issue for ontology re-use.

The system was found to function as expected with the given test data. 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 turned out to be very slow, but otherwise the performance is high enough to be used in real ontology development. Scalability issues are likely to come up in the function of the performance limitations of the ontology engine used in the Dependency Management System.

We expect to start the development of a modular upper level ontology using this prototype. The aim of the project is to create a general upper level ontology that can be used as a basis for domain specific ontology development in Finland. The work requires a solid framework that supports the ontology life cycle from both development and management points of view. We expect to further test the processes and architecture described in this paper and validate the requirements for tomorrow's ontology engineering environment. The prototype serves as a framework to test the analysis of the effects of ontology change operations on dependencies between ontologies and to test different problem solving methods to solve conflicts in distributed, collaborative development.

7 Discussion

Ontology re-use can be based on the modularization of ontologies and on the inclusion of those modules to form larger ontologies. However, in order to support controlled ontology evolution over time, the resulting ontological dependencies must then be managed in order to keep the ontologies mutually consistent in a library. This paper presented analysis of ontology change operations that form a basis to understand the effects of an edit operation on the ontological dependencies. 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.

Inter-dependent ontology libraries are typically developed by communities of collaborators. Collaborative ontology re-use in a distributed environment requires tool support to guarantee the consistency of the ontology library. To implement such tools, it is necessary to integrate together different aspects of ontology development: creation, versioning, publishing, re-use, and evolution. To support collaborative, distributed ontology development, a client-server based ontology library architecture and its prototype ONKI was presented. ONKI helps the development of backwards compatible ontologies and lets the developers to find out and compare the effects of different edit operations. In our view, consistent ontology development, versioning, and publishing should be supported in industry-quality tools for developing ontology oriented applications. However, current solutions are mostly concerned with ontology creation and re-use, and lack efficient maintenance and process support that is required for long-term ontology usage. ONKI integrates managed ontology publishing and evolution processes in order to support developers in ontology maintenance problems.

In our further work, we plan to investigate ways to automatically solve the conflicts identified during editing, and to more accurately reflect the processes of ontology development in an integrated tool.

References

1. D. Fensel. *Ontologies: Silver Bullet for Knowledge Management and Electronic Commerce.* Springer-Verlag, 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. 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.
4. Asuncíon Gómez-Pérez, Mariano Fernández-López, and Oscar Corcho, editors. *Ontological Engineering.* Springer-Verlag, 2004.
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. 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.
10. 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.
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. 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.
13. N. Noy and M. Klein. Ontology evolution: Not the same as schema evolution. *Knowledge and Information Systems*, 5, 2003.
14. Tuomas Korpilahti. Architecture for Distributed Development of an Ontology Library. Master's thesis, Helsinki University of Technology, Department of Computer Science, April, 2004.
15. 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.
16. Free Software Foundation. Concurrent versions system, http://www.cvshome.org, accessed February 23, 2004.
17. E. Hyvönen, M. Junnila, S. Kettula, E. Mäkelä, S. Saarela, M. Salminen, A. Syreeni, A. Valo, and K. Viljanen. Finnish Museums on the Semantic Web. User's perspective on museumfinland. In *Proceedings of Museums and the Web 2004 (MW2004), Arlington, Virginia, USA*, 2004. http://www.archimuse.com/mw2004/papers/hyvonen/ hyvonen.html.
18. E. Hyvönen, M. Salminen, S. Kettula, and M. Junnila. A content creation process for the Semantic Web, 2004. Proceeding of OntoLex 2004: Ontologies and Lexical Resources in Distributed Environments, May 29, Lisbon, Portugal (forthcoming).