Management of dependency between two or more ontologies in an environment for distributed development

Eiichi Sunagawa, Kouji Kozaki, Yoshinobu Kitamura, Riichiro Mizoguchi

The Institute of Scientific and Industrial Research, Osaka University
8-1 Mihogaoka, Ibaraki, Osaka, 567 -0047 Japan
Tel: +81-6-6879-8416, Fax: +81-6-6879-2123
E-mail: {sunagawa, kozaki, kita, miz} @ei.sanken.osaka-u.ac.jp

Abstract

This paper describes the management of the dependency between component ontologies in an ontology as a whole and its use for collaborative development of the ontology. It is necessary for collaborative development of an ontology to manage the influence of modification of ontologies to others. The dependency is investigated to see how many types exist and how to manage each of them. Functions to manage such dependency are also designed for supporting modification of the ontology caused by the modification of other ontologies which influences on the dependency. These functions make it easer to develop an ontology collaboratively and contribute to reusing ontologies.

1. Introduction

In general, an ontology can be divided into several component ontologies. Occasionally, building an ontology is done collaboratively in which case component ontologies are built and then they are compiled into a unified ontology. These component ontologies are identified according to their conceptual level or domains.

For example, Fig.1 shows "Plant Ontology", which was built on Human Media Project under the former Ministry of International Trade and Industry [Mizoguchi 00]. This ontology is separated into three parts: "Top Level Ontology", "Task Ontology" and "Domain Ontology". Furthermore, the domain ontology is divided into two ontologies: physical attribute and equipment. "Equipment Ontology" is further divided into ontologies of objects, plant parts and function. In Fig.1, arrows express the relation between an upper ontology and a lower ontology. This is named "*Super-sub* Relation" (discussed in section 2.1). Development of ontology as a whole is achieved by editing and modification of its component ontologies. Management of their relations and explicit control of influence propagation caused by the change in each components ontology contributes to realization of such a collaborative development of an ontology.

Hereinafter, section 2 discusses definitions of the dependency between ontologies and the method to keep the consistency of the dependency. Section 3 describes implementation of the proposed methods in Hozo followed by concluding remarks.

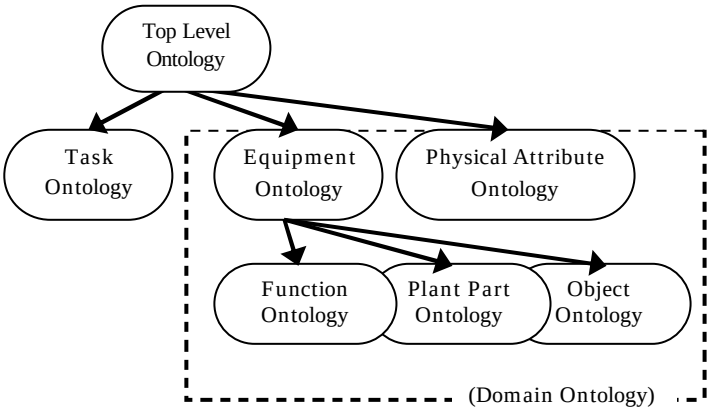

Fig.1 Plant Ontology

2. Dependency between ontologies and its management

We have argued the heavy part of ontology such as the role concept in [Kozaki 00]. The dependency this paper discusses is based on *is-a* relation and class constraint. As these relations can be treated in RDF(S) or OWL, our research will contribute to the development of ontologies for the Semantic Web.

2.1 Dependency between ontologies

When constructing an ontology, concepts are usually defined with reference to the definitions of other concepts. In collaborative construction, those referred concepts might exist in another ontology developed by another person. That means some concepts in an ontology depend on other concepts in another ontology. This section discusses the dependency between ontologies which is defined as in terms of the dependency between concepts defined in respective ontologies. The kinds of them are:

1) *Super-Sub* Relation (*is-a* relation)
Two ontologies are said to be in "*super-sub* relation", if and only if there are at least two concepts in *is-a* relation and each of the two concepts belongs to a different ontology of the two. We named these ontologies "upper ontology" and "lower ontology" respectively. The lower ontology depends on the upper one at the point of inheriting definition. In the "Plant Ontology"(Fig.1), we can find this relation between "Top Level Ontology" and "Equipment Ontology", between "Equipment Ontology" and "Plant Parts Ontology", etc.

2) *Referring-to* Relation (class constraint)
We define "*referring-to* relation" as the relation that a concept in one ontology refers to a concept in another as a class constraint. We named the ontology containing the slot being constrained "referring ontology" and the other "referred-to ontology". In the "Plant Ontology" (Fig.1), we can find this relation between "Plant Parts Ontology" and "Physical Attribute Ontology" etc.

To manage the dependencies, each component ontology has the information about them;

- a copy of the definition of the concept it depends on
- the name and the version of the ontology it depends on

Section 3 describes how to use this information.

2.2 Managing dependency between ontologies

2.2.1 Keeping consistency of the dependency

When editing an ontology, we should pay attention to the influence the change on other ontologies. In some cases, a change may destroy the consistency between ontologies. We investigated two approaches to keep consistency of the dependency. One is to prohibit simply the change which influences on others. The other is to modify the influenced ontology according to the type of the change. This paper is mainly concerned with the latter approach. 5 kinds of countermeasures taken in the influenced ontology are

- **To accept the change;**
  - **1-1) To modify influenced ontology for accepting the change;** The user makes agreement on the change of the ontology and tries to modify his/her ontology depending on it. The influenced ontology needs to be modified to adapt to the changed ontology. The way to reflect the change of the influencing ontology is mentioned later.
  - **1-2) To leave the depending ontology influenced by the change;** In some cases, the influenced ontology is not need to be modified, as the changed ontology doesn't contradict it.
- **To reject the change;** When the user does not agree on the change, his/her ontology depending on it should be modified in order to get rid of the possible contradictions at least in itself.
  - **To keep the dependency;**
    - **2-1) To modify influenced ontology for rejecting the change;** As far as keeping the consistency of the dependency, the user tries to modify his/her ontology against the change and reduce the influence. The way to negate the influence of the change is mentioned later.
    - **2-2) To stay compliant with the last version of the changed (depending) ontology;** Under controlling the version of ontologies, the dependency is kept in this way. If influencing ontology would be changed again, influenced one could adapt to the change and the consistency would be recovered.
  - **2-3) To break the dependency;** In order to make the influenced ontology independent of the others, concepts whose change influences on it are imported in it and cut the link of the dependency between the two.

In *Super-sub* Relation
- Operation of a concept
  - Deletion a concept
  - Addition a sub concept
- Change of the definition
  - Change of the label (name)
  - Deletion of a slot
    (a partial or an attribute concept)
  - Addition of a slot
    (a partial or an attribute concept)
  - Change of the class constraint *(inheriting)*
    - ✓ Generalizing
    - ✓ Specializing
    - ✓ Change to a completely different concept
  - Change of the class constraint *(overridden)*
    - ✓ Generalizing
    - ✓ Specializing
    - ✓ Change to a completely different concept

In *Referring-to* Relation
- Operation of a concept
  - Deletion of a concept
  - Addition of a sub concept
- Change of the definition
  - Change of the label (name)
  - Deletion of a slot
    (referred by a role concept)
  - Deletion of a slot
    (not referred by a role concept)
  - Addition of a slot

Table.1 Types of the change of the concept

In either case of accepting or not accepting, modification of the influenced ontology should be supported because of its complexity. So, we began with conceiving the patterns of the change. And, for the influence of each pattern, we investigated the possible way of modification to keep the dependency. The influenced ontology is modified based on this framework.

We have two major kinds of patterns of the change: operation on the concept itself and changing its definition. The former includes the cases where a concept has been deleted or a sub concept has been added. The latter does the cases where the label has been changed, a slot such as a part of or an attribute of a concept has been deleted, added or a class constraint has been changed. In all, we have 17 types of the change of the concept according to the kind of dependency (Table.1). And, as the countermeasures for the change, we have 32 ways of modification in all.

In addition, the same situation also appears in a single ontology that the concept is influenced by the change of other concepts. In this paper, we consider especially the influence between ontologies because support is more required for the case of inter-ontologies than for the case of intra-ontology.

2.2.2 Example of modifying an ontology to keep the consistency of its dependency

In this section, we show two examples of the dependency management in "Plant Ontology" of Human Media Project (Fig.1).

Ex.1: Fig.2 shows a portion of "Plant Ontology" (in Fig.1). "*Heat Exchanging Device*" is sub-concept of (*is-a*) "*Device*". "*Driving Device*" and "*Info Device*" are so, too. Then, we can define *super-sub* relation between "Equipment Ontology" and "Plant part ontology". Assume that the slot "*Input Thing*" has been deleted from the concept "*Device*" in "Equipment Ontology". The change influences "Plant Part Ontology".
To cope with the change such as "Deletion of a slot in *Super-sub* Relation", four ways are supported. The developer of "Plant Part Ontology" can select a countermeasure out the following four:

1-2) To do nothing (to accept the change)
Deletion of "*Input Thing*" is applied to all influenced concepts in "Plant Part Ontology". (In the case of this example, it is thought that manual change is needed because of importance of the deleted definition.)
2-1) To add the same as deleted slot to a depending concept in the lower ontology (to reject the change)
To reject the deletion of "*Input Thing*" in "Plant Part Ontology", the slot should be added to some concepts which are overriding it.

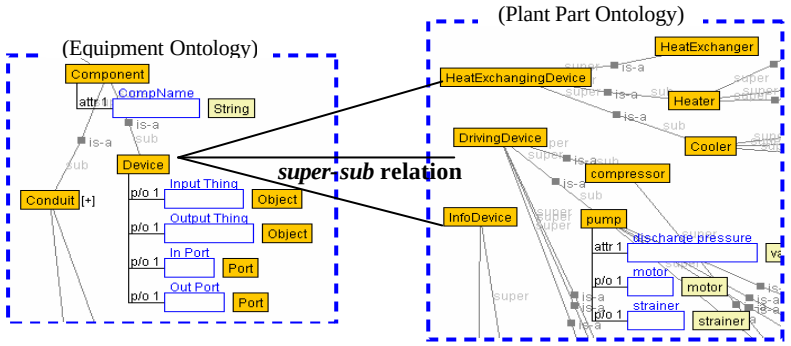

Fig.2 An example of *super-sub* relation

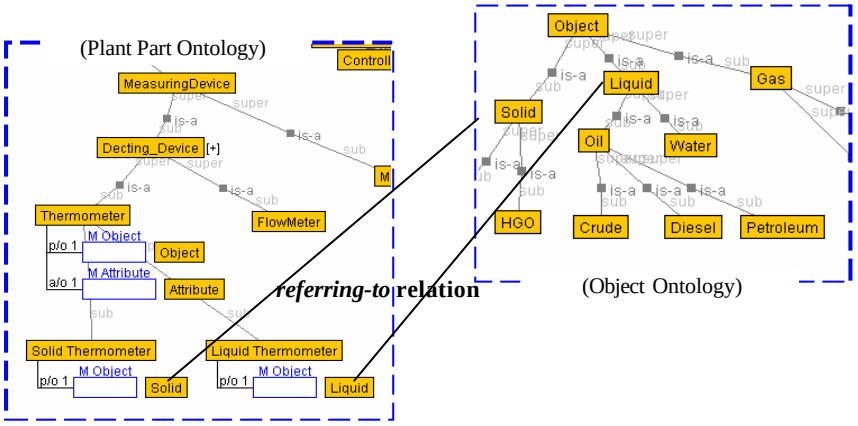

Fig.3 An example of *referring-to* relation

2-2) To stay compliant with the last version of the modified ontology (to reject the change)
The old version of "Equipment Ontology" has been saved in the ontology server (described in section 3.1). "Plant Part Ontology" can keep dependence on it under the version control.

2-3) to break the dependency (to reject the change)
Redefine *"Device"* with *"Input Thing"* in "Plant Part Ontology" and break the dependency between the ontologies. "Plant Part Ontology" is then changed to be independent of "Equipment Ontology".
It looks similar to 2-1). Superficially, it is the same. But, 2-3) breaks the dependency, while 2-1) keeps the dependency by copying with the current difficulty. "*Input Thing*" is added to all the concepts which need it as "*Heat Exchange Device*".

Ex.2: Fig.3 shows part of "Plant Ontology" (in Fig.1). "*Liquid Thermometer*" in "Plant Part Ontology" is referring to "*Liquid*" in "Object Ontology" as a class constraint of "*M_Object*". Then, we can define *referring-to* relation between these ontologies. Assume that the concept "*Liquid*" has been deleted from "Object Ontology". It influences "Plant Part Ontology".
To cope with the change such as "Deletion of a concept in *Referring-to* Relation", four ways are supported. The developer of "Plant Part Ontology" can select a countermeasure out the following four:

1-1) To refer a super concept of the deleted concept (to accept the change)
As the class constraint of "*Liquid Thermometer*", we can refer "*Object*" which is the super concept of "*Liquid*". This means the class constraint to "*Measurement Attribute*" become looser a little.

2-1) To add the same as the deleted concept to the referring ontology (to reject the change)
This way means the deletion of "*Liquid*" is denied in "Plant Part Ontology". The author redefines "*Liquid*" in "Plant Part Ontology", and establishes newly *super-sub* relation between "Plant Part Ontology" and "Object Ontology" through *is-a* relation between "*Liquid*" and "*Object*". (However, this method should be temporary adjustment. Because it is not desirable that only one concept, which is a "*Object*", is defined in the different ontology from "Object Ontology", in which the other concepts of "*Object*" are defined.)

2-2) To stay compliant with the last version of the modified ontology (to reject the change)
It is the same as Ex.1.

2-3) To break the dependency (to reject the change)
It is the same as Ex.1.

3. Distributed development with "Hozo"

On the basis of investigation described above sections, we designed the functions to manage the dependency between ontologies and to keep its consistency. And we have implemented these functions as a sub system of our ontology development system, "Hozo". The extension provides more effective collaborative development for user.

3.1 "Hozo", an environment for building ontologies

We have developed an environment, named "Hozo" [Kozaki 00, Kozaki 02], for building ontologies based on fundamental ontological theories. "Hozo" is composed of "Ontology Editor", "Onto-Studio" and "Ontology Server" (Fig.4). The ontology editor provides users with a graphical interface, through which they can browse and modify ontologies by simple mouse operations. This system manages properties between concepts in the *is-a* hierarchy. The Onto-Studio is based on a method of building ontologies, named AFM (Activity-First Method) [Mizoguchi 95], and it helps users design an ontology from technical documents. The ontology server manages the built ontologies and models.

Because the architecture is implemented in Java and the ontology editor is an applet, it can work as a client through Internet. Hozo manages ontologies and models considering who is its developer. For each ontologies in Hozo, its author can define and modify it, and the other users can only read and copy it. It lets share ontologies among users without explicit version control.

Models are built by choosing and instantiating concepts in the ontology and by connecting the instances. Hozo also checks the consistency of the model using the axioms defined in the ontology. The ontology and the resulting model are available in different formats (Lisp, Text, and XML/DTD) that make it portable and reusable.

3.2 Implementing the function for managing dependencies between ontologies

To management the dependency described in section 2.1, we use the information in each ontology about the dependency it has.

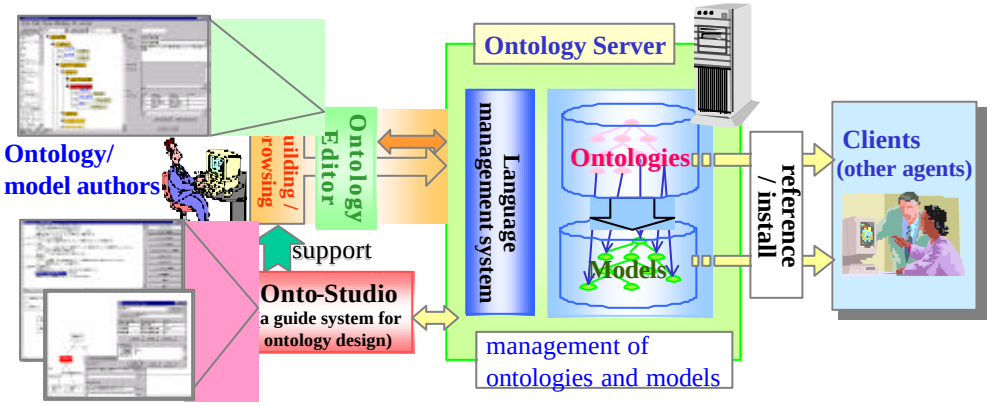

Fig.4 the architecture of Hozo

3.2.1 The function for managing dependencies between ontologies

We have designed a tool, named "Ontology Manager", for managing dependencies between ontologies. Fig.5 shows its interface. Ontology Manager consists of 4 panels: "Ontology List", "Ontology Viewer", "Ontology Information Panel" and "Dependency Panel". These panels are to show users a series of information about ontologies build with Hozo.

- **Ontology List** shows a list of ontologies which is registered in Ontology Server. Users can select an ontology, and the information about it is shown in other panels.
- **Ontology Viewer** shows dependencies graphically by using nodes and links which each of them represent an ontology and *super-sub* relation, respectively. Users can grasp easily the outline of dependencies.
- **Ontology Information Panel** shows the name, file name, author, version, last update of the selected ontology.
- **Dependency Panel** shows the lists of ontologies which have a dependency with the selected ontology. They are classified by their types. In section 2.1, we defined 4 types: upper, lower, referring and referred-to. Users can select any of them by tabs. The table informs users of names of ontologies, concepts which constitute a dependency, version of ontologies and whether that concept is changed or not. They are necessary to support modification to cope with changes.

3.2.2 The function for supporting modification to cope with the change

When the user is going to edit an ontology and select it on Ontology Manager, this system checks the change of the ontologies it depends on. And, Ontology Manager shows him/her which ontology has been changed and might destroy the consistency of its dependency. The user can see that the color of the node of changed ontology is turning in Ontology Viewer and the changed concept is checked in Dependency Panel.

Next, to keep the consistency of

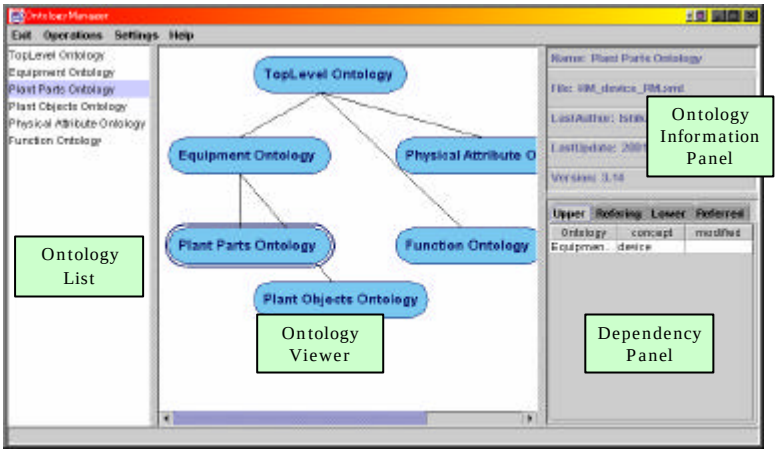

Fig.5 Ontology Manager

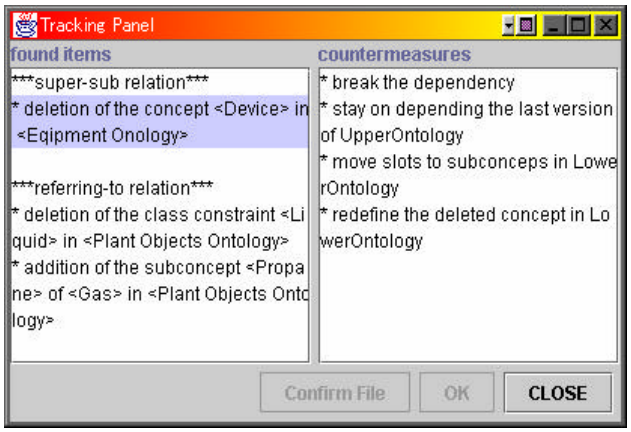

Fig.6 Tracking Panel

dependency, the user should get more information that how the influencing ontology has been changed and what countermeasures are supported. These are shown in the panel, named "**Tracking Panel**" (Fig.6). It lists the change of the influencing ontology and the possible countermeasures for coping with each change. The user selects the change of the ontology and the countermeasure for it. Then his/her ontology is modified semi automatically and the dependency is kept its consistency.
This function is used when the user opens the influenced ontology to edit it and whenever he/she requests the change information of other ontologies..

Checking the change of influencing ontology is achieved by a comparison between the definition of depended concepts and its copy the influenced ontology has (described in section 2.1). If the consistency of dependency may be broken, the system lists the kind of detected change and countermeasures for it based on the patterns described in 2.2.1.

4. Related Work

Some other ontology building tools have been developed in the similar way to us. While OntoEdit [Sure 02] allow multiple users and control their access to the same ontology to develop it collaboratively, we do not allow such operation. Instead, we allow users to divide an ontology into several component ontologies and to manage the dependency between them. The Karlsruhe Ontology and Semantic Web framework (KAON) has developed an application, which resembles our system [Ljubljana 02]. They have investigated a dependency management similar to ours mainly for supporting evolution of ontology. From the view point of collaborative development of ontologies, our system has types of the change of an ontology and tracking it for providing sophisticated help.

5. Conclusion

In this paper, we discussed the management of dependency between ontologies to facilitate collaborative development of an ontology. By using this system, we can manage the dependency and keep the consistency for some patterns of modification of ontology. The prototype of this system has been implemented. Further, we understand this system needs improvement.

- Other relations between ontologies;

We have dealt two types of dependency between ontologies such as "*super-sub*" relation and "*referring-to*" relation. Now we can see other kinds of relations, which are based on content of the ontology; such as a task-domain relation, a role concept - a basic concept relation and so on. It may be useful for supporting the development of ontology to accommodate users with a framework to manage content relations.

- Support for agreement making between users;

Agreements are indispensable for building ontology. Especially for accepting the modification of the

ontology, it is important for authors to know the intention of it. However, in general, it is difficult to communicate in a distributed development. To support communication between users make development ontology more easily.

● Division and integrate of ontology;
On mentioning "Plant Ontology", we didn't explain why it can be divided so. It is true that the component ontologies are identified according to their conceptual level or domains, but we didn't discuss how to divide and integrate ontology in this paper. To make it clear the border of component ontologies is useful for dividing, integrating and also reusing an ontology.

6. References

[Kozaki 00] Kouji Kozaki, Yoshinobu Kitamura, Mitsuru Ikeda, and Riichiro Mizoguchi: Development of an Environment for Building Ontologies Which Is Based on a Fundamental Consideration of "Relationship" and "Role", Proc. of the Sixth Pacific Knowledge Acquisition Workshop (PKAW2000), pp.205-221, Sydney, Australia, December 11-13, 2000

[Kozaki 02] Kouji Kozaki, Yoshinobu Kitamura, Mitsuru Ikeda, and Riichiro Mizoguchi: Hozo: An Environment for Building/Using Ontologies Based on a Fundamental Consideration of Role" and "Relationship", Proc. of the 13th International Conference Knowledge Engineering and Knowledge Management (EKAW2002), pp.213-218, Sigüenza, Spain, October 1-4, 2002

[Ljiljana 02] Stojanovic Ljiljana, Maedche Alexander, Motik Boris, Stojanovic Nenad: User-driven Ontology Evolution Management, Proc. of the 13th International Conference Knowledge Engineering and Knowledge Management (EKAW2002), pp.213-218, Sigüenza, Spain, October 1-4, 2002

[Mizoguchi 95] R. Mizoguchi, M. Ikeda, K. Seta, and V. Johan: Ontology for Modeling the World from Problem Solving Perspectives, Proc. of IJCAI-95 Workshop on Basic Ontological Issues in Knowledge Sharing, pp. 1-12, 1995.

[Mizoguchi 00] Mizoguchi, R., Kozaki, K., Sano, T., and Kitamura, Y.: Construction and Deployment of a Plant Ontology, Proc. of the 12th International Conference Knowledge Engineering and Knowledge Management (EKAW2000), pp.113-128, Juan-les-Pins, France, October 2-6, 2000

[Sure 02] York Sure, Michael Erdmann, Juergen Angele, Steffen Staab, Rudi Studer, and Dirk Wenke: OntoEdit: Collaborative Ontology Development for the Semantic Web, Proc. of the First International Semantic Web Conference (ISWC2002), Sardinia, Italy, June 9-12, 2002.