

# An Environment for Distributed Ontology Development Based on Dependency Management

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  
{sunagawa, kozaki, kita, miz} @ei.sanken.osaka-u.ac.jp  
<http://www.ei.sanken.osaka-u.ac.jp/>

**Abstract.** This paper describes a system for supporting development of ontology in a distributed manner. By a distributed manner, we mean ontology is divided into several component ontologies, which are developed by different developers in a distributed environment. The target ontology is obtained by compiling the component ontologies. These component ontologies are identified according to their conceptual level or domain characteristics. The distributed development of ontologies applies to many situations such as cooperative development, reusing ontologies and so on. To support such a way of ontology development, we investigate the dependency between component ontologies and design some functions for management of these ontologies based on their dependencies. We next consider the influence of a change of one ontology to others through its dependencies and design a function to suggest a few candidate modifications of the influenced ontology for keeping the consistency. We also present some examples of how the system works.

## 1 Introduction

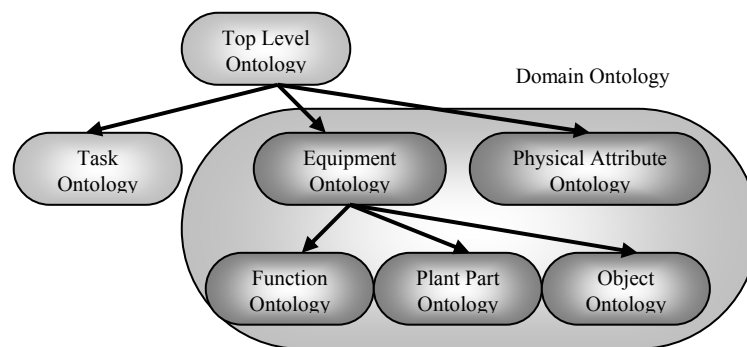
The more Semantic Web attracts attention, the more importance of ontology increases. In the Semantic Web, ontologies are developed by different developers in a distributed environment. So, Distributed Ontology Development is one of the most significant issues.

In general, ontology can be divided into several component ontologies. Building ontology means occasionally building portions and compiling them. These component ontologies are identified according to their conceptual level or domains.

For example, Fig.1 shows “Plant Ontology”, which was built in the Human Media Project sponsored by the former Ministry of International Trade and Industry [1]. 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 individually. We call development of ontology in this manner “Distributed Ontology Development”, and we aim to develop a system supporting it through our research. To support such a way of ontology development, we have been investigating the dependency management between component ontologies. We consider how a change of one ontology influences on others through its dependencies and what countermeasures are effective for the change in order to keep the consistency of them [2].

In this paper, we discuss more details of distributed ontology development. Section 2 discusses its basic philosophy and summarizes our work to date. Section 3 describes implementation of the proposed methods in Hozo [3, 4] followed by concluding remarks. In section 4, we discuss the future work.



**Fig. 1.** Plant Ontology. It can be divided into several ontologies according to their conceptual levels or domains

## 2 Distributed Ontology Development

The purpose of our research is to realize a distributed ontology development. We assume a situation where target ontology is divided into several component ontologies and to construct each ontology individually (perhaps in parallel) by different developers in a distributed environment.

The distributed development of ontologies applies to many situations such as cooperative development, understanding the total picture of conceptual hierarchy, reusing ontologies and so on. For example, a developer would divide one ontology by categorizing concepts roughly, and then he/she builds each component ontologies in cooperation with other developers. He/she can divide in such a manner before and during the course of construction. In another case, he/she would carve out a constructed ontology for reusing it as a part of another one. In these cases, we can assume that such component ontologies are valuable on every phase of ontology development.

In this section, we argue requirements on developing ontology in such a distributed manner. We especially note how to keep the consistency of component ontologies which are constructed individually and we define the dependency between ontologies.

## 2.1 Dependencies between Ontologies and Their Management

In this paper, we treat basic concepts mainly syntactically or formally while we have argued a part of heavy weight ontology such as the role concept in [3]. 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.

**Dependency between Ontologies.** When constructing 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 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.

These 4 types of component ontology based on dependencies, “lower”, “upper”, “referring” and “referred”, are determined on the basis of conceptual dependency with each other. So, an ontology does not have its own type intrinsically. If there are several concept pairs making dependency between two ontologies, we can define several numbers (and kinds) of dependencies and each ontology takes multiple positions. In such a case, ontologies and their dependencies form a graph (rather than a tree) as a whole.

**Management of Dependency between Ontologies.** When editing ontology, we should pay attention to the change influencing on other ontologies. In some cases, that change may destroy the consistency between ontologies. We investigated two approaches to keep consistency of the dependency. One is to restrict the change which influences on others. The proactive restriction helps developers to avoid inconsistency. The other approach is to modify the influenced ontology according to the type of the change. This paper is mainly concerned with the latter approach, and the

former is argued in section 4 as remaining work. 5 kinds of countermeasures taken in the influenced ontology are:

- **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.
- **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. However, this should be a temporal method to keep the dependency. Its problem is argued in section 4.
- **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.

1-1) and 1-2) are selected when the author agrees on the change and accept its influence on his/her ontology. 2-1), 2-2) and 3) are to reject the change. Then, he/she is able to deny the change influencing his/her ontology at least in itself. All but 3) of these countermeasures are selected in order to keep the dependency.

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. And, as the countermeasures for the change, we have 67 ways of modification. Table.1 shows what type of the change are supported and how many countermeasures for each type are supported. We cannot always take all kinds of countermeasures. Some types of the change influence strongly and restrict the choices of authors.

**Table 1.** The patterns of the change and countermeasure for them

patterns of the change	countermeasures for the change				
	accept the change		reject the change		
	keep the dependency			break the dependency	
	modify	leave	modify		
1-1	1-2	2-1	2-1	3	
super-sub relation					
operation of a concept					
deletion of a concept	1	-	1	1	1
addition of a sub concept	2	1	-	1	1
change of the definition					
change of the label	1	1	-	1	1
deletion of a slot	1	-	1	1	1
addition of a slot	2	1	-	1	1
change of the class constraint (inheriting)					
generalizing	1	1	-	1	1
specializing	-	1	-	1	1
different concept	-	1	-	1	1
change of the class constraint (overridden)					
generalizing	-	1	-	1	1
specializing	2	-	-	1	1
different concept	2	-	-	1	1
referring-to relation					
operation of a concept					
deletion of a concept	1	-	1	1	1
addition of a sub concept	1	1	-	1	1
change of the definition					
change of the label	1	1	-	1	1
deletion of a slot (referred by a role concept)	1	-	1	1	1
deletion of a slot (not referred by a role concept)	-	1	-	1	1
addition of a slot	2	1	-	1	1

**Examples of Modifying ontology to Keep the Consistency of Its Dependency.** In this section, we show two examples of the dependency management in “Plant Ontology” (described in section 1).

**Ex.1:** Fig.2 shows a portion of “Plant Ontology”. “*Heat Exchanging Device*” is sub-concept of (*is-a*) “*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*”. That change influences “*Plant Part Ontology*”.

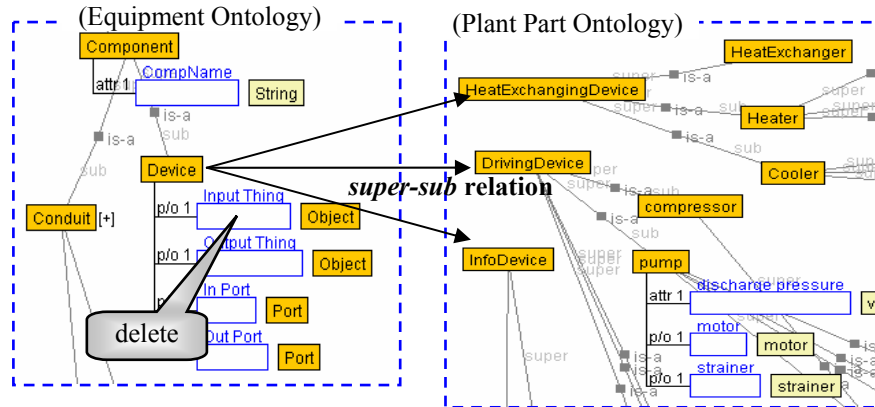


Fig. 2. An example of *super-sub* relation

According to Table.1, four ways are supported to cope with the change of “Deletion of a slot in *Super-sub* Relation”. The developer of “*Plant Part Ontology*” can select a countermeasure out the followings:

- **1-2) To delete the slot in all influenced concepts (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 appropriate concepts which are inheriting it. (In this example, the slot “*Input thing*” is inherited by “*Heat Exchanging Device*”, “*Driving Device*” and “*Info Device*”. Then, we should add the slot to them.)
- **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.
- **3) To break the dependency (to reject the change):** Re-define “*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*”.

**Ex.2:** Fig.4 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*”.

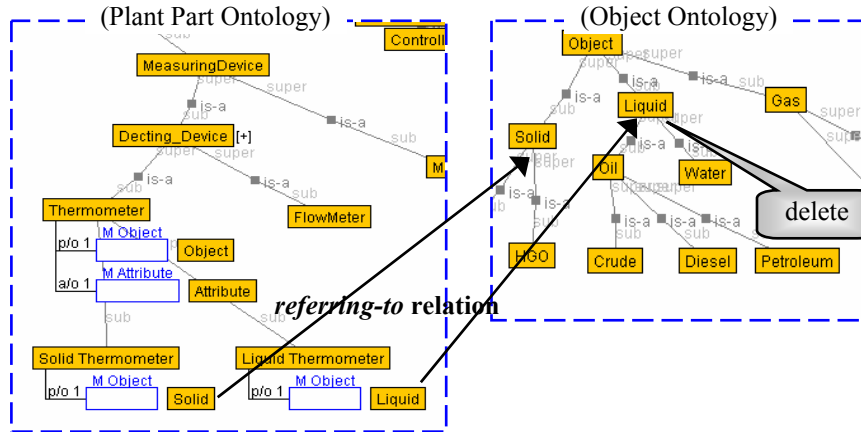


Fig. 3. An example of *referring-to* relation

According to Table.1, four ways are supported to cope with the change such as “Deletion of a concept in *Referring-to* Relation”. The developer of “Plant Part Ontology” can select a countermeasure out the followings:

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

## 2.2 Other Contentions

The management of dependencies between ontologies is not enough for realizing distributed ontology development smoothly because dependency is one of the aspects which appear particularly in the consistency management of conceptual hierarchy. So, we have to consider more general functions for supporting distributed development. In this section, we discuss the management of ontology from the point of *version control, updating and reusing*. Next, the management of developers is discussed together with cooperative development. Here, we need to note that component ontologies are managed together with their developers by each target ontology as a whole.

## **Management of Ontologies.**

*Version Control of Ontologies.* In distributed ontology development, each component ontology is updated independently. To manage the dependency between them with our method described in section 2.1, the system have to preserve old versions of ontologies.

When under construction, it is not hard to control the version of ontologies. An old version of ontology needs to be preserved in the system only if it has dependencies with some component ontologies. So, our system manages dependencies with information about the version of ontology which has the dependency. And, we can manage them more easily if ontology avoids having dependency with an old version of other ontology unnecessarily.

However, it is very hard to manage dependencies if many versions of the same ontology have dependencies with other ontologies individually. It causes a problem especially in a stage of compiling component ontologies to the target ontology. This is discussed in section 4.

*Update of Ontologies.* In our distributed ontology development system, its user constructs his/her own component ontologies on a local computer. And when he/she decides to publish them, the ontology is updated on a shared space of server computer. At the same time, the user can access to other published ontologies which other developers have developed, whenever he/she needs to check the dependency of his/her ontology. If it is needed to be refined, he/she modifies his/her ontology to cope with the change of ontologies it depends on. Then, he/she updates the modified ontology again.

*Reuse of Ontologies.* In our system published component ontologies are reused as some part of other ontologies. We support 3 types of reusing ontologies as follows

- 1) reusing ontology which is under construction
- 2) using a particular version of ontology
- 3) importing some version of ontology, and arranging it

In general, ontology reused should be of the final version because it is hard to develop ontology with keeping its consistency in several different target ontologies. For this reason 1) is not good way to reuse ontologies. On the other hands, if the developer uses a particular version of ontology, as 2), and ignores its evolution later on, the reuse of ontology becomes easier. 3) is an evolved case of 2) , and it is supposed the most practical way. In this case, arranged ontologies should be regarded not as the updated version of the imported ontology. To simplify the management of ontologies, it is regarded as a newly developed ontology rather than a new version of the original ontology. And its versions should be managed separately.

At any types, to reuse ontologies we should consider not only computer processing (e.g. management of consistency of the conceptual hierarchy) but also communication and agreement among the developers. To support reuse of ontology in the distributed ontology development, we will investigate more these topics.

**Cooperative Development and User Management.** It is very hard to manage the consistency of dependencies if several developers construct the same part of ontology at the same time. For this reason, our system does not allow multiple accesses to a concept by different developers at the same time.

When developers edit their ontology, the system manages and shows the information they need. For example, which ontologies are related it, how their relations are, the developer of related ontologies, and so on. Because the methodology of ontology development is still argued and outside of this paper, we do not describe detail of how this information support developers in cooperative development. But, at least, we can say that the information will be used in order to support communication between developers because it is difficult for them to make agreement about target ontology in a distributed manner.

However, we still have some issues to consider. Firstly, our approach might become unnecessarily complex when many developers divide a large ontology into many component ontologies and construct them in parallel. One solution to this problem is to control the access of developers at a conceptual level in ontologies. We will investigate it in the future work. Secondly, we can take some cases in which several developers would like to construct the same component ontology of target ontology. This issue is mentioned together with the construction of component ontologies in section 4.

### **3 Distributed Ontology Development with “Hozo”**

On the basis of our consideration described in the above section, we designed two tools for supporting to practice distributed ontology development. First, we summarize our system “Hozo” developed as an environment for building ontologies. Next, we describe how distributed ontology development is realized in “Hozo” and how two tools works for supporting developers.

#### **3.1 “Hozo”, an Environment for Building Ontologies**

We have developed an environment, named “Hozo” [3, 4], for building ontologies based on fundamental ontological theories. Hozo is composed of “Ontology Editor”, “Onto-Studio” and “Ontology Server” (in Fig.4). Ontology Editor provides users with a graphical interface, through which they can browse and modify ontologies. This system manages properties between concepts in the *is-a* hierarchy. Onto-Studio is based on a method of building ontologies, named AFM (Activity-First Method) [5], and it helps users design ontology from technical documents. 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 its developer is. 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.

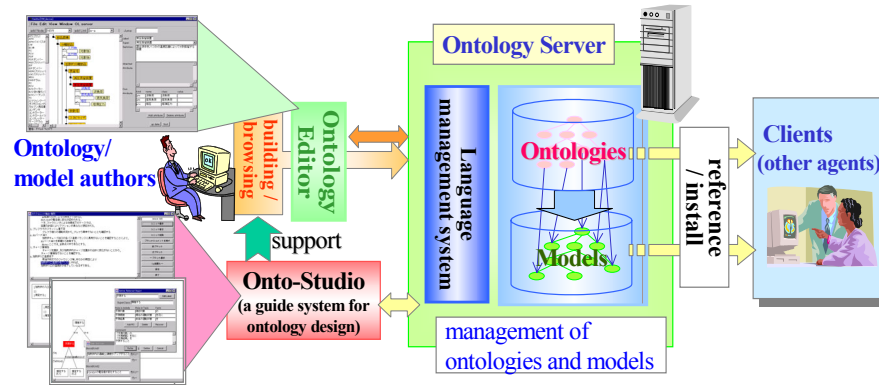


Fig. 4. HoZo, an Environment for Building Ontologies

### 3.2 Practice of Distributed Ontology Development

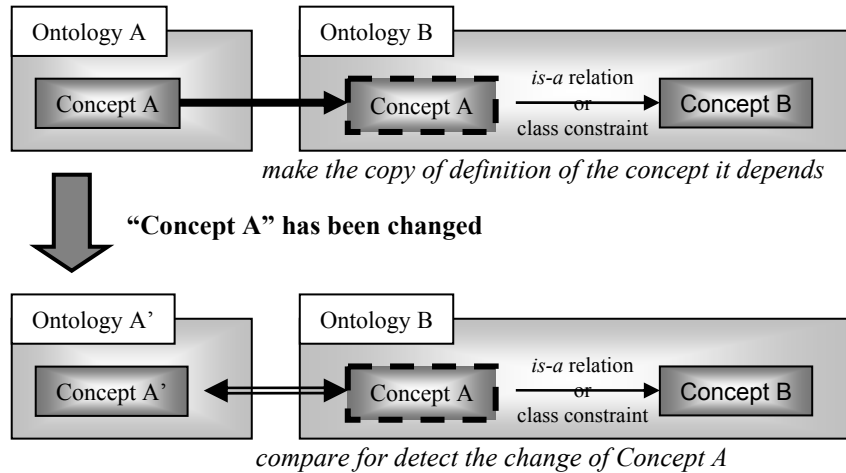
**Flow of Distributed Ontology Development.** Distributed ontology development is performed as described in section 2.2. The development has been done by the repetition of the following steps; **1)** a developer logs in HoZo and runs Ontology Manager (described in section 3.3). Then, he/she can get a total picture of the target ontology and information of dependencies between component ontologies. One of the important information is which ontology has been changed. **2)** he/she selects ontology to edit and open it by Ontology Editor. At the same time, Ontology Editor accesses other component ontologies it depends on and compiles them temporarily. At this phase, a concept hierarchy is built up as one ontology (e.g. inheritance of the definition). Then, its dependencies are checked automatically and he/she knows their conditions. If their consistencies may be broken, he/she can select countermeasures listed on Tracking Panel (described in section 3.4) to cope with the change. **3)** he/she starts editing his/her ontology. In addition, dependency is checked whenever he/she needs it is under editing. **4)** After editing, he/she updates his/her ontology on Ontology Server and publishes it to others.

**Data Structure and Its Use.** To manage the dependencies, the system manages the information about each component ontology as follows:

- its name, its version, its developer and the last update time of itself
- the name and the version of ontologies it depends on
- a copy data of the definition of the concept it depends on in other ontology

A copy data of the definition is used to check the consistency of dependency and to identify the type of change of ontology (in Fig. 5). The copy data is mounted in ontology when its developer makes or rebuilds a dependency. Our system checks the change of influencing ontology by comparing the definition of depended concepts with its copy data the influenced ontology has. If the consistency of dependency may

be broken, the system lists the kind of detected changes and countermeasures to keep consistency of the dependency based on the patterns in Table.1.



**Fig. 5.** Data Structure of dependency. When ontology B depends on concept A in ontology A, the system make the copy of definition of concept A in ontology B. This copy is used for detecting and identifying of the change of concept A

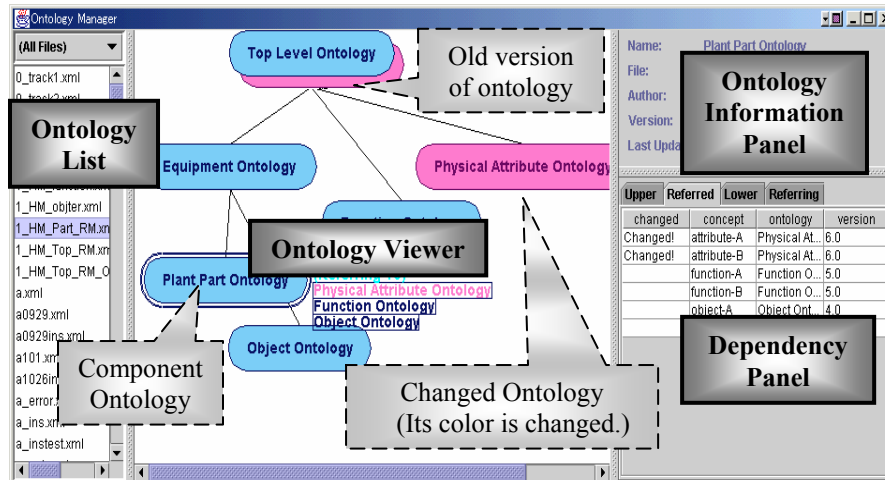
**Operations for Distributed Ontology Development.** Ontology Manager provides four operations for distributed ontology development; to create new component ontology, to divide a component ontology, to compile ontologies and to reuse ontology as a component. These are available mainly in the case where a developer specifies component ontologies and their dependencies before constructing every component ontology and then participants start development according to the specification. On the other hand, we can assume a case where a developer constructs each component ontology before its borderline and dependency is defined. In such a case, he/she has to make a dependency on occasions. Ontology Editor provides the functions to find a concept in other ontologies and make a dependency with it.

### 3.3 Ontology Manager

We have designed a tool, named "Ontology Manager". Fig.6 shows its interface. Ontology Manager consists of 4 panels:

- **Ontology List** shows a list of ontologies which is registered in Ontology Server. Users can select ontology, and then the information about it is shown in other panels.
- **Ontology Viewer** shows dependencies between ontologies graphically by using nodes and links each of which represents ontology and *super-sub* relation, respectively.
- **Ontology Information Panel** shows the name, file name, developer, version, last update of the selected ontology.

- **Dependency Panel** shows the list of ontologies which have a dependency with the selected ontology. They are classified in 4 types (described in section 2.1): *upper*, *lower*, *referring-to* and *referred-to*. Users can select shown type by tabs. The table informs users the names of ontologies, concepts which constitute the dependency, version of ontologies and whether that concept is changed or not.

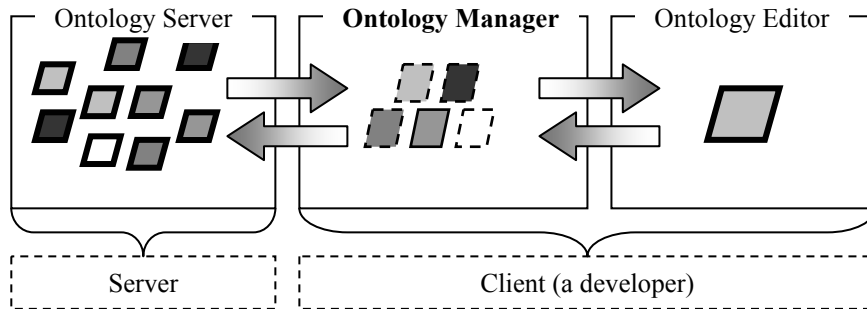


**Fig. 6.** Ontology Manager. It consists of 4 panels: Ontology List, Ontology Viewer, Ontology Information Panel and Dependency Panel

These panels are to show users a series of information about ontologies built by Hozo. Besides, Ontology Manager acts as a bridge between ontologies edited in local and ones open to the public (Fig.7). Furthermore, Ontology Manager carries out 3 functions as follows.

- **Management of Dependency;** In both local and public cases, only component ontologies exist and developers edit them individually. Ontology Manager compiles them virtually to form target ontology and shows its configuration on Ontology Viewer. It enables developers to grasp easily the outline of dependencies. And, if developers need to get details about dependencies, they can use Dependency Panel. In addition, this function is used as a first step for supporting modification to cope with changes as described in section 2.1. More details are described in section 3.4.
- **Version Control;** To manage dependency between ontologies, Ontology Manager treats their versions also. If some ontology depends on an old version of another, the caution is given graphically on Ontology Viewer. These old versions of ontologies are preserved in Ontology Server, if they have been open to the public. Ontology Manager searches Ontology Server and find necessary version of the ontology.
- **Management of Developers and Their Access;** Ontology Server has information about every target ontology together with its developer. The system manages the developer's access to ontologies by considering which target ontology

he/she joins to construct and which component ontologies he/she may edit. A developer cannot edit every component ontology. As described in section 2.2, we assume that one component ontology is built by one developer. However, if a developer gives permission, the system allows another to edit his/her ontology. In this case, unless he/she admit, the edited ontology cannot be updated.



**Fig. 7.** Roles of Ontology Manager. It acts as a bridge between Ontology Editor and Ontology Server. Its roles are to: (1) download component ontologies of a target ontology from Ontology Server, (2) propagate the influences of the change to the ontology which is edited with Ontology Editor (3) upload the component ontology, which is edited in local, to Ontology Server

### 3.4 Tracking Panel: the function to keep the consistency of the dependency

Ontology Manager shows developers which ontology has been changed and might destroy the consistency of its dependency. To keep the consistency of dependency, the developer should get more information that how the influencing ontology has been changed and what countermeasures are supported. These are shown in Tracking Panel. The panel lists the change of the influencing ontology and the possible countermeasures for coping with each change. He/she selects the change of the ontology and the countermeasure form the list. Then his/her ontology is modified semi automatically and the dependency is kept its consistency. This function is available whenever he/she requests the change information of other ontologies.

## 4 Remaining Work

To advance distributed ontology development further, we have some issues to discuss. In this paper, we have already mentioned them a little. Here, we describe them in more details.

- *Keeping the Consistency from the Influencing Ontology;*

In this paper we took an approach to keep the consistency from the depending ontologies. On the other hand, we can consider the approach to keep the consistency of the dependency from the influencing ontology. We plan two stages to realize this. In the first stage, a developer is informed of the influence of the operation on other on-

tologies by the system. It is exceedingly helpful for developers to know the effect of their operation to other ontologies and take a low-risk approach. This function is like to be realized easily by using the information our system has. In the next stage, the system constrains user's changing operation on their own ontology so that consistency between the current ontology and other ontology may not be broken. This will be hard to realize because it may disturb developers operations to evolve ontology unless the levels of prohibition is discussed full well. However, this topic is abstruse to discuss.

- *Consistency of the Target Ontology and Version Control;*

In some situations, we cannot assume the consistency of the target ontology as a whole obtained by simply compiling these component ontologies. For example, it is a probable case that multiple component ontologies depend on several different versions of the same ontology. Then, we need to discuss the construction of component ontologies again from the view point of versions of ontology. We may have to construct the last version of ontology as different new ontology by dividing its several older versions.

- *Construction of Component Ontologies;*

This topic is concerned with determination of borderlines among component ontologies. And it connected with so many aspects of distributed ontology development that we cannot fully comprehend it. Here, we mention just three topics.

One is related to cooperative development. We have already designed the framework to support several developers to construct the same part of target ontology (described in section 3.3). However, if they construct in parallel, we have still problems to argue. We would allow several ontologies to describe the same part of target ontology when we aim to compare them, switch them, choice the best one and so on. They may be built in parallel or may be imported from the other target ontology. They are not regarded as the same, but resemble to ontologies which are several versions of ontology. So, in constructing phase, we may be able to keep their consistency in the framework like a version control.

Next is connected with the kind of relations between ontologies. We have dealt with two types of dependency between ontologies such as "*super-sub*" relation and "*referring-to*" relation. And now, we can see other kinds of relations between ontologies based on concepts hierarchy they have. For example *inclusive* relation, *parallel* relation and so on. These relations may be available to discuss the construction of target ontology as a whole. Besides this, we can find more kinds of relations based on content of ontology; such as a *task-domain* relation, a *role concept- basic concept* relation and so on. It may be useful for supporting the development of ontology to accommodate developers with a framework to manage content relations.

Last topic is how to determine borderlines among ontologies. In the case of "Plant Ontology", we did not explain why it can be divided so. It is true that the component ontologies are identified according to their conceptual levels or domains, but we didn't discuss how to divide and integrate ontology in this paper. Especially in distributed development, it is related to the working domain each developer has.

## 5 Related Work

Our basic motivation and design philosophy share a lot with CVS [6]. CVS (Concurrent Version System) has been often used in system and software development. CVS manages objects (in general, they are source code files) and controls their update. To avoid confliction caused by concurrent work done by several developers, CVS needs them to merge their own object and shared object updated in a repository. Our system also provides developers with the space for sharing target ontology. Each developer finds the difference between his/her editing ontology and shared one and modifies his/her own ontology to avoid conflict. However, in general, a programmer edits a source code in many ways. So, it is hard to understand the difference of versions considering variety of coding ways of the same semantics. On the other hand, ontology description languages are systemized and well-structured. Then, system can manage developer's action and offer them possible modifications to keep the consistency of their own ontology.

Some other ontology building tools also have been developed with functions for supporting collaborative development. OntoEdit [7] allows multiple users control their access to the same ontology to develop it collaboratively. We don't allow such operation. Instead, we allow users to divide one ontology into several component ontologies. While developers in OntoEdit treat all parts of their target ontology, each developer in Hozo can construct his/her own part of the target ontology individually and in parallel.

The Karlsruhe Ontology and Semantic Web framework (KAON) has developed an application, which resembles our system [8]. They intend to support evolution of ontology with its consistency kept. Their approach is very similar to ours. Both of them investigate the types of the change of ontology and strategies to keep the consistency. The differences between the two are as follows:

- 1) While KAON uses those strategies in the case of evolution of an ontology, we use them in different ontologies from the view point of distributed ontology development. But, their strategies seem to be also available in distributed development.
- 2) We discriminate dependency of "*Super-sub Relation*" and that of "*Referring-to Relation*".
- 3) Hozo can treat role concept unlike them.

## 6 Conclusion

In this paper, we discussed distributed ontology development and described our system to realize it. In our system, ontologies are managed based on their dependency among them. This relation is available also to keep the consistency of the dependency even if the change of ontology influences on another. These two aspects are located centrally in our research. Beside them, we discussed the framework for practicing distributed ontology development. Ontologies are managed from three view points;

*version control, updating and reusing*. And with considering cooperative development, we argued also the management of developers and their access to ontologies.

Our framework is realized in “Hozo” which is the system we have developed as an environment for building ontologies. The prototype of this system has been implemented although some details have not been done yet. And, we still have remaining works to evolve our framework.

Distributed ontology development can apply to many situations such as collaborative development, cooperative development, reusing ontologies and so on. In each of them, it will support developers to construct their target ontology.

## References

1. R. Mizoguchi, K. Kozaki, T. Sano, and Y. Kitamura: 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
2. E. Sunagawa, K. Kozaki, T. Kitamura, and R. Mizoguchi: Management of dependency between two or more ontologies in an environment for distributed development, Proc. of the International Workshop on Semantic Web Foundations and Application Technologies (SWFAT2003), pp.35-41, Nara, Japan, March 12, 2003
3. K. Kozaki, Y. Kitamura, M. Ikeda, and R. 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
4. K. Kozaki, Y. Kitamura, M. Ikeda, and R. 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
5. R. Mizoguchi, M. Ikeda, K. Seta, and V. Johontology for Modeling the World from Problem Solving Perspectives, Proc. of IJCAI-95 Workshop on Basic Ontological Issues in Knowledge Sharing, pp. 1-12, 1995
6. <http://www.cvshome.org/>
7. Y. Sure, M. Erdmann, J. Angele, S. Staab, R. Studer, and D. Wenke: OntoEdit: Collaborative Ontology Development for the Semantic Web, Proc. of the First International Semantic Web Conference (ISWC2002), Sardinia, Italy, June 9-12, 2002
8. L. Stojanovic, M. Maedche, B. Motik, N. Stojanovic: 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