Agent-based Architecture for Service Ontology evolution management

Soumaya Slimani, Salah Baina, Karim Baïna
ENSIAS, Mohammed V-Souissi University,
BP 713 Agdal, Rabat, Morocco
slimani.soumaya@gmail.com, {sbaina, baina}@ensias.ma

Abstract- Communication between services requires a common vocabulary to allow reliable exchange of information. However services are evolving independently and their ontologies evolve too. The existing distributed ontology evolution approaches are not scaled to dynamic environments like Semantic Web service architecture. As the distributed system grows in size, the complexity of ontology change management increases, especially if the ontologies are heterogeneous. In this paper, a new approach is proposed to manage ontology evolution in distributed architecture. Our proposal consists of a set of collaborative agents which dynamically manage different ontologies and their mapping.

I. INTRODUCTION

Semantic interoperability of heterogeneous Web services can be achieved through an agreement between the underlying ontologies. Indeed, ontologies allow services taking into account semantics of data during exchanges. Based on ontologies, several standards were proposed to describe semantic Web services (WSDL-S, WSMO, OWL-S, etc.) and thus, motivate the development of semantic services. Several works have developed tools and adopted approaches to develop semantic services \([1, 2, 3]\). In this context two scenarios are possible: (1) Distributed System based on single ontology and multiple instances of this ontology (SOMI). (2) Distributed System based on multiple ontologies and single or multiple instances of these ontologies (MOSI, MOMI). Indeed, ontology management can be complex and time consuming. Moreover, in a collaborative context, an ontology that evolves can cause the evolution of other ontologies or instances of ontology. Several studies have addressed this issue \([4, 5, 6]\). However, research in ontology evolution and change management deal particularly with the versioning and evolution of the same ontology. When ontologies are considered as instances of a source ontology, the ontology evolution will require applying the same changes on the instances.

In this paper we present a new approach for managing ontologies in a distributed environment, we treat the case of different ontologies and not instances of the same ontology. Note also that in the case of different ontologies, related mappings must be managed in parallel.

II. OUR PROPOSITION FOR MANAGING ONTOLOGY EVOLUTION

The typical characteristics of ontology and service ontology-based applications can be identified as: distributed, heterogeneous, and highly dynamic. Agent technology is thus ideally suited to this context.

A. Architecture overview

Our architecture consists of a set of Ontology Agent (OA) that acts on a service ontology. Agents are responsible of change detection and propagation of these changes to other dependent agents. Dependent agents manage and update their ontologies and the related mapping. The overall architecture of our proposal is shown in Fig. 1.

Fig. 1. Architecture overview

An agent will take one of two roles: (i) Initiator Ontology Agent, when its ontology changed, and (ii) Dependant Ontology Agent, when its ontology depends on a changed ontology. So, we present two algorithms to illustrate agent roles: (1) Detection/propagation of change (initiator role). (2) Receipt/processing of change (receiver role).

B. Algorithm of detection / propagation of change (initiator side)

Our algorithm illustrated in Fig.2, takes as input an ontology and a list of receivers (agents of the dependent ontologies). Once a change is made, the agent lists these changes automatically. We define a change \(C_i\) by a quadruplet:

\[
C_i = \langle \text{Operation}, \text{Object}, \text{Subject}, \text{Predicate} \rangle
\]  

An Operation specifies the change operation (Add, Delete, etc). The Object specifies the type of component that has changed (Class, Property, individual). The Subject represents the name of the component that has changed. Predicate represents a set of information about
the change. If a class was changed, this field contains the position of the class in the hierarchy of the ontology (its super-classes and sub-classes). But, if a property was changed, it contains the Range, Domain, super-property and sub-properties.

We also define a message or changeSet by a list of changes:
\[ \text{changeSet} = \{C_1, C_2, ..., C_n\} \tag{2} \]

After listing all changes, the agent classifies these changes into two categories:

1. Changes contained in the related mapping between initiator ontology and receiver ontology or that are connected (sub-class, property, etc.) to an elements in this mapping. These changes are encapsulated into a message and sent to all dependent ontologies agents.
2. Changes that are not related to the mapping between the two ontologies. These changes are ignored because they do not involve communication between the two services.

For changes to be propagated, translation phase is necessary. The translateChange method generates a new list of changes by replacing the objects of the field Predicate in the original list by their match in the related mapping.

C. Algorithm receipt / processing of change (receiver side)

As shown in Fig. 3, the receiver side algorithm consists of six main operations:

1. Semantic search: this method looks for a semantic corresponding object of change (synonymy, etc.).
2. Syntactic search: find if there is a concept in the ontologie which has the same syntax as the changed object.
3. Computation of object definition: Class definition is expressed by the position of the class in the hierarchy of the ontology (its super classes and subclasses).

Property definition is made based on its "Range", its "Domain", and its supers or under properties.

4. Negotiation of object definition: Negotiations, in our case, correspond to the reformulation of the object definitions, and the exchange of these definitions. The receiver agent calculates a definition of an object, and then sends it to the initiator agent. If the initiator agent finds that the two definitions are equivalent, it sends its agreement; otherwise it recalculates the definition, and send the new definition to the receiver agent, and so on. The negotiation ends when either receiver and initiator agent have a common definition of the object or they find a conflict.

5. Annotation of ontology objects: the annotateOntology method allows us to add new object as annotation of existing object. An annotation is a comment, a note or any other external remarks that can be attached to an entity of the ontology. In OWL DL, it is possible to annotate classes, properties and individuals. There are five predefined annotation properties in OWL: Owl:versionInfo, Rdfs:label, Rdfs:seeAlso, Rdfs:isDefinedBy.

6. Ontology update Mapping: If ontologies evolve, the mapping must evolve too. The updateMapping method allows adding new correspondence between two concepts in the mapping.

---

Algorithm : Change Detection and propagation

Require : Ontology O, List of Receiver receivers

While (true)
O.changed()=listen(O)
If(O.changed() == true) then
changeSet= calculateChangeSet()
For each Receiver Ri of receivers do
Mapping Mi= GetMapping(O, Ri)
For each change c of changeSet
If (c.isRelated(Mi) == true) then
Change Nc = translateChange(c,Mi)
newChangeSet.add(Nc)
end if
end For
Send( newChangeSet, Ri)
end For
end if
end While

Fig. 2. Initiator side Algorithm

Algorithm Receive ChangeSet: ADD Operation

Require: changeSet:List of Changes, m:Mapping, O : Ontology

Waiting for Message

For each change c in changeSet do
Object S, cS
cS= c.Subject
Boolean synSearch=O.syntacticSearch(cS)
Boolean semSearch=O.semanticSearch(cS, S)
If (c.Operation == ADD) then
If (synSearch && semSearch) then
m.updateMapping(S, cS)
end if
If (synSearch && !semSearch) then
Definition d = calculateDefinition(cS)
Boolean agree = negotiate(d)
If agree == !false then
informUser ()
else
m.updateMapping(cS, cS)
end if
end if
If (!synSearch && semSearch) then
O.annotateOntology( cS,S)
m.updateMapping(S, cS)
end if
If (!synSearch && !semSearch) then
O.add(cS)
m.updateMapping(cS, cS)
end if
end for

Fig. 3. Receiver side Algorithm
Based on the results of the syntactic and semantics search, the agent chooses one of four alternatives and executes related actions summarized below:

<table>
<thead>
<tr>
<th>TABLE I</th>
<th>DEPENDENT AGENT ACTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>True</td>
</tr>
<tr>
<td>Semantic search</td>
<td>-Updates the related mapping</td>
</tr>
<tr>
<td>True</td>
<td></td>
</tr>
<tr>
<td>False</td>
<td>-Reformulates the change definition. -Negotiates the change definition.</td>
</tr>
</tbody>
</table>

III. CASE STUDY

In this section, we present an example of communication between services. We limit to an example of two services. But this example can be generalized to a more complex architecture. The running example ontologies come from the domain of Telecom and concern two services. The first service is the SMS billing service (S₁), the second one is the Internet billing service (S₂). S₁ is described by an ontology O₁ and S₂ is described by an ontology O₂. Initially, services (S₁ and S₂) communicate via a Mapping M. S₂ adds a new service providing information to the client by SMS. This SMS, called INTERNET_SMS, contains information about the status of the client account. These SMS is sent by the client service to the client. Then, S₂ adds a new Internet SMS service, which a client can send to another via Internet. We call this new type of SMS: INTERNET_SMS and we rename the old one INTERNET_SMS_INFO.

Following these evolutions, both services have two different interpretations of the concept INTERNET_SMS. So, communication can not be done properly.

The application of the algorithm in this scenario begins on the Initiator side (O₂ agent). First, changes are listed as shown in Fig.4.

```
changeSet = <
C₁=<ADD:INTERNET_SMS:subclassof SMS>,
C₂=<ADD:ObjectProperty:INTERNET_SMS_SENDER:
   <Domain:INTERNET_SMS,Range:CLIENT>>
C₃=<ADD:ObjectProperty:INTERNET_SMS_RECEIVER:<Domain:
   INTERNET_SMS,Range:CLIENT>>
```

Fig. 4. The change set after adding INTERNET_SMS

Changes are classified according to their relationship with the mapping. All changes are sent to O₁ agent. In the receiver side (O₁ agent), the syntactic and semantic search will return false. Because all added objects are new objects for the ontology O₁. Thus the agent chooses the fourth case. So, C₁, C₂, and C₃ are added to O₁ and the mapping is updated (i.e. mapped(O₁,C₁, O₂,C₃b, ...)). Thus, the Internet service can communicate with the SMS service using the concept INTERNET_SMS. However, after adding the new Internet SMS type, the mapping between O₁INTERNET_SMS and O₂INTERNET_SMS becomes wrong. Semantic of INTERNET_SMS has changed in O₂, but it kept the same semantics in O₁. Our algorithm handles this case and O₂ agent generates a list of changes as shown in Fig.5.

```
changeSet = <
C₁=<ADD:INTERNET_SMS:subclassof SMS>>,
C₂=<ADD:ObjectProperty:INTERNET_SMS_SENDER:
   <Domain:INTERNET_SMS,Range:CLIENT>>,
C₃=<ADD:ObjectProperty:INTERNET_SMS_RECEIVER:
   <Domain:INTERNET_SMS,Range:CLIENT>>
```

Fig. 5. The change set after adding the new INTERNET_SMS

These changes are sent to O₁ agent. The object INTERNET_SMS in O₁ has a different semantic that INTERNET_SMS in O₂. In this case the agent chooses the second case in the algorithm. O₁ agent calculates a definition for INTERNET_SMS as shown in Fig.6.

```
<<class:O₁.INTERNET_SMS:subclassof shortMessage>
<Class:O₁.SERVICE_CLIENT:subclassof Thing>
<ObjectProperty:O₁.SENDER:Domain:
   O₁.INTERNET_SMS,Range:O₁.SERVICE_CLIENT>>
```

Fig. 6. The new INTERNET_SMS definition calculated by O₁

O₁ agent sends this definition to O₂ agent, which calculates a new definition:

```
<<class:O₁.INTERNET_SMS:subclassof shortMessage>
<Class:O₁.SERVICE_CLIENT:subclassof Thing>
<ObjectProperty:O₁.SENDER:Domain:
   O₁.INTERNET_SMS,Range:O₁.SERVICE_CLIENT>
<Class:O₂.INTERNET_SMS:subclass of SMS>
<ObjectProperty:O₂.SENDER:Domain:
   O₂.INTERNET_SMS,Range:CLIENT>
<0₁.CLIENT#O₂.SERVICE_CLIENT>>
```

Fig. 7. The new INTERNET_SMS definition calculated by O₂

The last matching, mapped (CLIENT, SERVICE_CLIENT), introduces a conflict in the definition of SERVICE_CLIENT.

Indeed, as we have already in the mapping that:

mapped (O₁.CLIENT, O₂.CLIENT)

Introducing in the mapping the following correspondence:

mapped (O₁.CLIENT, O₂.CLIENTSERVICE)

Implies that:

mapped (O₂.CLIENT, O₂.SERVICE_CLIENT)

This operation can generate an error. In this case, the agent sends an alert message to the user. The user can modify the mapping and the ontology O₁ or validate any of the definitions contained in the negotiations exchange.

Our algorithm provides a support for communication between services. In this example, it allows to services (S₁ and S₂) to be aware of evolution in other services. Agents take the necessary decisions to maintain this
communication by applying the necessary changes to the mapping and to the ontologies. But if the agent detects a conflict it only informs the user of the problem.

IV. IMPLEMENTATION OVERVIEW

The initiator algorithm and the receiver side algorithm have been implemented. Our proposal presents a solution that can be easily used by service designers and developers within their SOA technical infrastructures and SOBA business solutions. As shown in Fig. 8, four layers are defined: (1) Service layer: This is outside our architecture, represents services that are described by service ontologies. (2) Mapping layer: Service dependencies are represented by service ontology mappings defined by human designers too using ontology mapping generators. (3) Ontology layer: Service ontologies are defined by human designers using ontology editors, and these service ontologies are then monitored synchronously. (4) Agent layer: Ontology agents are defined following our Ontology Agent Model and algorithms described in the section II.

For each layer we have studied many alternatives to develop our prototype. For storing ontologies, we have studied DAML+OIL, OWL, RDF, RDF(S). We have finally chosen OWL (http://www.w3.org/TR/owl-features/). We have chosen to work with Protégé 3.4 (http://protege.stanford.edu), but other editors may be used since the Ontology Agent is listening to ontology basic file change events before their processing. For ontologies mapping, we have studied LOM, IFMap, OMEN, GLUE, FCA-merge, Prompt (Protégé plugin). Our final choice was for Prompt (http://protege.stanford.edu/plugins/prompt/t/prompt.html) that generates an OWL format for the mapping.

Finally, we have studied many alternatives for implementing the Ontology Agent Model: AgentBuilder, ASL, FIPA-OS, MOLE, Open Agent Architecture, RETSINA, Zeus, JADE. And, we have finally chosen JADE (http://jade.tilab.com/) to develop our Ontology Agent Model. Interactions between agents (i.e. ontology change event messages) are handled through an ontology change communication channel.

V. CONCLUSION AND FUTURE WORKS

Algorithms for ontology evolution are not suited for semantic service based application. In these applications, services are described by ontology and collaborate with each other. In this context, each entity has its own semantic and its own interpretation of the exchanged data. This makes communication between services difficult and maintaining this communication is more difficult and complex. We adopt a MAS perspective to model ontologies in ontology-based applications. An agent-based approach aims to provide a flexible way to automate the process of ontology evolution management as much as possible. Moreover, agents which are autonomous and engaged in flexible interactions can perceive changes in the environment and adopt corresponding actions to maintain the mapping between ontologies. Our algorithm covers the basic changes in initiator side and treats the Add operation in the receiver side. We consider the update operation as a combination of deletion and addition operation. Deletion is still being analyzed in our algorithm.

The ontology agent model contains three behaviors: (1) a listener to detect change, (2) a behavior for changes formulation and propagation (initiator side Algorithm), and (3) a behavior for receiving and processing changes (receiver side algorithm). We have developed our model in the agent platform JADE.

The approach could be extended and improved in several ways. There is a need to extend the algorithms to include different types of changes in our algorithms (removal, update, etc...). This paper is thus a first step to lay the groundwork for future research.

REFERENCES