World Academy of Science, Engineering and Technology
International Journal of Information and Communication Engineering
Vol:1, No:10, 2007

Semantic Web Agent Communication Capable of Reasoning with Ontology and Agent Locations

Visit Hirankitti and Vuong Tran Xuan

***Abstract*—**Multi-agent communication of Semantic Web information cannot be realized without the need to reason with ontology and agent locations. This is because for an agent to be able to reason with an external semantic web ontology, it must know where and how to access to that ontology. Similarly, for an agent to be able to communicate with another agent, it must know where and how to send a message to that agent. In this paper we propose a framework of an agent which can reason with ontology and agent locations in order to perform reasoning with multiple distributed ontologies and perform communication with other agents on the semantic web. The agent framework and its communication mechanism are formulated entirely in meta-logic.

***Keywords*—** Semantic Web, agent communication, ontologies.

I. Introduction

COMMUNICATION of Semantic Web (or briefly "SW") information between browsers and servers can be understood as multi-agent communication. However, this communication cannot actually be realized without the need to reason with ontology and agent locations. This is because for an agent to reason with an external SW ontology, it must know where and how to access to that ontology. Similarly, for an agent to communicate with another agent, it must know where and how to send a message to that agent. To achieve this, in this paper we propose a meta-logical model of SW communication among agents using meta-information of ontology and agent locations.

Some previous works on an agent system related to SW are: Zou et. al. [4] used SW languages, as the languages for expressing agent's messages and knowledge base, to specify and publish common ontologies; [5] presented a multi-agent based scheduling application in which data sources are described by SW languages and encapsulated in the agents. In [6], an agent is built to perform scheduling with distributed ontologies about events, e.g. conferences, classes, published on the SW. Those approaches are mainly related to applying the SW technology in a multi-agent system. However, here we are concerned with multi-agent communication and reasoning with distributed ontologies and some works [1, 2] were done previously. In this paper we have extended the communication framework in [2] to reason with ontology and agent locations.

The rest of this paper is organized as follows. Next we give an overview of our framework. Section III presents our meta-representation of SW ontologies and section IV describes our single agent architecture. Section V describes the meta-interpreter which can reason with ontology and agent locations, and section VI introduces multi-agent communication. Section VII shows how to query and reason with SW ontologies by multi-agent communication. Section VIII covers some implementation issues. Finally, we discuss about other related works and conclude this paper.

II. Our Framework

The meta-logical system for one agent consists of three main parts: meta-programs for multiple ontologies, a meta-interpreter, and the communication facility. Each meta-program contains meta-logical representations of ontologies obtained from the transformation of these ontologies defined in RDF, RDFS, and OWL. Some elements in one ontology may be related to some elements in another. The meta-interpreter is the inference engine for infering implicit information from the multiple ontologies. The communication facility supports the communication among the agents. One block in Fig. 1 illustrates one agent.

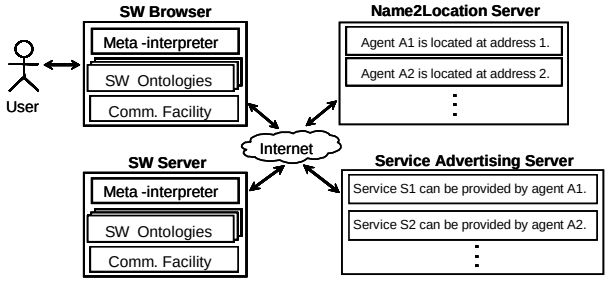

Fig. 1 Our SW multi-agent communication system

When several agents of this kind are formed as a community, the way the multi-agent system works is that initially the user queries an SW browser to get answers from an SW ontology on SW. The browser can perform two alternative ways.

Firstly, it may retrieve this ontology from SW, transform it into a meta-program, and then reason with the program to infer

Manuscript received November 30, 2005. This research was supported by the Japan International Cooperation Agency (JICA) under the ASEAN University Network / Southeast Asia Engineering Education Development Network (AUN/SEED-Net) Program.

Visit Hirankitti is with the Department of Computer Engineering, Faculty of Engineering, King Mongkut's Institute of Technology Ladkrabang, Bangkok, Thailand (corresponding author to provide phone: +66-2-739-2400; fax: +66-2-739-2404; e-mail: visit@ce.kmitl.ac.th).

Vuong Tran Xuan is with the Department of Computer Engineering, Faculty of Engineering, King Mongkut's Institute of Technology Ladkrabang, Bangkok, Thailand (corresponding author to provide phone: +66-2-739-2400; fax: +66-2-739-2404; e-mail: txvuong@yahoo.com).

the answers; if some elements in this ontology are related to some elements of another ontology, the interpreter will try to reason with that ontology in itself (by retrieving it first), or request reasoning of that ontology in an SW server and obtain the answers from that server, and this scenario may repeat itself. For the browser to be able to retrieve an ontology, it must know which server the ontology belongs to, and how and where to access to it. This is the ontology's meta-information provided in the ontology. The browser will use this to contact with that server and request that ontology from it, or to pass a query to that server so that the server can derive an answer from the ontology.

Alternatively, the browser passes the query to an SW server to answer and gets the answers back for the user. The server infers those answers based on its inferential results which sometimes also require support of the inferential results derived from other servers. In case the browser does not know which server can answer that query, it will consult the Service Advertising Server which gathers information telling which server can provide what service. The browser then uses this information to communicate with the selected server directly. For the browser to communicate to any server as said earlier, having known the server name the browser will pass the name to the Name2Location server to obtain the server location and then make contact with that server at that location. Note that conceptually the term 'location' we use here is intended to be an abstract one; an agent location could be the place, such as an address (IP address) on the Internet, or even a (postal) address, where the agent can be reached.

III. The Meta-Languages and Meta-Programs

A. Language Elements of the Semantic Web Ontology

The language elements of ontology are classes, properties, class instances, and relationships between and among them described in the object level and the meta-level as in Fig. 2.

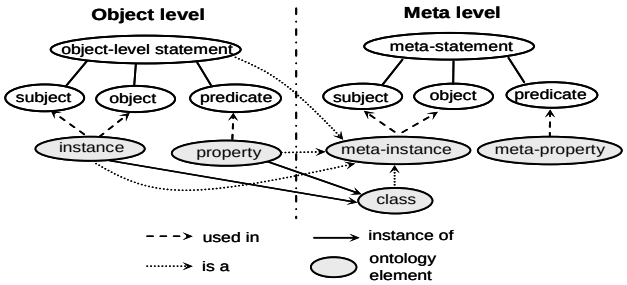

Fig. 2 The elements of an SW ontology at the object level and meta-level

At the object level, an instance can be an individual or a literal of a domain; and a property is a relationship between individuals, or is an individual's attribute. At the meta-level, a meta-instance can be an individual, a property, a class, and an object-level statement; and a meta-property is a property to describe a relationship between and among meta-instances, or is a meta-instance's attribute.

B. Meta-information of Ontologies

To facilitate communication between agents and reasoning with multiple distributed ontologies, language elements of an ontology should be associated with an ontology name. An ontology should also be related to the agent possessing it, the agent's communication channel used to access to that ontology, the file containing this ontology, and the file's path location. This is some *meta-information* of the ontology and it should be treated as a part of a meta-level of the ontology.

C. Meta-languages of the Semantic Web Ontology

In our framework, for an SW ontology we distinguish between its object and meta levels, and similarly its object and meta languages. Hence, we have formulated two meta-languages: one discusses mainly about objects and their relationships we call "meta-language for the object level (**ML**)" and the other, called "meta-language for the meta-level (**MML**)", discusses mainly about classes, class instances, properties, and their relationships. **MML** includes the meta-language representing the meta-information of an ontology discussed earlier in III.B. Due to some connections between the object and meta levels, **ML** and **MML** are slightly overlapped.

- **A meta-language for the object level (ML)**

Objects and their relationships at the object level as well as some provability and references at the meta-level are specified in an SW ontology and this information is expressed at the meta-level by the elements below. (Note that the linguistic elements of provability are a part of **AgentML** (see section V) and the elements expressing references are a part of **MML**.).

Meta-constant specifies a name of an object and a literal, including a reference, e.g. a namespace and an ontology name.

Meta-variable stands for a different meta-constant at a different time, e.g. `Person`.

Meta-function symbol stands for the name of a relation between objects, or of an object's property—i.e. an object-level predicate name, such as `'fatherOf'`—including the name of provability predicate, i.e. `demo`. The meta-function symbol also stands for other meta-level function symbol, e.g. `'←'`, `'∧'`, `':'`, `'#'`. Finally the meta-function symbol can also be a term in the form `<ontology_name>:<namespace>#<object-level predicate name>` where `':'` and `'#'` are meta-function symbols, and `<ontology_name>` and `<namespace>` are meta-constants or meta-variables.

Meta-term is either a meta-constant or a meta-variable or a meta-function symbol applied to a tuple of terms, e.g. `'family_ont':'f'#'M1'.`

To express an object-level predicate, it has the form: **`P(S, O)`** where **`P`** is an object-level predicate name, **`S`** and **`O`** are meta-constants or meta-variables (we presume all meta-variables appearing in the tuple are universally quantified), e.g. `'f'#'fatherOf'('f'#'M2', 'f'#'M1')`. To express a provability predicate, it has the form: **`demo(A, T, P(S, O))`**, e.g. `demo(a, o, 'o':'f'#'fatherOf'('o':'f'#'M2', 'o':'f'#'M1'))`.

The meta-term expressing an object-level sentence (sometimes with some provability) is a term **P(S, O)** or **demo(A, T, P(S, O))** or a logical-connective function symbol applied to the tuple of these terms. One form of it is a Horn-clause, e.g.

```
'o':'f'#'fatherOf'(F, Ch) ←
    demo(b, ob, 'ob':'p'#'parentOf'(F, Ch)) ∧
    demo(c, oc, 'oc':'m'#'male'(F)).
```

Meta-statement for the object level reflects an object-level sentence to its existence at the meta-level. It has the form: **statement(T, object-level-sentence)**, where **T** is an ontology name, e.g.

```
statement(o∪ob∪oc,'o':'f'#'fatherOf'(F, Ch) ←
  demo(b, ob, 'ob':'p'#'parentOf'(F, Ch)) ∧
  demo(c, oc, 'oc':'m'#'male'(F))).
```

- **A meta-language for the meta-level (MML)**

Additionally, an SW ontology defines classes, properties, and their relationships, and also class-instance relations. This information is precisely *meta-information of the object level*. Here we express this information by **MML** which includes:

Meta-constant specifying a name of an agent, a namespace, an ontology, a communication channel, a file's path location, a file, an instance, a property, a class, and a literal.

Meta-variable standing for a different meta-constant at a different time.

Meta-function symbol naming a meta-level function, e.g. `port`, `protocol`, `#`, `:`, `path`, `file`, `location`.

Meta-term is either a meta-constant or a meta-variable or a meta-function symbol applied to a tuple of terms, e.g. `port(80)`, `protocol(http)`, `location(path('/'), file('family.owl')).`

In our framework, a name of a class, a property, etc., can be referenced by a meta-term in these three forms: `uniqueName` or `namespace#name` or `ontologyName:namespace#name`, e.g. `'owl'#'inverseOf', 'o':'f'#'fatherOf'`.

Meta-predicate name naming a relation between entities, or a characteristic of a property, which fall into one of the following categories: class-class relations, class-instance relations, property-property relations, relations between literals and instances/classes/properties, and characteristics of properties [2]. A predicate name is labeled with a term to be associated with its namespace and the name of an ontology it belongs to.

Meta-predicate expressing a relation between entities of the form **Pred(Sub,Obj)**, or a characteristic of a property in the form **Pred(Prop)**, where **Pred** is a meta-predicate name, **Sub**, **Obj**, and **Prop** (a property) are meta-terms, e.g. `'owl'#'inverseOf'('o':'f'#'fatherOf', 'o':'f'#'childOf')`. Let all the meta-variables appearing in a meta-predicate be universally quantified.

Meta-operator expressing a set operation between classes such as union, intersection.

Meta-statement being a meta-predicate or meta-predicates connected by logical connectives. One form of the meta-statement is a Horn-clause **meta-rule**. Here are some examples of the meta-rules:

```
meta_statement(o, 'owl'#'inverseOf'('o':'f'#
  'fatherOf', 'o':'f'#'childOf') ← true).

axiom(t, 'owl'#'equivalentClass'(C, EC) ←
   'owl'#'equivalentClass'(C, EC1) ∧
   'owl'#'equivalentClass'(EC1, EC)).
```

The second rule represents a mathematical 'axiom'.

- **A meta-language for the meta-information of ontologies**

A meta-language expressing the meta-information of ontologies discussed earlier in the section III.B is also included in **MML**, although it could be taken to be at a higher meta-level than **MML**; but for the simplicity we did not do that. The meta-information relates an ontology to its agent, the communication channel used to access to it, a file's path location, and the file that contains it; this meta-information is formulated in **MML** in the form of `meta_info_statement(ontology, agent, port, protocol, location(path, file))`, e.g. `meta_info_statement(dmp, bookShopAgent, 80, http, location('/', 'DocOnto.owl')).`

D. Meta-programs of the Semantic Web Ontology

Each ontology is transformed to a meta-program containing a (sub-)meta-program expressed in **ML**, called "**MP**", and/or a (sub-)meta-program expressed in **MML**, called "**MMP**". Another meta-program expresses some mathematical axioms using MML, called "**AMP**". The inference engine often requires **AMP** to reason with **MP** and **MMP**.

- **The meta-program for the object level (MP)**

MP contains meta-statements for the object level: `statement(T, P(S, O) ← true)` and `statement(T, P(S, O) ← Body)`, where `Body` expresses a conjunction of object-level predicates and some provability; the latter is its Horn-clause form. Note that to state that a meta-statement belongs to an ontology `T` we put `T` as the first argument; and this form of ontology labeling will be used henceforth.

- **The meta-program for the meta-level (MMP)**

MMP contains description of classes, properties, their relations, and class-instance relationships in terms of meta-rules. It also contains statements expressing ontology meta-information. Here is an example of a statement in **MMP**:

```
meta_info_statement(dmp, bookShopAgent, 80,
   http, location('/', 'DocOnto.owl')).
```

- **The meta-program for the axioms (AMP)**

AMP contains axioms for classes and properties. For example, an axiom defining an equivalence of classes:

```
axiom(T, 'owl'#'equivalentClass'(C, EC) ←
   'owl'#'equivalentClass'(C, EC1) ∧
   'owl'#'equivalentClass'(EC1, EC)).
```

IV. Single SW Agent Architecture

An agent in our framework is denoted by <**Meta-interpreter**, **Knowledge Base**, **Communication, Historical Memory**, **Transformation**> whose components are depicted in Fig. 3. The Transformation module transforms ontologies obtained from SW to **MP**s, **MMP**s, and **AMP**s, and the knowledge base stores them. The Meta-interpreter reasons with them in order to answer queries posed by the user, and communicates with other agents to get ontologies or answers for queries. The Historical Memory stores information required for advance reasoning by the meta-interpreter. The Communication module facilitates communication with other agents.

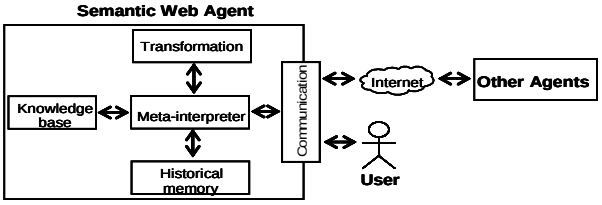

Fig. 3 Single SW Agent Architecture

- **Meta-language of the agent (AgentML)**

AgentML is a meta-language we use to formulate the agent. It discusses about the agent's components, such as the `demo(.)` definition, the agent's name and resources, assumptions in ontologies (this part is connected with **MP** and **MMP**), communication methods and facilities, its locations used for communication, other agents and their ontologies, and so on.

V. A Communicative DEMO

The `demo` predicate [7] is used as our meta-interpreter. Our `demo` definition, which can reason with multiple distributed ontologies and communicate with other agents proposed in [1, 2], has been extended here to reason with ontology and agent locations in order to realize its task of communication of SW information. For `demo(Agent,T,A)`, it means an answer `A` can be inferred from a theory `T` by an agent `Agent`. In [2] the Vanilla is adapted for reasoning with multiple ontologies where we identified three kinds of meta-level statements, (1) `statement(T,A←B)`, (2) `meta_statement(T,A←B)` for the meta-level of an ontology, and (3) the mathematical axioms `axiom(T,A←B)`. The definition of `demo/3` is:

```
demo(_,empty,true).                                   (true)

demo(Agent,T1∪T2,A '∧' B) ←                           (conj)
  demo(Agent,T1,A) ∧ demo(Agent,T2,B).

demo(Agent,T,demo(Agent',T,A)) ←                      (ref)
  demo(Agent',T,A).

demo(Agent,T1∪T2,A) ←                                 (ost)
 statement(T1,A '←' B) ∧demo(Agent,T2,B).

demo(Agent,T1∪T2,A) ←                                 (mst)
  meta_statement(T1,A '←' B) ∧
  demo(Agent,T2,B).

demo(Agent,T1∪T2,A) ←                                 (ast)
  axiom(T1,A '←' B) ∧ demo(Agent,T2,B).
```

The clauses (true), (ost), and (conj) form the Vanilla. The clause (ref) states that when the meta-interpreter tries to prove `demo(Agent,T,demo(Agent',T,A))`, it will prove `demo(Agent',T,A)` by a reflection.

For distributed ontologies, some ontologies may be referred to in others. In this case while `demo` is reasoning with an ontology to derive an answer, this may require it to reason with another unavailable ontology. So we add the following clause to allow `demo` to retrieve that ontology from its location on the web, transform it into **MP** and **MMP**, and then reason with it to complete all the inference steps so that it can derive the answer.

```
demo(Agent,T,demo(Agent',T,A)) ←                      (retr)
  myName(Agent') ∧ unavailable(T) ∧
  O:NS#Goal = A ∧
  meta_info_statement(
    O,Agent'',Port,Channel,Location) ∧
  retrieve(O,Agent'',Port,Channel,Location) ∧
  transform(O,P) ∧ demo(Agent,P,A).
```

With this clause, `demo` can work analogously to a browser.

Additionally, when each server storing an ontology is equipped with this `demo` definition, for `demo` (at the client) to derive an answer from an unavailable ontology, this can be done by that the `demo` sends the query (for an unavailable ontology) to the server, which has that ontology, to answer the query. For this to be done, we may add two more `demo` clauses:

```
demo(Agent,T,                                 (certain-agent-comm)
     demo(Agent',T,A)) ←
  not myName(Agent') ∧ known(Agent') ∧
  unavailable(T) ∧
  agentLocation(Agent',Location,
                Port,Channel) ∧
  connect(Location,Port,Channel,ConnectID) ∧
  communicate(ConnectID,demo(Agent',T,A)) ∧
  disconnect(ConnectID).

agentLocation(Agent,Addr,Port,Ch) ←
  connect(www.n2l.net,80,http,ConnectID) ∧
  communicate(ConnectID,demo(www.n2l.net,T,
    name_location(Agent,Addr,Port,Ch))) ∧
  disconnect(ConnectID).

demo(Agent,T,                           (applicable-agent-comm)
     demo(Agent',T,A)) ←
  unknown(Agent') ∧ unavailable(T) ∧
  findAgent(Agent',A) ∧
  demo(Agent,T,demo(Agent',T,A)).

findAgent(Agent,Goal) ←
  agentLocation(sa_sever,Location,
                Port,Channel) ∧
  connect(Location,Port,Channel,ConnectID) ∧
  communicate(ConnectID,demo(sa_server,_,
    agentCapability(Agent,Service))) ∧
  matchOK(Goal,Service) ∧
```

```
disconnect(ConnectID).
```

In this clause, the `demo` searches for an agent who can provide the service by asking a service advertising server (see section VI).

Given all the above clauses, `A` can be inferred from `demo` in different ways: firstly `A` may be inferred using statements in one or many **MP**s, and/or using meta-statements in **MMP**s, and/or using axioms in **AMP**. Alternatively, the inference may require `demo` to retrieve some ontologies from different sources on SW or to send `demo(Agent,T,A)` to other servers to request for the answer.

- **Agent creation and the agent's life cycle**

To create and start a new agent, we perform: (1) assign a unique name to the agent by asserting `myName(agentName)`; (2) set up its communication channels; (3) register its name, locations, ports, and channels to the Name2Location server; (4) start the agent to do an endless observation—action cycle—listening to the communication channels to get a request from the user or other agent, and responsing to that request accordingly; when the response is done it returns back to the observation stage again.

VI. Multi-Agent Communication

An individual agent created by our agent framework can behave in two fashions. One is to work as an SW browser and the other is to work as an SW server. The only difference is that the former communicates with a human user and SW servers, whilst the latter communicates with SW browsers and other SW servers. Due to the usage of the current web, we expect that a multi-agent community of SW would consist of SW browsers, SW servers, Name2Location servers and Service Advertising servers virtually linked together on the web (see Fig. 1).

A Name2Location Server provides a communication location of an agent when being asked with an agent name. It has the fixed address: `'www.n2l.net'`. It possesses the facts in the form of `name_location(Agentname, AgentAddress, Port, Channel)`.

A Service Advertising server has the name `'sa_server'`. It maintains information telling which agent can provide which service in the form of `agentCapability(Agent, Service)`, where `Agent` is a name of a registered agent and `Service` is a service provided by the agent in the form of `OntologyName:Namespace#PredicateName(…)`.

VII. The Query Answering

To illustrate our framework, we use a book purchase scenario. Suppose we have an online bookshop selling books supplied by some publishers and providers. The bookshop, the publishers, and the providers have their own SW servers which provide information about the books able to be supplied by them. This information is described by some ontologies and there are differences between the ontologies in the servers of different bookshops, different publishers, and different providers.

An online book purchase begins with a customer wants to buy some books from a bookshop. He then uses an SW browser to get some book information—i.e. title, short description about the book—(expressed in some ontologies) from a bookshop SW server. This information helps him decide which titles to buy. Sometimes, he may want to get more information of the interested titles, such as publishers, book cover types (e.g. paperback, hardcover), and prices before placing an order with the bookshop server. Suppose this information is not stored in the bookshop server, but the server can request it from some (probably unknown) publisher servers and/or provider servers. In Fig. 4, we list only some parts of the meta-programs, **MP** and **MMP**, possessed by a publisher server, a provider server, the bookshop server, and also a part of service advertising information in a service advertising server, respectively.

A demonstration of the query answering of the SW browser is shown in Fig. 5. To answer the first query, the SW browser reasons with its ontologies obtained from the bookshop server. However, for the second, the browser adopts **BMP**'s the fourth statement, and this requires it to pass this query to the bookshop server to answer. The bookshop server uses **DMP**'s fifth statement to infer the ISBN from the title; and it then queries an unknown publisher and the provider `providerAgent` for the cover type and price respectively. That is, for the book cover, the bookshop server does not know which agent to ask but for the book price, the book shop server knows that it may ask the `providerAgent` server. For both cases, the bookshop server has to consult to the service advertising server to find the locations and services of these servers and to post its corresponding queries to them and get the answers back. The bookshop server then returns all the answers to the SW browser for presenting to the user.

Publish SW Server

PMP: *Meta-program for the publication ontology*
`meta_info_statement(pmp,publisherAgent,80,http,location('/','PublisherOnto.owl')).`
`statement(pmp,'pmp':'p'#'bCover'('pmp':'p'#'0262635828','hard') ← true).`

Provider SW Server

PPMP: *Meta-program for the publication provider ontology*
`meta_info_statement(ppmp,providerAgent,80,http,location('/','PubProviderOnto.owl')).`
`statement(ppmp,'ppmp':'pp'#'bPrice'('pmp':'p'#'0262635828','$40') ← true).`

Bookshop SW Server

BMP: *Meta-program for the book ontology*

```
meta_info_statement(bmp,browser,80,http,location('/','BookOnto.owl')).
meta_info_statement(dmp,bookShopAgent,80,http,location('/','DocOnto.owl')).
meta_statement(bmp,'rdf'#'type'(
  'Genetic Algorithm','bmp':'b'#'GeneticProgramming') ← true).

statement(dmp u T,'dmp':'d'#'bookInfo'(Title,Cover,Price) ←
  demo(bookShopAgent,T,'dmp':'d'#'bookInfo'(Title,Cover,Price))).
```

DMP: *Meta-program for the documentation ontology*

```
meta_info_statement(dmp,bookshopAgent,80,http,location('/','DocOnto.owl')).
meta_info_statement(pmp,_,80,http,location('/','PublicationOnto.owl')).
meta_info_statement(ppmp,providerAgent,80,http,location('/','PubProviderOnto.owl')).
statement(dmp,'dmp':'d'#'bTitle'('pmp':'p'#'0262635828','Genetic Algorithm') ← true).
statement(dmp u pmp u ppmp, 'dmp':'d'#'bookInfo'(Title,Cover,Price) ←
  'dmp':'d'#'bTitle'(ISBN,Title) ∧
  demo(_,pmp,'pmp':'p'#'bCover'(ISBN,Cover)) ∧
  demo(providerAgent,ppmp,'ppmp':'pp'#'bPrice'(ISBN,Price))).
```

Service Advertising Server

```
agentCapability(publisherAgent, 'pmp':'p'#'bCover'(ISBN,Cover)).
agentCapability(providerAgent, 'ppmp':'pp'#'bPrice'(ISBN,Price)).
```

Fig. 4 The MMP and MP programs for the demonstration

```
?- demo(browser,_,'rdf'#'type'(X,'bmp':'b'#'GeneticProgramming')).
   X = 'Genetic Algorithm'
?- demo(browser,_,'dmp':'d'#'bookInfo'('Genetic Algorithm',Cover,Price)).
   Cover = 'hard', Price = '$40'
```

Fig. 5 Query answering with the multi-agent communication

VIII. Implementation Issues

To state an ontology location in our framework, in the following we give an example of how the declaration looks like (see section VII) in OWL as follows:

```
<owl:Ontology rdf:about="dmp">
  <OntologyReferences>
    <Ontology rdf:resource="pmp">
      <path>/</path>
      <file>PublicationOnto.owl</file>
      <port>80</port>
      <protocol>http</protocol>
    </Ontology>
    <Ontology rdf:resource="bmp">
      <agentName
        rdf:resouce="providerAgent"/>
      <path>/</path>
      <file>PubProviderOnto.owl</file>
      <port>80</port>
      <protocol>http</protocol>
    </Ontology>
  </OntologyReferences>
</owl:Ontology>
```

After the transformation, we get an **MMP** fragment:

```
meta_info_statement(dmp,bookshopAgent,80,
  http,location('/','DocOnto.owl')).
meta_info_statement(pmp,_,80,http,
  location('/','PublicationOnto.owl')).
meta_info_statement(bmp,providerAgent,80,
  http,location('/','PubProviderOnto.owl')).
```

This kind of declaration is used throughout the paper to support the meta-information concerning ontology locations.

IX. Related Works

Some works investigated a multi-agent system adopting SW ontologies. In [3], Serafini et. Tamilin proposed a distributed reasoning architecture for SW using Distributed Description Logic (DDL) to formulate multiple ontologies interconnected by semantic mappings and a tableau method for performing inference in DDL. To compare it with our work, here we use meta-logic to represent SW ontologies, and a `demo(.)` predicate to perform the inference. We also formulate the predicate to be able to reason with ontology and agent locations in order to perform multi-agent communication for SW.

X. Conclusion

We have developed a meta-logical framework for agent communication of SW information. Our agent can reason with distributed ontologies while exchanging the SW information with other agents. The agent can do this by adopting a `demo` predicate which can reason with ontology and agent locations.

References

[1] Hirankitti, V., and Tran, X. V. Meta-reasoning with Multiple Distributed Ontologies on the Semantic Web. To appear in *Proc. of the 6th Int. Conf. on Intelligent Technologies*, December 2005.

[2] Hirankitti, V., and Tran, X. V. A Meta-logical Approach for Multi-agent Communication of the Semantic Web Information. In *Proc. of the 16th International Conference on Applications of Declarative Programming and Knowledge Management*, Japan, October 2005, pp. 7-16.

[3] Serafinil, L., and Tamilin, A. DRAGO: Distributed Reasoning Architecture for the Semantic Web. In Proc. of the 2nd European Semantic Web Conf.. LNCS, Vol. 3532, Springer-Verlag, pp. 361-376, 2005.

[4] Zou, Y., Finin, T., Ding, L., Chen, H., and Pan, R. Using Semantic Web technology in Multi-Agent systems: a case study in the TAGA Trading agent environment. In Proc. of the 5th Int. Conf. on Electronic Commerce. ACM Press, pp. 95-101, 2003.
[5] Grimnes, G. A., Chalmers, S., Edwards, P., and Preece, A. GraniteNights - A Multi-agent Visit Scheduler Utilising Semantic Web Technology. In Proc. of the 7th Cooperative Information Agents. LNCS, Vol. 2782, Springer-Verlag, pp. 137-151, 2003.
[6] Payne, T. R., Singh, R., and Sycara, K. Processing Schedules using Distributed Ontologies on the Semantic Web. In Proc. of the Int. Workshop on Web Services, E-Business, and the Semantic Web. LNCS, Vol. 2512, Springer-Verlag, pp. 203-212, 2002.
[7] Kowalski, R. A., and Kim, J. S. A Metalogic Programming Approach to Multi-agent Knowledge and Belief. In AI and Mathematical Theory of Computation, 1991, pp. 231-246.