International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:1, No:7, 2007

Meta-reasoning for Multi-agent Communication of Semantic Web Information

Visit Hirankitti and Vuong Tran Xuan

***Abstract*—**Meta-reasoning is essential for multi-agent communication. In this paper we propose a framework of multi-agent communication in which agents employ meta-reasoning to reason with agent and ontology locations in order to communicate semantic information with other agents on the semantic web and also reason with multiple distributed ontologies. We shall argue that multi-agent communication of Semantic Web information cannot be realized without the need to reason with agent and ontology locations. This is because for an agent to be able to communicate with another agent, it must know where and how to send a message to that agent. Similarly, 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. 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. In multi-agent communication of SW information, agents can employ meta-reasoning to communicate semantic information with other agents on the SW and also reason with multiple distributed ontologies. This meta-reasoning employs some meta-information to reason with agent and ontology locations. This is because for an agent to communicate with another agent, it must know where and how to send a message to that agent. Similarly, for an agent to reason with an external SW ontology, it must know where and how to access to that ontology. To achieve this, in this paper we propose a meta-logical model of SW communication among agents using meta-information of agent and ontology locations.

Some previous works on an agent system related to SW are: Zou et. al. [6] used SW languages, as the languages for expressing agent's messages and knowledge base, to specify and publish common ontologies; [7] presented a multi-agent based scheduling application in which data sources are described by SW languages and encapsulated in the agents. In [8], 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, in this paper we are concerned with multi-agent communication and reasoning with distributed ontologies and some previous works [1, 2, 3, 4] were published elsewhere; we have extended the framework in [3] to reason with agent and ontology locations. This paper is a revision and extension of [4].

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

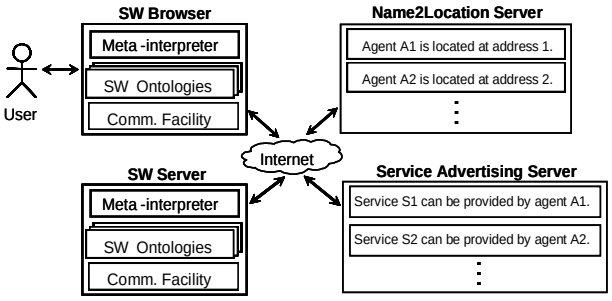

Fig. 1 Our SW multi-agent communication system

Manuscript received January 25, 2006. 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: v_hirankitti@yahoo.com).

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).

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 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 an 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.

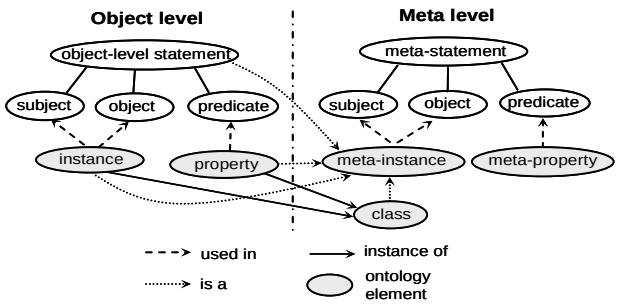

Fig. 2 The elements of an SW ontology at the object and meta levels

At the object level, an instance can be an individual or a literal of a domain, e.g. `'john'`, and a property is a relationship between individuals or an individual's attribute, e.g. `'hasFather'`, `'name'`. At the meta-level, a meta-instance can be an individual, a property, a class, and an object-level statement. A meta-property is a property to describe a meta-instance's attribute or a relationship between and among meta-instances, e.g. `'subClassOf'`, `'equivalentClass'`, `'equivalentProperty'`, etc.

Notice that according to the SW convention, to make a name appearing in an ontology unique, we qualify it with a namespace like `<namespace>:<name>`, for example, `'f':'M01'`, `'f':'hasFather'`, `'f':'hasChild'`, `'owl':'Class'`, `'rdfs':'Property'`, etc. Henceforth this qualified name will be used throughout.

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** also 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. A reference is a meta-constant of **MML**. This indicates that **ML** and **MML** are not entirely separated.

Meta-variable stands for a different meta-constant at a different time, e.g. `Person`. The first letter of a meta-variable name must be a capital letter.

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. 'o'#'f':'fatherOf'('o'#'f':'M2', 'o'#'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))).

In the above example, 'o'#'f':'fatherOf'(F, Ch)← demo(b, ob, 'ob'#'p':'parentOf'(F, Ch))∧demo(c, oc, 'oc'#'m':'male'(F)) is a rule expressing an object-level sentence which becomes a meta-representation (meta-term) to be manipulated at the meta-level by a meta-interpreter.

- **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. The first letter of a meta-variable name must be a capital letter.

Meta-function symbol standing for a logical connective—e.g. '←', '∧'—, or port, protocol, path, file, location (for meta-information of agents and ontologies), or '#', ':' (for namespace labeling), or a name of set operators applied on classes—e.g. union, intersection, and complement—, or a meta-predicate name being a name of a relation between entities, or being a name of characteristic of a property, which may fall into one of the following categories:

Class-class relations: subclass of, equivalent class of, disjoint with, etc.

Class-instance relations: instance of, class of, etc.

Property-property relations: sub property of, inverse of, etc.

Relations between literals and instances/classes/properties. We can find these relations as attributes of instances, of classes, or of properties, e.g. comment etc.

Characteristics of properties: symmetric, transitive, functional, etc.

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'.

When a meta-term expresses a meta-level predicate stating a relation between entities, it has the form of **Pred(Sub,Obj)**; and when it expresses a meta-level predicate stating a characteristic of a property, it has the form of **Pred(Prop)**, where **Pred** is a meta-predicate name, **Sub**, **Obj**, and **Prop** (a property) are meta-constants or meta-variables.

The meta-term expressing a *meta-level sentence* is a term **Pred(Sub,Obj)** or **Pred(Prop)** or a logical-connective function symbol applied to the tuple of these terms. Let all meta-variables appearing in the meta-level sentence be universally quantified. One form of the sentence is a Horn-clause **meta-rule**, e.g.

'owl':'equivalentClass'(C, EC) ←
  'owl':'equivalentClass'(C, EC1) ∧
  'owl':'equivalentClass'(EC1, EC).

Meta-statement being a meta-representation of a meta-level sentence to be accessible by the meta-interpreter. It has two forms: **meta_statement(meta-level-sentence)** and **axiom(meta-level-sentence)**; the latter form represents a rule for a mathematical axiom. Here are some examples of the meta-statements:

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. Notice that we put `T` as the first argument in `statement(T, meta-statement)` to signify that `meta-statement` belongs to the ontology (or theory) `T`.

- **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 are some typical elements of an **MMP** program:

Some meta-statements about classes and their relationships:

```
meta_statement(T, 'rdfs':'subClassOf'(
  C, SC) ← true).
```

// The class C is sub-class of the class SC.

```
meta_statement(T, 'owl':'equivalentClass'(
  C, EC) ← true).
```

// Classes C & EC are equivalent.

```
meta_statement(T, 'owl':'disjointWith'(
  C, DC) ← true).
```

// Classes C & DC are disjoint.

```
meta_statement(T, 'owl':'unionOf'(
  C, Cs) ← true).
```

// The class C is union of classes in Cs.

```
meta_statement(T, 'owl':'intersectionOf'(
  C, Cs) ← true).
```

// The class C is intersection of classes in Cs.

```
meta_statement(T, 'owl':'complementOf'(
  C, CC) ← true).
```

// The class C is complement of the class CC.

```
meta_statement(T, 'rdf':'type'(I, C) ← true).
```

// The instance I is an instance of the class C.

…

Some meta-statements about properties and their relationships:

```
meta_statement(T, 'rdfs':'subPropertyOf'(
  P, SP) ← true).
```

// The property P is sub-property of the property SP.

```
meta_statement(T, 'owl':'equivalentProperty'(
  P, EP) ← true).
```

// Properties P & EP are equivalent.

```
meta_statement(T, 'owl':'symmetric'(P) ← true).
```

// The property P is symmetric.

```
meta_statement(T, 'owl':'transitive'(P)← true).
```

// The property P is transitive.

```
meta_statement(T, 'owl':'functional'(P)← true).
```

// The property P is functional.

```
meta_statement(T, 'owl':'inverseFunctional'(P)
  ← true).
```

// The property P is inverse functional.

```
meta_statement(T, 'owl':'inverseOf'(P, IP)
  ← true).
```

// The property P is inversion of the property IP.

```
meta_statement(T, 'rdfs':'domain'(P, D)← true).
```

// The domain of the property P is D.

```
meta_statement(T, 'rdfs':'range'(P, R) ← true).
```

// The range of the property P is R

…

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

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

AMP contains axioms for classes and properties. They are expressed in the meta-rule form. Here are some typical elements of the **AMP**.

// the following relations of classes and properties are transitive.

```
axiom(T, 'owl':'equivalentClass'(C, EC) ← (atec)
  'owl':'equivalentClass'(C, EC1) ∧
  'owl':'equivalentClass'(EC1, EC)).
axiom(T, 'rdfs':'subClassOf'(C, SC) ←      (atsc)
  'rdfs':'subClassOf'(C, SC1) ∧
  'rdfs':'subClassOf'(SC1, SC)).
axiom(T, 'owl':'equivalentProperty'(P, EP)
←(atep)
  'owl':'equivalentProperty'(P, EP1) ∧
  'owl':'equivalentProperty'(EP1, EP)).
axiom(T, 'rdfs':'subPropertyOf'(P, SP) ←     (atsp)
  'rdfs':'subPropertyOf'(P, SP1)) ∧
  'rdfs':'subPropertyOf'(SP1, SP)).
axiom(T, 'owl':'sameAs'(I, SI) ←           (atsa)
  'owl':'sameAs'(I, SI1) ∧
  'owl':'sameAs'(SI1, SI)).
```

…

// the following relations of classes and properties are symmetric.

```
axiom(T, 'owl':'equivalentClass'(C, EC) ← (asec)
  'owl':'equivalentClass'(EC, C)).
axiom(T, 'owl':'disjointWith'(C, DC) ← (asdc)
  'owl':'disjointWith'(DC, C)).
axiom(T, 'owl':'equivalentProperty'(P,EP)← (asep)
  'owl':'equivalentProperty'(EP, P)).
axiom(T, 'owl':'inverseOf'(P, IP) ← (asip)
  'owl':'inverseOf'(IP, P)).
axiom(T, 'owl':'sameAs'(I, SI) ← (assa)
  'owl':'sameAs'(SI, I)).
axiom(T, 'owl':'differentFrom'(I, DI) ← (asdf)
  'owl':'differentFrom'(DI, I)).
…
// some axioms are related to characteristics of a property
axiom(T, P(S, O) ← (acip)
  'owl':'inverseOf'(P, IP) ∧ IP(O, S)).
axiom(T, P(S, O) ← (acsp)
  'rdfs':'subPropertyOf'(SP, P) ∧ SP(S, O)).
axiom(T, P(S, O) ← (acep)
  'owl':'equivalentProperty'(P, EP) ∧
  EP(S, O)).
axiom(T, P(S, O) ← (actp)
  'owl':'transitive'(P) ∧
  P(S, O1) ∧ P(O1, O)).
axiom(T, P(S, O) ← (acsmp)
  'owl':'symmetric'(P) ∧ P(O, S)).
…
```

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 into MPs, MMPs. The Knowledge base already stores AMPs and is later added with the transformed MPs and MMPs. The Meta-interpreter reasons with the three kinds of meta-programs in order to answer a query posed by the user, and communicates with other agents to get ontologies or answers for a query. Historical memory stores information required for advance reasoning by the meta-interpreter, such as backtracking between alternative answers derived from several agents. The Communication module facilitates communication with other agents and users.

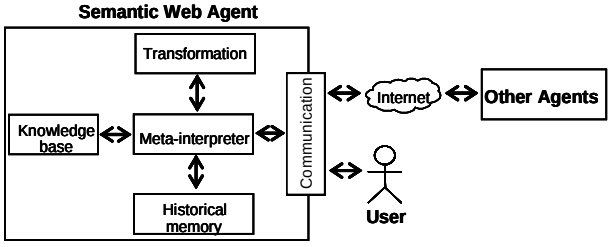

Fig. 3 Single SW Agent Architecture

- **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 [9] is used for our meta-interpreter. Our `demo` definition, which can reason with multiple distributed ontologies and communicate with other agents proposed in [1, 2, 3, 4], has been extended here to reason with agent and ontology 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 [3] 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), (conj), and (ost) 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) ∧
  T#NS:Goal = A ∧
  meta_info_statement(
    T, Agent'', Port, Channel, Location) ∧
  retrieve(T, Agent'', Port, Channel,Location) ∧
```

```
  transform(T, P) ∧ demo(Agent, P, A).
```

With this clause, `demo` can work analogously to a web browser. `meta_info_statement(T, Agent'', Port, Channel, Location)` returns `Agent''`, `Port`, `Channel`, and `Location` as the meta-information of the location of ontology `T`. Note that every remote ontology that is referred and used in an ontology will be declared together with its location in that ontology, see §VIII, and this information will be transformed into this meta-information statement.

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).
```

For the `demo` clause `(certain-agent-comm)`, `demo` will ask another agent (not itself) `Agent'`, already known, to answer question `A` from a theory possessed by `Agent'`. To communicate with `Agent'`, `demo` must find its location on the network using the meta-information given by `agentLocation(Agent', Location, Port, Channel)`. After the location is known, `demo` establishes the connection to that location and asking `Agent'` for an answer for question `A`. If it successes `demo` obtains the answers and finally terminate the communication with `Agent'`.

```
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).
```

For the last `demo` clause, `demo` tries to get an answer for question `A` from any agent who can answer it. In this case, `demo` uses `findAgent(Agent', A)` to search for an agent who can provide an answer (service) by consulting the Service Advertising Server (`sa_server`) which is another agent. When this applicable agent is known, `demo` will call itself so that the `(certain-agent-comm)` clause will be employed later.

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, it knows that it may ask the `providerAgent` server. For the first case, the bookshop server has to consult to the Service Advertising Server to find the location and service of the server and to post its corresponding queries to it and get the answers back. The bookshop server then returns all the answers to the SW browser to present to the user.

Publish 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 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 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 multi-agent communication

VIII. ONTOLOGY REPRESENTATION AND TRANSFORMATION

To allow an SW agent to reason with referenced ontologies and communicate with other agents, an ontology the agent possesses must contain meta-information of locations of all ontologies referred to in this possessed ontology as well as contain the names of all the agents which will provide inferential results under provability adopted in this possessed ontology. In this section we give an example to show how a rule declared in the SWRL form (slightly different from the conventional SWRL form) in an ontology, and its resulting rule after the transformation look like, see Fig. 6. Then we show how we declare, in OWL, the meta-information concerning locations of referenced ontologies that are referred to in an ontology and the result of the transformation in Fig. 7. Notice that information of the locations of all referenced ontologies are asserted by predicate `meta_info_statement()`. In this case there are three referenced ontologies referred to in this ontology. These kinds of declarations are used throughout the paper to support the rule declaration and the meta-information concerning ontology locations.

```
// Rule declation
<swrl:Imp rdf:ID="get-BookInfo">
  <swrl:head>
    <swrl:AtomList>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:first>
        <swrl:IndividualPropertyAtom>
          <swrl:propertyPredicate rdf:resource="#bookInfo"/>
          <swrl:numArguments>3</swrl:numArguments>
          <swrl:argument1><swrl:Variable rdf:ID="Title"/></swrl:argument1>
          <swrl:argument2><swrl:Variable rdf:ID="Cover"/></swrl:argument2>
          <swrl:argument3><swrl:Variable rdf:ID="Price"/></swrl:argument3>
        </swrl:IndividualPropertyAtom>
      </rdf:first>
    </swrl:AtomList>
  </swrl:head>
  <swrl:body>
    <swrl:AtomList>
      <rdf:first>
        <swrl:IndividualPropertyAtom>
          <swrl:propertyPredicate rdf:resource="#bTitle"/>
          <swrl:numArguments>2</swrl:numArguments>
          <swrl:argument1><swrl:Variable rdf:ID="ISBN"/></swrl:argument1>
          <swrl:argument2 rdf:resource="#Title"/>
        </swrl:IndividualPropertyAtom>
      </rdf:first>
      <rdf:rest>
        <swrl:AtomList>
        <rdf:first>
            <swrl:IndividualPropertyAtom>
              <swrl:propertyPredicate rdf:resource="ppmp#pp:bCover"/>
              <swrl:numArguments>2</swrl:numArguments>
              <swrl:argument1 rdf:resource="#ISBN"/>
              <swrl:argument2 rdf:resource="#Cover"/>
```

```
            </swrl:IndividualPropertyAtom>
          </rdf:first>
          <rdf:rest>
            <swrl:AtomList>
            <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
            <rdf:first>
              <swrl:IndividualPropertyAtom>
                <swrl:propertyPredicate rdf:resource="pmp#p:bPrice"/>
                <swrl:numArguments>2</swrl:numArguments>
                <swrl:argument1 rdf:resource="#ISBN"/>
                <swrl:argument2 rdf:resource="#Price"/>
              </swrl:IndividualPropertyAtom>
            </rdf:first>
           </swrl:AtomList>
          </rdf:rest>
        </swrl:AtomList>
      </rdf:rest>
    </swrl:AtomList>
  </swrl:body>
</swrl:Imp>
```

// The meta-statement resulted from the transformation

```
statement(onto, 'dmp'#'d':'bookInfo'(Title, Cover, Price)
   if 'dmp'#'d':'bTitle'(ISBN, Title) and
      'ppmp'#'pp':'bCover'(ISBN, Cover) and
      'pmp'#'p':'bPrice'(ISBN, Price)).
```

Fig. 6 Rule declaration and the result after the transformation

// Declaration of meta-information concerning ontology locations

```
<owl:Ontology rdf:about="dmp">
  <o:OntologyReferences>
    <rdf:List>
      <rdf:first rdf:resource="bmp"/>
      <rdf:rest>
        <rdf:List>
          <rdf:first rdf:resource="pmp"/>
          <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
        </rdf:List>
      </rdf:rest>
    </rdf:List>
  </o:OntologyReferences>
</owl:Ontology>
<rdf:Description rdf:about="bmp">
  <o:protocol>http</o:protocol>
  <o:port>80</o:port>
  <o:path>/</o:path>
  <o:file>BookOntology.owl</o:file>
</rdf:Description>
<rdf:Description rdf:about="pmp">
  <o:protocol>http</o:protocol>
```

```
  <o:port>80</o:port>
  <o:path>/</o:path>
  <o:file>PublicationOntology.owl</o:file>
  <o:agentName rdf:resource="publisherAgent"/>
</rdf:Description>
```

// The meta-statements resulted from the transformation

```
meta_info_statement(bmp, _, 80, http, location(/, 'BookOntology.owl')).
meta_info_statement(pmp, publisherAgent, 80, http, location(/, 'PublicationOntology.owl')).
```

Fig. 7 Meta-information declaration of the locations of referenced ontologies and the result after the transformation

IX. RELATED WORKS

Some works investigated a multi-agent system adopting SW ontologies. In [5], 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 agent and ontology locations in order to perform multi-agent communication for SW.

X. CONCLUSION

We have developed a meta-logical framework for multi-agent communication of SW information. Our agent performs meta-reasoning in order to 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 meta-information of agent and ontology locations.

REFERENCES

[1] Hirankitti, V., and Tran, X. V. A Meta-logical Approach for Reasoning with Semantic Web Ontologies. In proc. of the 4th IEEE International Conference on Computer Sciences: Research, Innovation & Vision for the Future, February 2006, Viet Nam, pp. 228-235.

[2] Hirankitti, V., and Tran, X. V. Meta-reasoning with Multiple Distributed Ontologies on the Semantic Web. In proc. of the 6th International Conference on Intelligent Technologies, Thailand, December 2005, pp. 301-309.

[3] 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.

[4] Hirankitti, V., and Tran, X. V. Semantic Web Agent Communication Capable of Reasoning with Ontology and Agent Locations. In proc. of the 8th International Conference on Cybernetics, Informatics, and Systemics, Poland, December 2005, pp. 98-104.

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

[6] 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 International Conference on Electronic Commerce. ACM Press, pp. 95-101, 2003.

[7] 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.

[8] Payne, T. R., Singh, R., and Sycara, K. Processing Schedules using Distributed Ontologies on the Semantic Web. In proc. of the International Workshop on Web Services, E-Business, and the Semantic Web. LNCS, Vol. 2512, Springer-Verlag, pp. 203-212, 2002.

[9] 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.