A *Q*-Based Architecture for Semantic Information Interoperability on Semantic Web

Zhen-jie Wang, Huan-ye Sheng, and Peng Ding

Department of Computer Science and Engineering,
Shanghai Jiao Tong University, 200030 Shanghai, China
{wang-zj,sheng-hy,dingpeng}@cs.sjtu.edu.cn

Abstract. Semantic Web supports a fire-new infrastructure for solving the problem of semantic information interoperability, and it promises to support an intelligent and automatic information-processing platform for multi-agent system whose ultimate objective is to provide better services for end-users, for example, interoperable information query. Therefore, except agent-to-agent interaction in multi-agent system, there is human-to-agent interaction. To unify the two kinds of interaction, this paper introduces *Q* language - a scenario description language for designing interaction among agents and users. A *Q*-based architecture, which integrates ontology servers, ontology-mapping servers, semantic information sources, and multi-agent query system, is presented as a system solution to semantic information interoperability on Semantic Web. Furthermore, we investigate key technologies of interoperability: domain ontology, ontology-mapping service, and related multi-agent system, and give an implementation to demonstrate how our architecture works.

1 Introduction

On Semantic Web, a kind of new profiles including semantic information will be used to weave a fire-new infrastructure with machine-readable and machine-interpretable information [1], which supports many new functions, such as interoperability, fusion and integration of information. The idea for achieving information interoperability lies in that the meanings of the interchangeable information should be understood across the entire information infrastructure. A key basis of information interoperability is ontology, which makes it possible to abstract knowledge model of real world using concepts and annotate semantic information into web information sources. Obviously, domain ontology and annotated semantic information sources intend to improve information query and address semantic information interoperability. Semantic Web endowed with many ontology languages such as XML, RDF/S, and DAML+OIL [2,3,4], etc. These languages are foundation for ontology-based information query that will provide more effective information query service.

Because same information is widely applied in many domains, and users have different views for conceptualizing them, there must be different definitions of concepts and relationships for same information. To implement information interoperability, concept-switching or concept-transformation function between different domain ontology is necessary.

Additionally, Semantic Web promises to support a semantic information-processing platform for multi-agent systems, on which agents will be first-class citizens. Therefore, besides domain ontology component and ontology-mapping component, another necessary component is multi-agent system that can automatically retrieve and manipulate semantic information for end-users. Thus, one of the most necessary requirements placed on agents is the capability to interact with end-users. To unify agent-to-agent and human-to-agent interaction, we introduce Q language - a scenario description language for describing interaction among agents and end-users [5].

The paper is organized as follows. Section 2 introuduces syntax facilities of Q language. In section 3, we present a Q-based architecture for implementing semantic information interoperability on Semantic Web, and then discuss its crucial components. Section 4 gives a prototype implementation of proposed Q-based architecture. Lastly, we conclude the paper and discuss related works.

2 *Q* Language Overview

Some inter-agent protocol description languages, such as KQML and AgenTalk [6], often regulate an agent's various actions on the basis of computational model of agent internal mechanisms. Obviously,these agents based on strict computational models have to be designed and developed by computer experts. It is necessary to design a new agent interaction description language, which makes it possible that those non-computer-professional application designers, such as sale managers and consumers, might write scenarios that describe and model the behaviors of agents, so that a practical multi-agent system can be easily established. Under the background, we start working on Q language - a scenario description language for designing interaction among agents and humans. This kind of scenario-based language is to design interaction from the viewpoint of scenario description, but not for describing agent internal mechanisms.

2.1 Syntax Facilities

Q extends Scheme by introducing sensing/acting functions and guard command to realize scenario description. Why Scheme [7] becomes Q's mother language is in that its Lisp-like characteristic: programs (here scenarios) can be handled as data. The basic facilities of Q language for scenario description include Cue, Action, Guard Command, Scenario, and Agent. Execution architecture of Q scenario consists of Execution Layer and Meta Layer [5].

Cue. A sensing function is defined as a cue, which represents agent's perception to its outside environment. A cue doesn't produce any side effect on the environment. The syntax of cue definition and its example are as follows.

```
(defcue cue-name {(parameter in|out|inout)}*)
(defcue ?receive (:sentence in) (:from out))
(?receive request :from user)
```

Action. An acting function is defined as an action, which may change and impose effects on the environment of agent system. The syntax and example of an action definition are showed below.

```
(defaction action-name {(parameter in|out|inout)}*)
(defaction querydaml (:damlquery in) (:queryresult out))
(!querydaml $queryrule $result)
```

Guard Command. A guard command is used for describing an agent's behavior control mechanism to await multiple cues. If one of cues is perceived, corresponding "form" will be performed. If no cue is perceived, the guard command will perform the "therwise" clause. The syntax and example of Guard Command are showed as follows.

```
(guard {(cue {form}*)* [(otherwise{form}*)]})
(guard ((?hear "Hello" Peedy) (!play "Greet")
                        (!speak "Hi, Nice to meet you"))
       ((?see-select 0) (!speak "Please input a query!")
                        (!askqueryinput $queryrule))
       (otherwise (!move 100 200) (!play "Wave")
                        (!speak "Bye-Bye")))
```

Scenario. A scenario is used for defining several different states represented by state1, state2, etc.

```
(defscenario scenario-name ({variable}*)
   (state1 {(cue {form}*)}*
           [(otherwise {form}*)])
   (state2 {(cue {form}*)}*
           [(otherwise {form}*)]))
```

An example of scenario is represented as follows, in which state transition is implemented by "form" (go state) between different states (state1, state2, etc.).

```
(defscenario query-agent-scenario (message)
   (let(($x #f))
      (state1((?equal $x Peedy)(!say message)(go state2))
             (otherwise (!say "Hello")(go state3)))
      (state2((guard
             ((?hear "Hello" Peedy)(!play "Greet")
                                   (!speak "Hi") (go state4))
             ((?see-select 0)(!askqueryinput $queryrule)
                                               (go state5))
             (otherwise (!move 100 200)(!play "Wave")
                        (!speak "Bye-Bye")))))))
```

Agent. Agent is defined together with a specified scenario that will be executed by the specified agent. An agent is defined as follows.

```
(defagent agent :scenario scenario-name {key value}*)
(defagent query-agent  query-agent-scenario
   :ip_address "192.168.100.188"
   :port_number 8080)
```

Conclusively, the most prominent property of *Q* Language is its simplicity on both design and application of agent systems with following characteristics: End-user-oriented, focusing on the interaction behavior of agent from the viewpoint of users, not agent; Error-allowed (no requirement for correctness); The complex behavior of agent can be realized by the combination of scenarios.

3 Semantic Information Interoperability

The basic architecture of semantic information interoperability on Semantic Web is described in Fig. 1. It is composed of end-users, agent system, ontology sources, and semantic information sources four parts. Agent system consists of user agent, query agent, wrapper agent, and facilitator agent. The interaction among these agents and interaction between end-user and agent system are described by a library of *Q* scenarios, each of which control an agent's behaviors and a group of which specify a multi-agent cooperation strategy to cope with a specific request. Ontology sources are ontology-mapping servers and domain ontology servers. Semantic information sources include web pages and web services and web databases, in which information providers have annotated semantic information using related domain ontology.

3.1 Domain Ontology

Ontology is generally defined as "a formal explicit specification of a shared conceptualization" [8], which is often used to abstract knowledge models of real world. Many advanced ontology languages, such as RDF/S, OIL and DAML+OIL, make it possible to describe domain ontology and annotate semantic information into web sources. Obviously, domain ontology that conceptualizes domain knowledge and corresponding semantic information sources intend to improve information query and address semantic information interoperability.

The decentralized characteristic of Semantic Web determines that users themselves are able to construct a large number of small domain ontology in much the same way that today's Web content is created [1]. These constructed domain ontology may be distributed in ontology servers of any physical web

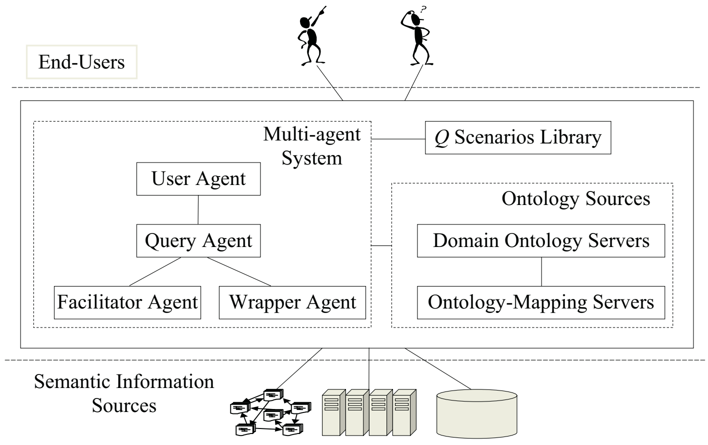


Fig. 1. Architecture for semantic information interoperability

sites. To same information applied in different domains, users have different views for conceptualizing them, so even if all domains' ontology is described in a common ontology language, there still exist different definitions of concepts and relationships for same objects. To implement information interoperability, concept-switching or concept-transformation function between different domain ontology is necessary. By different ontology definitions about the same objects Park and Service Advertisement respectively, we will clearly observe this necessary about concept transformation. These examples are represented by ontology language DAML+OIL.

The first example is about park information. One is about the definition of Park Ontology of Travel Domain in http://www.ichi.sjtu.edu.cn/Travel/Park-Ont. The other is about the definition of Garden Ontology of Administration Domain in http://www.Travel-Adm.com/Landscape/Garden-Ont.

```
Park Ontology:
<daml:Class ID="Park">
   <rdfs:subClassOf>
       <daml:Restriction daml:cardinality="1">
          <daml:onProperty rdf:resource="#Name"/>
       </daml:Restriction>
   </rdfs:subClassOf>
</daml:Class>
<daml:DatatypeProperty rdf:ID="Name">
   <rdfs:domain rdf:resource="#Park"/>
   <rdfs:range rdf:resource="#string"/>
</daml:DatatypeProperty>
<daml:DatatypeProperty rdf:ID="City">
   <rdfs:domain rdf:resource="#Park"/>
   <rdfs:range rdf:resource="#string"/>
</daml:DatatypeProperty>
<daml:DatatypeProperty rdf:ID="Country">
   <rdfs:domain rdf:resource="#Park"/>
   <rdfs:range rdf:resource="#string"/>
</daml:DatatypeProperty>
```

```
Garden Ontology:
<daml:Class ID="Garden">
   <rdfs:subClassOf>
      <daml:Restriction daml:cardinality="1">
         <daml:onProperty rdf:resource="#Name"/>
      </daml:Restriction>
   </rdfs:subClassOf>
</daml:Class>
<daml:DatatypeProperty rdf:ID="Name">
   <rdfs:domain rdf:resource="#Garden"/>
   <rdfs:range rdf:resource="#string"/>
</daml:DatatypeProperty>
<daml:DatatypeProperty rdf:ID="Place">
   <rdfs:domain rdf:resource="#Garden"/>
   <rdfs:range rdf:resource="#string"/>
</daml:DatatypeProperty>
```

The second example is about service advertisement of web service. One is about ServiceProfile Ontology of Service Domain in http://www.daml.com/Service/Profile-Ont. The other is about ServiceAdvertisement Ontology of Agent Domain in http://www.ichi.sjtu.edu.cn/Agent/Advertisement-Ont.

```
ServiceProfile Ontology:
<daml:Class ID="ServiceProfile">
   <daml:subClassOf rdf:resource="#Profile"/>
</daml:Class>
<daml:DatatypeProperty rdf:ID="ServiceName">
   <rdfs:domain rdf:resource="#ServiceProfile"/>
   <rdfs:range rdf:resource="#string"/>
</daml:DatatypeProperty>
<daml:DatatypeProperty rdf:ID="ServiceType">
   <rdfs:domain rdf:resource="#ServiceProfile"/>
   <rdfs:range rdf:resource="#Classification"/>
</daml:DatatypeProperty>
<daml:Class rdf:ID="Classification">
   <daml:oneOf rdf:parseType="daml:collection">
      <Type rdf:ID="Database-Query"/>
      <Type rdf:ID="E-Commence"/>
      <Type rdf:ID="Language-Translation"/>
   </daml:oneOf>
```

```
</daml:Class>
<daml:DatatypeProperty rdf:ID="Input">
   <rdfs:domain rdf:resource="#ServiceProfile"/>
   <rdfs:range rdf:resource="#string"/>
</daml:DatatypeProperty>
<daml:DatatypeProperty rdf:ID="Output">
   <rdfs:domain rdf:resource="#ServiceProfile"/>
   <rdfs:range rdf:resource="#string"/>
</daml:DatatypeProperty>

ServiceAdvertisement Ontology:
<daml:Class ID="ServiceAdvertisement">
   <daml:subClassOf rdf:resource="#Profile"/>
</daml:Class>
<daml:DatatypeProperty rdf:ID="ServiceName">
   <rdfs:domain rdf:resource="#ServiceAdvertisement"/>
   <rdfs:range rdf:resource="#string"/>
</daml:DatatypeProperty>
<daml:DatatypeProperty rdf:ID="ServiceType">
   <rdfs:domain rdf:resource="#ServiceAdvertisement"/>
   <rdfs:range rdf:resource="#Category"/>
</daml:DatatypeProperty>
<daml:Class rdf:ID="Category">
   <daml:subClassOf rdf:resource="#Classification"/>
</daml:Class>
<daml:DatatypeProperty rdf:ID="Precondition">
   <rdfs:domain rdf:resource="#ServiceAdvertisement"/>
   <rdfs:range rdf:resource="#string"/>
</daml:DatatypeProperty>
<daml:DatatypeProperty rdf:ID="Postcondition">
   <rdfs:domain rdf:resource="#ServiceAdvertisement"/>
   <rdfs:range rdf:resource="#string"/>
</daml:DatatypeProperty>
```

3.2 Ontology-Mapping Service

Furthermore, we discuss how to implement concept transformation through two concrete query examples. Supposed a user in Administration Domain wants to search for information about Yu Yuan Garden, and he sends Query A via a user agent to a query agent that will assist user in searching for information. A specific notation is used to identify concepts and specify ontology in users' query, for example, Query A is formalized by Garden Ontology with properties Name and Place. SEARCH <Administration:Garden> means that this user expects to search for objects of class Garden including all properties that are defined for this class, and searching results must be restricted to those objects with properties Name "Yu Yuan Garden" and Place "Shanghai, China".

```
Query A:
xmlns:Administration="http:// www.Travel-Adm.com /
                                        Landscape/Garden-Ont#">
SEARCH <Administration:Garden>
WHERE <Administration:Name>Yu Yuan Garden</Administration:Name>
      <Administration:Place>Shanghai,China</Administration:Place>
END
```

When the query agent browses a web page annotated with Park Ontology (see Annotation A), intuitively, if it is aware of equivalent mapping relation between property Place in Park Ontology of Travel Domain and properties City as well as Country in Garden Ontology of Administration Domain, it can find a object which includes information of Yu Yuan Garden in Shanghai of China.

```
Annotation A:
xmlns:Travel="http://www.ichi.sjtu.edu.cn/Travel/Park-Ont#">
<Travel:Park rdf:ID="YuYuan">
   <Travel:Name> Yu Yuan Garden</Travel:Name>
   <Travel:City> Shanghai</Travel:City>
   <Travel:Country> China </Travel:Country>
   <Travel:Area> 30 Acres </Travel:Area>
</Travel:Park>
```

In another example, a user in Agent Domain searches for information about a search engine, he sends Query B via a user agent to a query agent. Query B is formalized by ServiceAdvertisement Ontology. SEARCH <Agent:ServiceName> IN <Agent:ServiceAdvertisement> means that this user expect to search for property ServiceName in class ServiceAdvertisement, and searching results must be restricted to those objects with properties ServiceType "Language-Translation" and Postcondition "Web pages".

```
Query B:
xmlns:Agent="http://www.ichi.sjtu.edu.cn/
                                  Agent/Advertisement-Ont#">
SEARCH <Agent:ServiceName>
IN <Agent:ServiceAdvertisement>
WHERE <Agent:ServiceType rdf:resource="#Language-Translation"/>
      <Agent:Postcondition> Web Pages </Agent:Postcondition>
END
```

When the query agent contacts with facilitator agent, it acquires a service profile annotated with ServiceProfile Ontology (see Annotation B). If it is aware of mapping relations between ServiceAdvertisement Ontology of Agent Domain and ServiceProfile Ontology of Service Domain, for example, an equivalent mapping between properties Postcondition and Output, plus to a subsuming mapping between properties ServiceType and ServiceType, it can find a service with ServiceName "Google" in object GoogleProfile satisfies searching conditions.

```
Annotation B:
xmlns:Service="http://www.daml.com/Service/Profile-Ont#">
<Service:ServiceProfile rdf:ID="GoogleProfile">
   <Service:ServiceName> Google </Service:ServiceName>
   <Service:ServiceType rdf:resource="#Language-Translation"/>
   <Service:Input> Text of sentences </Service:Input>
   <Service:Output> Web pages </Service:Output>
</Service:ServiceProfile>
```

From above query examples, we conclude that ontology-mapping service, which accepts or collects mapping advertisement information from semantic information sources, and then updates and adds it in the form of mapping rules on ontology-mapping server, is a necessary component to implement information interoperability. The forms of mappings contain three types:

– Ontology-to-ontology: it specifies an equivalent or subsuming mapping between source ontology and target ontology.
– Class-to-class: it specifies an equivalent or subsuming mapping between classes of source ontology and target ontology.
– Property-to-property: it specifies an equivalent mapping between properties of class in source ontology and properties of class in target ontology.

A mapping rule has the form *r: Operator(o.c.p)* ***Rlation*** *Operator(o.c.p)*, where r is the rule's label, *Operator* is logical operator (AND, OR, or NOT), o represents ontology name, $o.c$ is a class of a ontology, and $o.c.p$ is a property of a class in a ontology. In symbol *()*, a set of ontology, classes, or properties is contained. ***Relation*** represents equivalent or subsuming relationships that are symbolized as: $=$, $\subseteq$. For example, the following rules represent the mappings between properties of Garden Ontology and Park Ontology.

```
R1:http://www.Travel-Adm.com/Landscape/Garden-Ont.Garden.Place =
   AND (http://www.ichi.sjtu.edu.cn/Travel/Park-Ont.Park.City,
        http://www.ichi.sjtu.edu.cn/Travel/Park-Ont.Park.Country)
R2:http://www.Travel-Adm.com/Landscape/Garden-Ont.Garden.Name =
   http://www.ichi.sjtu.edu.cn/Travel/Park-Ont.Park.Name
```

3.3 Multi-agent System

Multi-agent system including user agent, query agent, facilitator agent, and wrapper agent is to assist users in query inference and transformation, and it is used to connect users, ontology servers, ontology-mapping servers, and various semantic information sources (see Fig. 2).

In interoperable information query, query agent is used to assist users in query processing, for example concept extracting, ontology-mapping rules acquiring, query rewriting, knowledge finding, etc. Its primary function modules are depicted in Fig. 2. When end-users issue a query that is described by specified ontology to his user agent, user agent will submit it to query agent. Firstly,

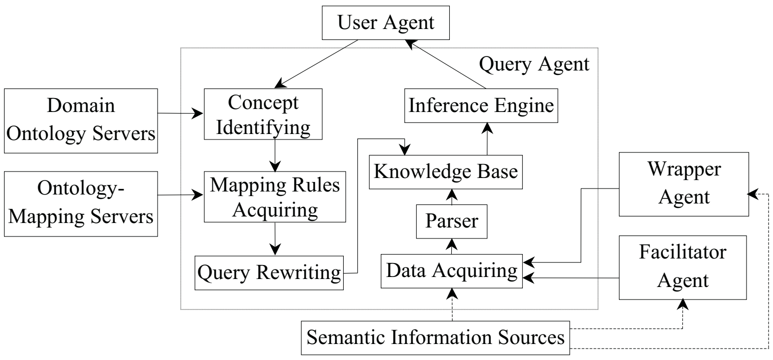


Fig. 2. Multi-agent query system

the query agent abstracts and identifies concepts in user's query; then it checks whether there are mappings between user ontology and ontology used by inspected semantic information. If query agent acquires ontology-mapping rules from ontology-mapping server, it will rewrite the user query. Parser is used to parse semantic data acquired from annotated semantic information sources. According to stored knowledge from Parser and rewritten query from Query Rewriting in Knowledge Base, Inference Engine will search and return corresponding query results to user agent.

If information source is a traditional Database, a wrapper agent is used to transform data in DB into semantic data according to its Database Schema. Here, wrapper agent acts as proxies for external information sources. Facilitator agent accepts semantic advertisements from services on Semantic Web, and matches service requests derived from a query agent.

In this multi-agent system, there is the interaction between users and user agent, for example, issuing query to user agent, returning results to user, reporting states of query; there are the interaction between query agent and user agent for transmission of query and results, the interaction between query agent and facilitator agent for searching service information, the interaction between query agent and wrapper agent for acquiring data from legacy system, and the interaction between query agent and domain ontology servers as well as ontology-mapping servers. All of interaction can be described by Q scenarios, which are used to control agents' behaviors and specify their cooperation strategy.

4 Implementation

The experimental environment consists of Q language, Microsoft Agent[1], Visual C++, and Prolog-based XSB[2]. The implementation architecture of query system is showed in Fig. 3. Q scenarios describe agents' behaviors (Cues and Actions)

[1] http://www.microsoft.com/msagent/.
[2] http://xsb.sourceforge.net/.

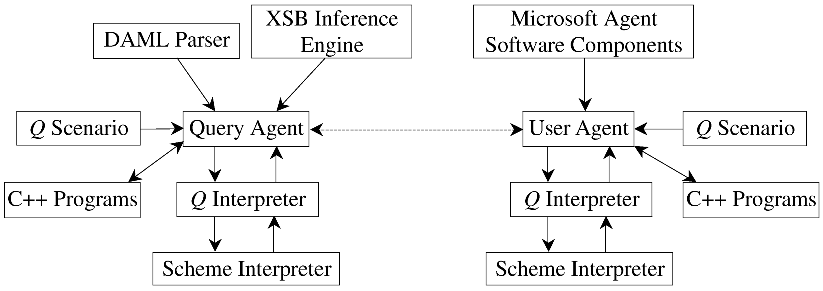


Fig. 3. Implementation architecture of query agent and user agent

Table 1. Main Cues and Actions of user agent

Cues and Actions	**Interpretation** (behaviors of user agent)
(?feel [:from user])	Checks whether to receive user's instruction, for example, quit, stop, etc.
(?receiveResult $Result [:from query-agent])	Checks whether to receive searching result from query agent.
(!speak Sentence [:to user])	Speaks to users, for example, query results.
(!play Animation)	Play some animations such as "GestureDown".
(!move X Y)	Moves to a point (x, y) in the screen with an appropriate manner, such as fly.
(!askqueryinput $Query[:from user])	Asks user to input query items, and variable $Query represents user's query.
(!submit $Requests [to: query-agent])	Submits user's requests to a query agent.

that are implemented by Visual C++. Designed on Microsoft Agent Software Components, our user agent is incorporated interactive abilities such as "speak", "play", "move", etc.

DAML+OIL is used to construct our domain ontology. Therefore, DAML Parser of query agent is designed to parse semantic annotation (DAML markup), and it is constructed on Repat[3]. According to RDF data model, this Parser should parse DAML annotation into triples, each of which includes three parts (Subject, Predicate, Object plus to their corresponding Namespace). These parsed triples are stored in Knowledge Base in a knowledge representation form that is consistent with XSB Inference Engine. As an efficient rule inference engine, XSB specifies ways of processing a pattern query and finding new knowledge from parsed semantic data.

Here, we will give two query examples mentioned in Query A and Query B. We assume that query agent directly acquires semantic information of "Yu Yuan Garden" from an annotated web page; in addition, it directly acquires semantic information of service "Google" from facilitator agent. We design user agent and query agent's main Cues and Actions as Table 1 and Table 2.

[3] http://www.daml.org/tools/repat.

Table 2. Main Cues and Actions of query agent

Cues and Actions	**Interpretation** (behaviors of query agent)
(?feel [:from user-agent])	Checks whether to get user agent's instruction.
(?receiveRequest $Query [:from user-agent])	Checks whether to get user agent's query request.
(?receiveRule $Mapping-rules [:from mapping-server])	Checks whether to receive ontology-mapping rules.
(!query-mapping $Mapping-rules [to: mapping-server])	Identifies concepts used in user query, and then queries mapping rules from ontology-mapping server.
(!query-rewriting $Mapping-rules $Query)	Rewrites this query according to mapping-rules.
(!parsedaml DAMLFile)	Parses a DAML file into triples.
(!querydaml $Query $Result)	Finishes Knowledge searching according to $Query, and stores query results in variable $Result.
(!return $Result [to: user-agent])	Returns query results in variable $Result to user agent.

The main states of user agent in a scenario are shown as follows. At the initial state state_Ask-query, user agent shows a dialog box and guides a user to input his query into the dialog box. After the user inputs the query, user agent's state shifts to state_Submit, in which the agent executes a action of sending a query to a query agent, and then user agent goes on shifting to state_MsgWait. In state_MsgWait, user agent waits for the user's instructions or the messages sent by query agent, and tells the user what is happening, and then shifts to the corresponding state. When user agent receives query results from query agent, it will speak results to user.

```
(state_Ask-query (otherwise
   (!speak " Hello! This is Agent Peedy!")
   (!play  "Greet")
   (!move 200 300)
   (!speak "Please input query in the dialog box!")
   (!askqueryinput $Query)  (go state_Submit)))
(state_Submit (otherwise
   (!submit $Query)(go state_MsgWait)))
(state_MsgWait
   ((?feel)(!speak "Please give me instructions!")
                            (go state_Instruction))
   ((?receiveResult $Result)
      (!speak "I receive results from query agent.")
      (!play "Read")
      (!move 300 450)
      (!speak $result) (go state_Ask-query))
   (otherwise
      (!speak "I am waiting message!" )(go state_MsgWait)))
```

The main states of query agent in a scenario are shown as follows. At the initial state state_MsgWait, query agent waits for the messages from user agent or ontology-mapping server, and then shifts to corresponding state. When query agent receives the query request from user agent, it shifts to state_Ask-rules, and acquires mapping rules from ontology-mapping server. After getting mapping rules, query agent shifts to state_Query for knowledge query.

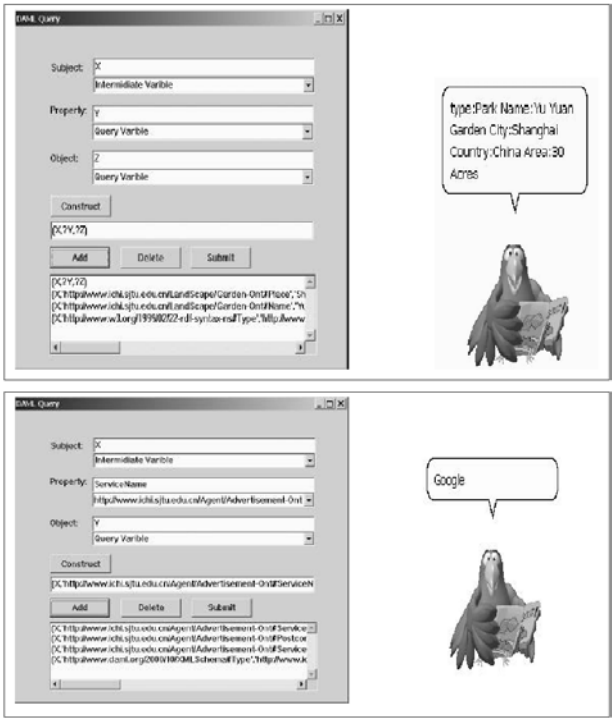
```
(state_MsgWait
   ((?feel)  (go state_Instruction))
   ((?receiveRequest $query)  (go state_Ask-rules))
   ((?receiveRule $Mapping-rules) (go state_Query)
   (otherwise  (go state_MsgWait)))
(state_Ask-rules (otherwise
   (!query-mapping $Mapping-rules ServerName)(go state_MsgWait)))
(state_Query (otherwise
   (!query-rewriting $Mapping-rules $Query)
   (!parsedaml DAMLFile)
   (!querydaml $Query $Result)(go state_Return-result)))
```

Fig. 4 exhibits user input interface and respective query results of examples of Query A and Query B. Query examples show that query agent with capability of information interoperability exactly returns results.

5 Conclusion and Related Works

Essentially, our interoperable information query is an ontology-based searching approach. Compared with some related works of ontology-based searching, the advantage of our work is in that it introduces ontology-mapping function. For example, SHOE (Simple HTML Ontology Extension) was proposed, which allowed HTML developers to annotate useful semantic knowledge into their web pages, and then a web-crawling agent could capture knowledge from these annotated pages [9]. There were authors who developed an annotation strategy and tool to help formulating annotations and searching for specific images based on domain knowledge contained in ontology [10]. Because these ontology-based searching works neglected concept-switching or concept-transformation function, they only solved information interoperability in a limited degree. Also, there were researchers who investigated information query on the DAML-enabled web [11]. This work suggested ontology mapping, but it mainly examined the problem of inference in searching and addressed the issue of describing dynamic procedures and services in DAML on the Web, and the mechanism and details of how to utilize ontology mapping were unclear.

The decentralized feature of Semantic Web makes it inevitable that different communities will use their own ontology to annotate semantic information in their own information sources. In the sense, the inter-ontology mapping plays a crucial role for information interoperability. In addition, Semantic Web promises to support a semantic information-processing platform for multi-agent systems,



```
Subject: X
Intermidiate Varible
Property: Y
Query Varible
Object: Z
Query Varible
[X,?Y,?Z]

[X,?Y,?Z]
[X,'http://www.ichi.sjtu.edu.cn/LandScape/Garden-Ont#Place','Sh
[X,'http://www.ichi.sjtu.edu.cn/LandScape/Garden-Ont#Name','n
[X,'http://www.w3.org/1999/02/22-rdf-syntax-ns#Type','http://www
```

```
Subject: X
Intermidiate Varible
Property: ServiceName
http://www.ichi.sjtu.edu.cn/Agent/Advertisement-Ont
Object: Y
Query Varible
[X,'http://www.ichi.sjtu.edu.cn/Agent/Advertisement-Ont#ServiceN

[X,'http://www.ichi.sjtu.edu.cn/Agent/Advertisement-Ont#Service
[X,'http://www.ichi.sjtu.edu.cn/Agent/Advertisement-Ont#Postcor
[X,'http://www.ichi.sjtu.edu.cn/Agent/Advertisement-Ont#Service
[X,'http://www.daml.org/2001/10/XMLSchema#Type','http://www.k
```

Fig. 4. Two query examples: find information about Yu Yuan Garden; find name of a web pages' search engine

on which agents will be first-class citizens. Considering the two points, many multi-agent systems have been proposed to cope with information interoperability issues, for example, in the areas of ontology heterogeneity, query reformulation, and data integration, etc. Even similar aims have been pursued by some multi-agent systems, such as BUSTER [12], KRAFT [13], Infosleuth [14], and Jeap [15]. Comparing them with our presented Q-based architecture for semantic information interoperability on Semantic Web, our architecture not only considers domain ontology servers and multi-agent system, but also introduces ontology-mapping services that provide concept switching or concept transformation function. Uniquely, we unify the two kinds of interaction: agent-to-agent and human-to-agent interaction in multi-agent query system by Q scenarios, through which end-users may control query agent's behaviors flexibly and interactively. Moreover, the property of Q language is in that it is more oriented to the non-computer professionals than KQML and FIPA ACL, which determines that application designers not only can use to Q scenarios model and describe the agents' behaviors, but also can use them to specify the multi-agent cooperation strategy by combining different Q scenarios in a library.

Next, further works need to be done for improving the practicability of our architecture. Firstly, a standard ontology-mapping representation and inference language have to be set up for make it easy to advertise and collect mapping information of semantic information sources. Certainly, mapping information may be fully manually specified in semantic information sources, or may be semi-automatically determined through some automatic mapping discover techniques, such as case-based reasoning technique.

Acknowledgement. This work was supported by AIEJ (Association of International Education, Japan). Q is being developed by Q consortium including Dept. of Social Informatics at Kyoto University, Dept. of Computer S&E at Shanghai Jiao Tong University, Research Institute of Image Information Science in Japan, and Japanese Science Technology Digital City Research Center. We thank Prof. Toru Ishida of Kyoto University for his supervising, and associate Prof. Zhi-qiang Gao for his detailed discussion.

References

1. J. Hendler. Agents and the semantic web. *IEEE Intelligent Systems*, 16(2): 30–37, 2001.
2. A. Gomez-Perez and O. Corcho. Ontology languages for the semantic web. *IEEE Intelligent Systems*, 16(2): 54–60, 2002.
3. T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 284(5): 34–43, 2001.
4. J. Hendler and D. McGuinness. The darpa agent markup language. *IEEE Intelligent Systems*, 15(6): 72–73, 2000.
5. T. Ishida and M. Fukumoto. Interaction design language Q: the initial proposal. *Transactions of JSAI*, 17(2): 166–169, 2002.
6. K. Kuwabara, T. Ishida, and N. Osato. AgentTalk: coordination protocol description for multi-agent systems. In *Proceedings of the First International Conference on Multi-Agent Systems*, pages 455, 1995.
7. R. Kent Dybvig. The scheme programming language, second edition. Prentice Hall Inc., 1996.
8. M. Gruninger and J. Lee. Ontology applications and design. *Communications of the ACM*, 45(2): 39–41, 2002.
9. S. Luke, L. Spector, D. Rager, and J. Hendler. Ontology-based web agent. In *Proceedings of the First International Conference on Autonomous Agents*, Pages 59–66, 1997.
10. Th. Schreiber, B. Dubbeldam, J. Wielemaker, and B. Wielinga. Ontology-based photo annotation. *IEEE Intelligent Systems*, 16(3): 66–74, 2001.
11. G. Denker, J. R. Hobbs, D. Martin, S. Narayanan, and R. Waldinger. Accessing information and services on the DAML-enabled Web. In *Proceedings of the Second International Workshop on the Semantic Web*, 2001.
12. H. Stuckenschmidt, H. Wache, T. Vögele, and U. Visser. Enabling technologies for interoperability. In Ubbo Visser and Hardy Pundt, editors, *Workshop on the 14th International Symposium of Computer Science for Environmental Protection*, pages 35–46, 2000.

13. A. D. Preece, K. Hui, W. A. Gray, P. Marti, T. J. M. Bench-Capon, D. M. Jones, and Z. Cui. The KRAFT architecture for knowledge fusion and transformation. *Knowledge Based Systems*, 13(2–3): 113–120, 2000.
14. M. Nodine, J. Fowler, T. Ksiezyk, B.Perry, M. Taylor, and A. Unruh. Active information gathering in InfoSleuth. *International Journal of Cooperative Information Systems*, 9(1–2): 3–28, 2000.
15. M. Panti, L. Penserini, and L. Spalazzi. A multi-agent system based on the P2P model to information integration. Computer Science Institute, University of Ancona, 2002.