Towards Enabling Communication among Independent Agents in the Semantic Web

Muhammed Al-Muhammed[⋆] and David W. Embley[⋆]

Computer Science Department, Brigham Young University, Provo, UT 84602, USA
{mja47,embley}@cs.byu.edu

Abstract. Independent agents can roam the semantic web searching for information and services. How can we enable them to communicate with one another? Ontologies also exist on the semantic web. We show in this paper that if we augment ontologies appropriately, we can attach agents to them and enable the agents attached to them to interact on the fly without requiring prior agreement on vocabulary and service protocols. This solution to agent communication, however, relies on the system being able to recognize local agent concepts, local agent services, and local agent service requests, all with respect to the ontology. Experiments we have conducted on the agent communication system we built show that we can successfully recognize concepts, services, and service requests and that we can successfully map them to a global ontology and thus enable communication among independently developed software agents.

Keywords. Agent communication, semantic interoperability, ontologies.

1 Introduction

Researchers frequently make three assumptions to enable agent communication: agents must (1) share ontologies, (2) speak the same language, and (3) pre-agree on a message format [1]. Although agents can communicate by satisfying these three requirements, these requirements prevent agents from interoperating on the fly. Interoperating on the fly requires agents to make the needed mapping between them without agreeing on language and message format, or sharing any type of ontological knowledge. Uschold [2] reflects on this idea by saying that "the holy grail of semantic integration in architectures" is to "allow two agents to generate needed mappings between them on the fly without a-priori agreement and without them having built-in knowledge of any common ontology."

To allow agents to interoperate on the fly, we have developed a MatchMaking System (MMS) that enables communication. In our solution we assume neither shared ontologies, nor a common language, nor a shared message format. Our approach has two key ideas.

⋆ This work is supported by the National Science Foundation under grant #IIS-0083127.

1. *Independent Global and Local Ontologies.* Rather than requiring agents to share ontologies, we provide our system with an agent-independent, domain-specific ontology, called the *global ontology.* When an agent wishes to communicate within a particular application domain, our system applies an information extraction engine to the agent's code to extract useful information. This useful information, which we call a *local ontology*, includes the names of concepts the agent uses such as class names, parameter names, and variable names, and the types of these concepts. To compensate for not having a shared ontology, our system maps each agent's local ontology to the application domain's independent global ontology.
2. *Automatic Message-Service Mapping.* Rather than having agents deal directly with in-coming messages, our system automatically maps an in-coming message to an appropriate service. As an immediate consequence, agents do not have to use the same communication language and pre-agree on a message format. As a necessary step to achieve the automatic mapping, our system parses an agent's code, finds its services, and expresses them in an agent-independent way. Once our system has an agent's services, it does a mapping between these services and an in-coming message. Then, using the local/global ontology mappings, our system can appropriately convert parameters of a requesting agent's message to parameters of a providing agent's service and receive results and convert them to results the requesting agent can "understand."

We know of no comparable systems. We do know, however, that the requirement for agent communication is well established in the literature. Bradshaw [3] defines a software agent as "a software entity, which functions autonomously and continuously in a particular environment, often inhabited by other entities;" Bradshaw continues "we expect an agent that inhabits the same environment with other agents to be able to communicate and cooperate with them." Genesereth [4] emphasizes agent communication by suggesting that any entity that cannot communicate is not an agent. To enable agent communication, researchers have proposed agent communication languages such as KQML [5] and FIPA [6]. Agents can communicate using these languages, provided that they agree on these languages, on message formats, and on a shared ontology prior to the communication. But these requirements prevent agents from interoperating on the fly. An attempt to solve the problem of pre-agreement on a message format has been recently proposed in [1]. Unfortunately in this solution agents must share ontologies and be able to parse templates, which to some extent is tantamount to the agents agreeing on a message format, and thus this solution fails to allow agents to interoperate on the fly. Attempts to match services with requests have been proposed in [7, 8]. Both solutions define languages that agents must use to describe their capabilities and represent their requests, and both give matching algorithms that can match capabilities and requests represented in the language. These solutions, however, require agents not only to share ontologies, use a common language, and pre-agree on a message format, but also to use the defined languages, and thus they fail to allow agents to interoperate on the fly.

The main contribution of our paper is that it shows a possible way to enable agent communication within a specified application domain without requiring prior agreement regarding ontological concepts and message handling protocols. We present the details of our contribution as follows. Section 2 describes the initialization, which consists of establishing a mapping between local and global ontologies and analyzing services to discover their properties and capabilities and express them in an agent-independent way. Section 3 describes the operation of our system. Section 4 describes the experiments we used to test the functionality of our system. In Sect. 5, we conclude and give directions for future work.

2 System Initialization

In order for the MatchMaking System (MMS) to enable agents to communicate, it must be initialized. The initialization includes two steps. In the first step, the MMS establishes mappings between its global ontology and an agent's local ontology. In the second step, the MMS analyzes an agent's services and expresses them in an agent-independent way.

2.1 Local-Global Ontology Mappings

The MMS parses an agent's code to extract useful information[1]. The MMS then utilizes this information to establish mappings between its global ontology and an agent's concepts, data formats, and units of measurement.

The system's global ontology is domain specific and contains the following information.

1. *Concept Names.* These names denote objects we expect to find in the domain. These names can be single words or short phrases.
2. *Concept Recognizers.* Dictionaries provide synonyms for global concept names. These synonyms can be obtained from any available resources such as web sites that match the ontology's domain. Using these dictionaries, we create regular-expression recognizers for the global concepts. The MMS uses the recognizers to find local concepts in an agent's code that may map to the global concepts. For example, the regular expression $(CPU|Processor)(Speed)|(Processor)(clock)(Speed)$ can be a recognizer for the global concept *ProcessorSpeed*[2].
3. *Value Recognizers.* These recognizers identify constant values for a concept. For example, the value recognizer $[1-9][0-9]^{*}[.]?[0-9]^{+}\backslash\backslash s^{*}(GHz|MHz)$ shows that a value of *ProcessorSpeed* is a number followed by either "GHz" or "MHz".

[1] For several years we have been working on data extraction (e.g. see [9]). Our work here uses data extraction to establish mappings from an agent's code to the global domain ontology, which is the key to enabling on-the-fly agent communication

[2] Because concept names become identifiers in an agent's code, we omit spaces for concept names throughout our discussion.

```
<daml:Class rdf:ID=”Computer”>
</daml:Class>
<daml:Class rdf:ID=”Processor”>
</daml:Class>
<daml:Class rdf:ID=”ProcessorManufacturer”>
</daml:Class>
<daml:Class rdf:ID=”ProcessorClass”>
</daml:Class>
...
<daml:ObjectProperty rdf:ID="hasProcessor">
  <rdfs:domain rdf:resource="#Computer"/>
  <rdfs:range rdf:resource="#Processor"/>
</daml:ObjectProperty>
<daml:Class rdf:about="Computer">
  <rdfs:subClassOf>
     <daml:Restriction daml:CardinalityQ="1">
        <daml:onProperty rdf:resource="#hasProcessor"/>
     </daml:Restriction>
  </rdfs:subClassOf>
</daml:Class>
```

```
...
<daml:ObjectProperty rdf:ID="isPartOfProcessor">
  <rdfs: domain rdf:resource="#ProcessorType"/>
  <rdfs: range rdf:resource="#Processor"/>
</daml:ObjectProperty>
<daml:ObjectProperty rdf:ID="isPartOfProcessorType">
  <rdfs: domain rdf:resource="#ProcessorClass"/>
  <rdfs: range rdf:resource="#ProcessorType"/>
</daml:ObjectProperty>
<daml:ObjectProperty rdf:ID="isPartOfProcessorType">
  <rdfs: domain rdf:resource="#ProcessorManufacturer"/>
  <rdfs: range rdf:resource="#ProcessorType"/>
</daml:ObjectProperty>
<daml:DatatypeProperty rdf:ID="ProcessorSpeed">
 <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
 <rdfs:range rdf:resource="http://www.w3.org/2000/10/XMLSchema#Literal"/>
</daml:DatatypeProperty>
...
```

Fig. 1. A DAML representation for a computer-shopping ontology.

4. *Data Format Recognizers.* Data can appear in different formats, which the MMS must be able to recognize. The regular expression $([0]?[1-9]|[1][1\text{-}-2])\backslash\backslash s^*[/]\backslash\backslash s^*([0]?[0-9]|[1][0-9]|[2][0-9]|[3][0-1])\backslash\backslash s^*[/]\backslash\backslash s^*[\backslash\backslash d]\{4\}$, for example, can recognize one of the date formats: Month/Day/Year.
5. *Unit of Measurement Recognizers.* For global concepts that have units, we must develop recognizers that can identify their units. For example, the regular expression $(US(D|\$)|(\$)|(EUR))$ can identify US dollars or Euros.
6. *Relationships and Participation Constraints.* Relationships determine how global concepts relate to each other. Participation constraints determine the number of times instances of a concept relate to instances of another concept. Figure 1 provides a DAML representation of some of the relationships between global concepts in the computer-shopping domain. For example, each Computer has one *Processor* (represented by the DAML restriction: *CardinaliyQ*=“1”) on the properties of *hasProcessor*, and a *ProcessorType* is part of a *Processor* (represented by the relation: *isPartOfProcessor*).

The strings in an agent’s code recognized by the recognizers in the global ontology constitute the agent’s local ontology. Conceptually, a local ontology of an agent is a subset of the global ontology. A local ontology includes the names of concepts the agent uses, which are found in the agent’s code, and the definition of data formats and units of measurement, which are usually found in literal strings and comments.

Although not necessarily designed for our system, the quality of an agent’s code is highly important for correct communication between our MMS and an agent. The code should have meaningful names of concepts rather than unintelligible names such as X or Y. In addition, the code should have definitions of data representations and units of measurement. These definitions can appear in the code in usual ways. Figure 2 provides a partial example. As can be seen, the *Price* is of type integer and is expressed in “US$” as mentioned in a comment.

```
int Price = 0; //US$
double ProcessorClockSpeed; //GHz
string ProcessorClass;
double RamStandardSize; //MB
string Date ="10/16/2003";
```

Fig. 2. Sample Java code.

```
//input: B is Book
public double getPrice(string B) {···}
```

Fig. 3. An agent's service.

The unit of measure for *ProcessorClockSpeed* is "GHz". The *ProcessorClass* and *RamStandardSize* declarations mention two more concepts and another unit of measurement in a comment. The initial value for *Date* represents the date in an acceptable format to the agent.

2.2 Service Analysis

Agents, in our system, do not directly handle messages because they do not share a communication language or a message format. The MMS, therefore, calls services that can answer messages rather than passing these messages directly to agents. This requires the MMS to "know" an agent's services. Specifically it should know what input each service takes and what output it returns. Thus, the MMS must be able to find a service declaration and discover from this declaration and associated information the input and output of the service.

Service declarations, which for our MMS implementation are methods implemented in a programming language, exhibit a specific pattern that can be captured in a regular expression. To find the services of an agent, we created a regular expression that can identify Java, C, or C^{++} service declarations.

Once we find a service declaration, we need to recognize the input parameters and the output information. The MMS determines the input parameters from the method declarations themselves or from the declarations and documenting comments. The MMS determines the output from the return type, documenting comments, the name of the service, or the results of executing the services. We consider three cases.

1. The MMS recognizes all the input parameters and output parameters of a service during the local-global ontology mapping process. In this case the MMS simply takes this information and builds a service signature, which consists of the name of the service, the return type, the input parameters, and the output information.
2. The MMS fails to recognize at least one of the input/output parameters. This can happen when an agent's programmers use names such as X or Y rather than recognizable names as input parameters of a service or when the return type of a service does not tell enough about the returned values. In this case, it may be possible to find some associated information (documenting comments or name of the service) to help recognize the parameter(s). The service in Fig. 3 shows that the name of the service can provide enough information to discover the output parameter. The return type of the service is *double*, which by itself is not enough to determine the output of the service, but the name of the service, *getPrice*, which includes the concept *Price* is a good signal that this service returns the *Price* for something. It is not

likely that the MMS can discover the input parameter of the the service, *B*, because it is not by itself recognizable, but from the comments the MMS can recognize *B* as a name for *Book*, which is recognizable.

3. Executing a service is another way to discover its output parameters. When a service executes successfully, it returns information. The MMS can apply value recognizers to this information in an attempt to determine what information the service returns. In order to execute a service, the MMS provides the input parameters with appropriate domain values. Therefore for this strategy to work, the global ontology must supply typical values for ontology concepts.

If the MMS cannot fully resolve the input/output parameters for a service, it may request help from the agent's developers. If the developers do not supply the needed information, the MMS ignores the service.

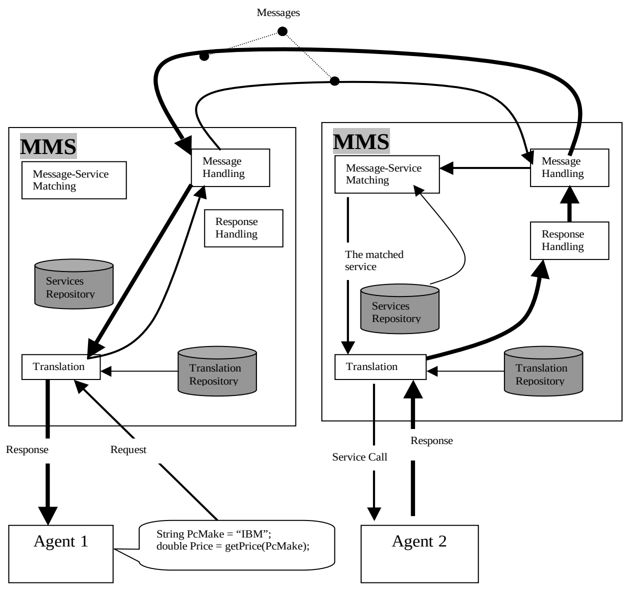
After analyzing all the services, the MMS outputs an agent-independent representation of each service. For each service, this representation consists of the name of the service, its input parameters along with their types, and its output parameters along with their types, all in terms of global ontology concepts.

3 System Operation

Once the MMS has been initialized, it becomes ready to operate and consequently enables its agent to communicate with other agents. Figure 4 shows the interactions between the MMS and agents. Each agent has its own copy of the MMS, and all these copies are identical except for the repositories, which contain data specific for each agent created during initialization.

The MMS handles communication between agents as follows. Agent 1 in Fig. 4 sends a request for some information about, say a PC, to the MMS. The MMS receives this request and then using information in the Translation Repository obtained during initialization translates the vocabulary of the request to the global vocabulary and normalizes the units (e.g. changes currencies to US$) and the data formats (e.g. changes dates to Month/Day/Year) and passes the request to the Message Handling component, which creates a KQML [5] message and routes the request to Agent 2. When the MMS of Agent 2 receives the message, in its Message-Handling component, it switches the message to the Message-Service Matching component. The Message-Service Matching component requests services from the Service Repository obtained during initialization and matches the message against these services. If a match is found, then the Message-Service Matching component passes the information to the Translation component, which translates the information using Agent 2's vocabulary, units, and data format. The MMS then calls the service.

Agent 2 executes the service and returns a response. The MMS receives the response, which is represented in Agent 2's local vocabulary. The Translation component translates the response to the global vocabulary and normalizes the units and data formats and passes the response to the Response-Handling component. The Response-Handling component filters out unwanted information by



```
String PcMake = "IBM";
double Price = getPrice(PcMake);
```

Fig. 4. The interaction among the MMS components.

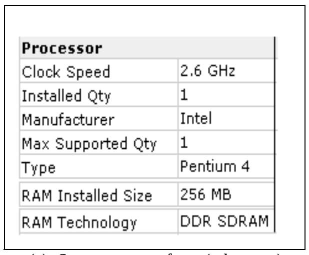
comparing the response to the requested information and passes the response to the Message-Handling component, which makes a KQML message and sends it to Agent 1. The MMS of Agent 1 receives the message in its Message-Handling component, which switches the response to the Translation component. The Translation component translates the answer to Agent 1's local vocabulary and local units and formats. Finally, the MMS returns the response to Agent 1.

Agents formulate their requests as ordinary function calls to each other although they do not know each other's services. The lines of code in Fig. 4 show an example. The agent simply initialized a parameter variable ($PcMake$="IBM") and calls a method ($double\ Price = getPrice(PcMake)$). Although oversimplified for the application (we would need more parameters to identify an actual PC), the example is sufficient to show how we couple our MMS with an agent[3]. To make all this work, the MMS needs to make some changes to the code for agent service requests. Figure 5 shows the translated code. The MMS transforms this request to the series of calls in Fig. 5. The first MMS-generated call allows the MMS to receive the input parameter, *PcMake*, along with its value "*IBM*". The second call passes the function name, *getPrice*, to the MMS. The return type of the second call is *double*. If the return type were supposed to be *int*, we would have called *MMS.sendint*("*getPrice*").

[3] For our actual prototype MMS, we initialized variables through a simple form interface so that we could enter the values by hand while the agent system was running

```
MMS.sendString("PcMake="+"IBM");
double Price = MMS.senddouble("getPrice");
```

Fig. 5. The MMS-generated calls.



Processor	
Clock Speed	2.6 GHz
Installed Qty	1
Manufacturer	Intel
Max Supported Qty	1
Type	Pentium 4
RAM Installed Size	256 MB
RAM Technology	DDR SDRAM

(a): Some concepts from (*cdw.com.)*

Global concept name	ProcessorManufacturer
Synonyms found in websites	ProcessorManufacturer, ProcessorMaker
Concept name recognizer	(Processor)(Manufacturer|Maker)
Memory capacity units found in websites	MB, GB
Memory capacity units recognizer	(MB|GB)

(b): A concept recognizer and a unit recognizer.

Fig. 6. Some concepts from a computer-shopping web site, a concept recognizer, and a unit of measurement recognizer.

4 Experimental Results

To measure the performance of the MMS, we implemented two multi-agent systems, namely Computer-Shopping Agents and Meeting-Scheduling Agents. To avoid a natural bias toward our work, we implemented the services of each agent as described by others, and we took the concept names, units of measurement, data representations, and data from sources defined by others.

4.1 Computer-Shopping Agents

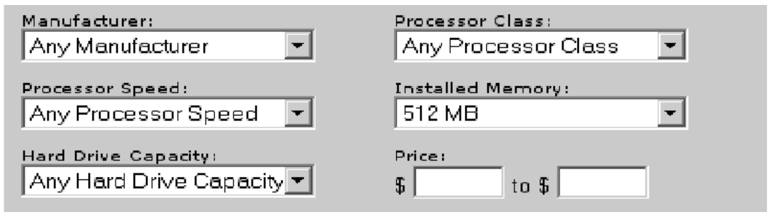
For any application we first create the global ontology. To create the global ontology for our computer-shopping application, we visited 8 web sites (*amazon.com*, *cdw.com*, *dell.com*, *half.ebay.com*, *gateway.com*, *plasmakings.com*, *price.com*, and *ubid.com*) and collected the concepts for each part/attribute of a computer, the units of measurement, and the data formats. Figure 6(a) shows an example of a web site, which contains concepts (e.g. *ProcessorManufacturer*) and units (e.g. *GHz*).

Given the concepts and the implied relationships among them, we created a global ontology for the application covering the concepts in which we were interested. Figure 1 shows part of the global ontology of our computer-shopping application. To form names for the concepts, we designated one of the concept names (e.g. *ProcessorManufacturer* in Fig. 6(a)) as the global concept name. We let that name and all other names be synonyms for the concept. We created a recognizer for the concept by placing its synonyms together in a regular expression. Figure 6(b) shows a global concept (*ProcessorManufacturer*), its synonyms, and the global concept name recognizer. In addition, we created recognizers for each unit of measurement that we found in the 8 sites. Figure 6(b) shows an example of the recognizer for the units used in the 8 sites to measure *MemoryCapacity*. For this application there were no data formats of interest.

After creating the global ontology for the computer-shopping application, we created seller and buyer agents. The seller agents need to provide services for buyer agents. We generated services for a seller agent according to a form we found in a particular web site, Fig. 7 shows a an example. We did not generate all possible services implied by a form because the number of the services would have been large[4] and, more importantly, because many of the services would have been redundant in the sense that they could not measure anything different from a few well chosen services. For a form with n field names, we generated n services, where the i^{th} service has i input parameters chosen from the field names. We made sure that each field name participated in the generated services at least once.

We generated a service signature by determining its name, input parameters, and return type as follows. We obtained input parameter names for the service directly from the field names of the form. We determined the type of each input parameter of the service according to the type of the allowed values in its matching field. If the value was a number without a decimal point, we chose the type to be *int*; if the value was a number with a decimal point, we chose the type to be *double*; otherwise we chose the type to be *String*. We defined the return type of the service according to the result we obtained by entering valid values for the field names we chose to be input parameters. If the site returned one value, we let the type of this value be the return type of the service (determined in the same way we determined input value types). Otherwise we defined the return type as a Java class with the site's concept names as the class's attributes, each with a type determined in the same way we determined input value types. Finally, we used the following convention to name the service. If the service returned one value, we used the concept name for the value prefixed with *get* as the service's name (e.g. *getPrice*); otherwise we used the generic name *getPcInfo*. For example, Fig. 7(a) shows a form provided by site *shopping.yahoo.com*, and Fig. 7(b) shows a service, *PcInfo getPcInfo*(*String InstalledMemory*), generated from the form. We chose the field name *InstalledMemory* to be the service input parameter. The type of the input parameter is *String* because the form allows the values for this field to be alphanumeric values (see "512 MB" in the form). The return type of the service is the class *PcInfo* with attributes *ProcessorManufacturer*, *ProcessorClass*, ... because when we filled in the field *InstalledMemory* with the value "512 MB", we obtained the information in Fig. 7(c), which has multiple values. We defined the units for each attribute through comments (e.g. *ProcessorSpeed* is in *GHz* and the *Price* is in US dollars). Finally, since the returned information has multiple values, we named the service using the generic name *getPcInfo*. The generated services for seller agents need data to answer a buyer agent's requests. We obtained the data for these services from the same sites we used to generate services.

[4] Any combination of fields can constitute a service. For example, in a form with 6 fields, the number of possible services (excluding services with 0 input parameters) is C(6,1) + C(6,2) +··+ C(6,6) = 63.



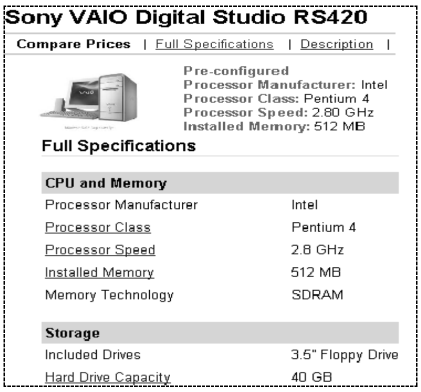
Manufacturer: Any Manufacturer	Processor Class: Any Processor Class
Processor Speed: Any Processor Speed	Installed Memory: 512 MB
Hard Drive Capacity: Any Hard Drive Capacity	Price: $ to $

(a): A form (*shopping.yahoo.com*).

```
PcInfo getPcInfo (String InstalledMemory)
{
   …
}

//user-type definition
class PcInfo
{
  String ProcessorManufacturer;
  String ProcessorClass;
  String ProcessorSpeed; //GHz
  …;
  double Price; //$
}
```

(b): A service signature generated from the form and a type definition.



Sony VAIO Digital Studio RS420

Compare Prices | Full Specifications | Description |

Pre-configured
Processor Manufacturer: Intel
Processor Class: Pentium 4
Processor Speed: 2.80 GHz
Installed Memory: 512 MB

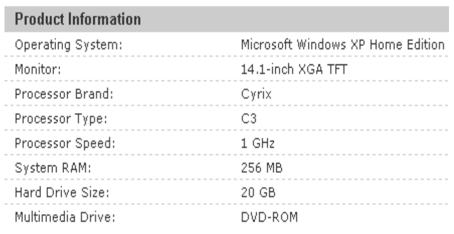
Full Specifications

CPU and Memory	
Processor Manufacturer	Intel
Processor Class	Pentium 4
Processor Speed	2.8 GHz
Installed Memory	512 MB
Memory Technology	SDRAM
Storage	
Included Drives	3.5" Floppy Drive
Hard Drive Capacity	40 GB

(c): Returned results for Installed Memory = “512 MB”.

Fig. 7. Service generation.

The buyer agent needs request messages to obtain information from a seller agent. A request is a call to a service. For each request, we must determine what information it provides for the called service’s input parameters and what information it requests. We use concept names to hold the information that a request provides for the called service’s input parameters and to specify a parameter to which the result of the call is assigned. The input parameters of a request receive their data through a user interface form, which we created using field names we took from the web site we chose for the buyer agent. We generated requests for the buyer agent using concepts and units of measurement that came from the web site. The number of requests was sufficient to cover all the concepts and units of measurement in the web site and to invoke all services of every seller agent. The requests return values, which should be represented in units and formats acceptable for the buyer agent. We took the units of measurement and the data formats for the concepts from the chosen buyer web site. Figure 8(a) shows the web site from which we created the request message in Fig. 8(b). We created the request as function call to an assumed service, *getPrice*, using three concept names, *ProcessorSpeed* and *SystemRAM*, which receive their values at runtime from a user interface, and *Price*, which holds the result of the call. The request also specifies the units of measurement and data formats the buyer agent accepts (i.e. the buyer agent accepts *Price* in US$).



Product Information	
Operating System:	Microsoft Windows XP Home Edition
Monitor:	14.1-inch XGA TFT
Processor Brand:	Cyrix
Processor Type:	C3
Processor Speed:	1 GHz
System RAM:	256 MB
Hard Drive Size:	20 GB
Multimedia Drive:	DVD-ROM

(a): Concepts from website(*walmart.com.*)

```
string ProcessorSpeed; //GHz
string SystemRAM; //MB
double Price; //$
//get values from a user interface
ProcessorSpeed=ProcSpdFiled.getText();
SystemRAM = MemFiled.getText();
//call to assumed service
Price = getPrice(ProcessorSpeed, SystemRAM);
```

(b): An example of a request.

Fig. 8. Site: walmart.com.

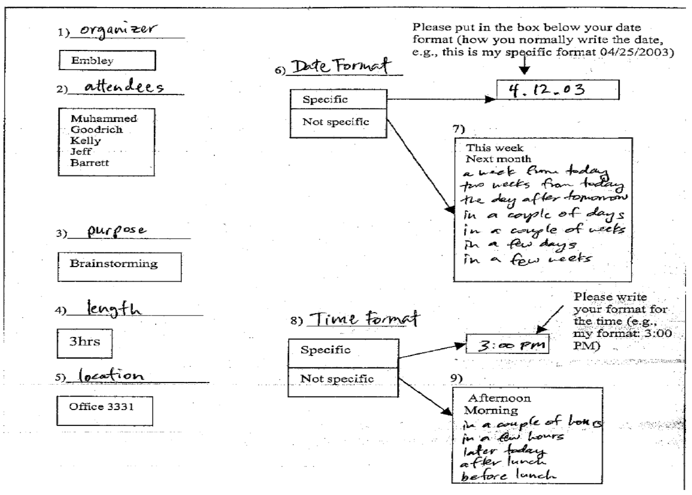
To measure the performance of the MMS for the computer-shopping application, we fixed the global ontology and used 9 test sites (different from the 8 sites we used to build the global ontology), one for a buyer agent and the rest for seller agents. We measured the MMS's performance in mapping concepts, units of measurement, and data formats used by agents to the global ontology. The agents' code included 104 concepts, which the MMS needed to map to global concepts. The MMS generated 94 mapping pairs of the form (*Local*, *Global*), of which 91 were correct, yielding (91/104) or 88% recall and (91/94) or 97% precision. The units of measurement, in the agents' code, that the MMS needed to recognize were currencies, processor/hard-drive speed units, and memory/hard-disk capacity units. The currency types in the 9 test sites were US$, GBP (Great Britain Pound), and EUR (Euro). There were 9 currency instances that the MMS needed to recognize. The MMS recognized 9 all of which were correct. The number of processor/hard-drive speed units and memory/hard-disk capacity units was 23. The MMS recognized 25, of which 23 were correctly associated with their global counterparts. Altogether there were 32 unit instances; the MMS recognized 34, of which 32 were correct, yielding 100% recall and 94% precision.

Before we discuss our results, we give our rationale for excluding all mappings, except local-global ontology mappings, from our performance measurement. The local-global ontology mapping generates mapping pairs between local concepts and global concepts, recognizes units of measurement, types, and data formats. This information is necessary and sufficient to translate messages and services from local to global and vice versa. Representing messages and services in terms of the global ontology makes message-service matching straightforward because the concepts to be matched belong to the same ontology. Furthermore, it makes result filtering straightforward because the response handler only needs to make simple comparisons between information represented in terms of the same ontology. Type conversions are also straightforward because the MMS knows both the source and the target types, and thus the conversion is straightforward (we do not consider loss of precision). Finally, since the MMS will have recognized all the concept names during the local-global ontology mapping, the input/output parameters of the services, which we need to build service signatures, are already recognized. These observations allow us to focus only on the local-global ontology mapping and ignore measuring the other processes.

We now discuss the results of the local-global ontology mapping. The MMS failed to recognize and consequently map 13 concepts out of 104 in the agents' code. *ProcessorBrand* and *ProcessorFrequency* are examples of an agent's concepts that did not map to global concepts. They are synonyms for the global concepts *ProcessorManufacturer* and *ProcessorSpeed* respectively, but the recognizer for *ProcessorManufacturer* does not include *ProcessorBrand* (see Fig. 6(a)) and the recognizer for *ProcessorSpeed* does not include *ProcessorFrequency.* Although we can simply fix this problem for these sites by adding the synonyms that appear in these sites to the global concept recognizers, we are aware that this solution may not resolve the problem because there may be additional synonyms, which are still not included. A possible technique, which we will further investigate in future work, to fix this recall problem is to generalize the recognizers by not limiting ourselves to the concepts that we had seen in web sites or provided by subjects, but also to exploit our knowledge of the domains to augment the recognizers with the concepts that we would expect to see in these domains. Further we can use auxiliary synonyms dictionary, such as WordNet [10], and we can use more sophisticated matching techniques [11, 12].

The MMS generated 3 incorrect mapping pairs all of them of the form (*ProcessorType*, *ProcessorClass*). The incorrect pairs arise because of a naming ambiguity among the web sites we visited. Most of the web sites either represented the processor as one concept *Processor* or in terms of three different concepts, namely *ProcessorManufacturer*, *ProcessorType*, and *ProcessorSpeed*, using these names or synonyms for these names. A few, however, represented a processor in terms of only two different concepts, namely *ProcessorType* and *ProcessorSpeed*, using these names or synonyms for these names. The web sites that represented the processor in terms of three concepts used the names *ProcessorType* or *ProcessorClass* to represent the type of the processor (e.g. "Pentium 4"), and we selected *ProcessorClass* to be the global concept for our global ontology. The web sites that represented the processor in terms of two concepts also used the names *ProcessorType* or *ProcessorClass* to represent the manufacturer and the type of the processor (e.g. "Intel Pentium III"), and we selected *ProcessorType* to be the global concept for our global ontology. As a result, when we used our predetermined method for generating recognizers, both global concepts *ProcessorType* and *ProcessorClass* were identical—both recognizers were $(Processor)(Type|Class)$. Thus, the MMS incorrectly declared the pairs (*ProcessorType*, *ProcessorClass*) because both recognizers recognized *ProcessorType* in an agent's code and produced two pairs: (*ProcessorType*, *ProcessorType*), which is correct, and (*ProcessorType*, *ProcessorClass*), which is incorrect.

Possible techniques, which we plan to investigate further in future work, to fix this precision problem are to use more sophisticated matching techniques [11, 12] and to use reasoning rules. For example, consider the reasoning rule: "If *ProcessorType* (or *ProcessorClass*) is recognized in an agent's code and there is no occurrence of *ProcessorManufacturer*, then *ProcessorType* (or *ProcessorClass*) maps to the global concept *ProcessorType*; otherwise it maps to the global concept *Processor Class*." This rule would detect and remove the incorrect pairs.



1) organizer

Embley

2) attendees

Muhammed
Goodrich
Kelly
Jeff
Barrett

3) purpose

Brainstorming

4) length

3hrs

5) location

Office 3331

6) Date Format

Specific

Not specific

Please put in the box below your date format (how you normally write the date, e.g., this is my specific format 04/25/2003)

4.12.03

7)

This week
Next month
a week from today
two weeks from today
the day after tomorrow
in a couple of days
in a couple of weeks
in a few days
in a few weeks

8) Time format

Specific

Not specific

Please write your format for the time (e.g., my format: 3:00 PM)

3:00 PM

9)

Afternoon
Morning
in a couple of hours
in a few hours
later today
after lunch
before lunch

(a): An example of a filled-in worksheet.

(1) The user will put the name of the person who is calling for the meeting (e.g. Embley is calling for the meeting). Choose a label (preferably a single word but possibly a short phrase) to indicate that.
(2) The user will choose from the list the names of the people to attend the meeting. Choose a label (preferably a single word but possibly a short phrase) to indicate that.
(3) ...

(b): Examples of descriptions of the semantics of the blanks in the worksheet.

Fig. 9. An example of a filled-in worksheet and semantic descriptions of its blanks.

Regarding the units, the MMS also incorrectly recognized 2 units. It recognized *MB* (Megabyte) twice as a hard-disk capacity, but the agents used *GB* as their hard-disk capacity unit. If any web site we used to create the global ontology had had *MB* units for hard-disk capacity, we would have included *MB* as a unit and would not have encountered this error.

4.2 Meeting Scheduling Agents

For our meeting scheduling application, we requested some of the people in our research group to create their own scheduling information for agents in their own terms. The scheduling information consisted of concept names used to schedule meetings, data formats, and services. Figure 9(a) shows the worksheet we used to request concepts and data formats. As can be seen, the worksheet has numbered boxes. The numbers refer to a high-level description of the semantics of each box; Fig. 9(b) shows examples of high-level descriptions for Boxes 1 and 2. Subjects filled in each numbered blank with the concept name (a single word or a phrase) to reflect the semantics of the description associated with that number. Subjects filled in Boxes 7 and 9 with non-specific dates and non-specific times respectively. In addition, there were two more boxes, one for a sample date and one for a sample time.

These subjects from our research group also translated some high-level descriptions for services into service signatures. In the translation we asked them to use the concept names that they had chosen when they filled in their work sheets. They chose their own names of services and input/output parameter types.

We received 12 completely filled-in work sheets from 12 different individuals, 8 of which turned in work sheets several weeks before the remaining 4. Using methods similar to those used in our computer-shopping application, we used the concepts and data formats in the first 8 work sheets to build the global ontology. We used the concept names, data formats, and service signatures in the other 4 work sheets to build four agents. We implemented the agents based on ideas presented in [13].

The MMS needed to map 28 concepts in the agents' code to global concepts. The MMS generated 22 mapping pairs of the form (*Local*, *Global*), of which 22 were correct, yielding (22/28) or 79% recall and (22/22) or 100% precision. The four agents used four different formats for *Date*, instances of which are "25 Apr 04", "4/25/04", "4-25-2004", and "4.25.04". The MMS recognized all four of these formats and no others, yielding 100% recall and precision. The four agents used only one time format, namely a format with 12-hours with AM or PM. The MMS recognized this format, yielding 100% recall and precision.

The MMS missed 6 concepts, namely *Initiator*, *Authority*, *ScheduledBy*, *PersonsInvited*, *DateFormat*, and *TimeFormat*. The first three concepts are synonyms for the concept *Inviter*; the fourth concept is a synonym for the concept *Invitee*; the fifth concept is meant to be a synonym for the concept *Date*; and the sixth concept is meant to be a synonym for the concept *Time*. Because the concept recognizers for *Inviter*, *Invitee*, *Date*, and *Time* do not have these synonyms, the MMS missed these 6 concepts. Although an obvious fix for the problem of missing the first 4 concepts is to add more synonyms to the global concept recognizers (e.g. adding *Initiator* to the *Inviter* recognizer), this fix raises the same concerns we discussed earlier. We believe that missing the 5^{th} (*DateFormat*) and 6^{th} (*TimeFormat*) concepts is inevitable because these concepts are not synonyms for *Date* and *Time*. Most likely they originated because of a subject's misunderstanding of the semantics of Boxes 6 and 8 in Fig. 9.

5 Conclusions and Future Work

We have developed a framework to allow agents to communicate with no need to share ontologies, use a common language, and pre-agree on a message format. In our framework, our MMS has a predefined global domain ontology, which is a standard web ontology augmented with regular expressions recognizers. The MMS maps local agent concepts to global domain concepts and uses these mappings to translate requests and services to terms specified in the global domain ontology. With requests and services all expressed in common ontological terms communication is straightforward.

We realize that using concepts derived from web sites and meaningful names or phrases requested from subjects reveals our assumption that the concept names must be human readable in order for the MMS to work properly. However, we believe that this assumption is quite reasonable especially for agents, which typically use ontological concepts, with human readable names, to define their knowledge and to communicate with other agents. Tests we conducted on two domains (Computer-Shopping and Meeting Scheduling) showed that for these two applications our system performs reasonably well with an average of 84% recall and an average of 99% precision for concept recognition, 100% recall and an average of 97% precision for units recognition, and 100% recall and precision for data format recognition.

Although we have largely achieved our initial goal of enabling agents within a pre-specified domain to communicate on the fly without the usual pre-agreement requirements, much remain to be done. For example, the local-global mapping process could be strengthened as proposed in Sect. 4.1. Further, the ideas presented in this paper should be integrated into the semantic web, allowing agents to roam the web and attach themselves dynamically to domains of interest.

References

1. Payne, T.R., Paolucci, M., Singh, R., Sycara, K.: Facilitating Message Exchange through a Middle Agent. In: Proceedings of the 1st International Joint Conference on Autonomous Agents and Multi-Agent Systems. (2002) 561–562
2. Frank, M., Noy, N.F., Staab, W.: The Semantic Web Workshop at the 11th International WWW Conference (WWW-2002). SIGMOD Record **31** (2002)
3. Bradshaw, J.M.: Software Agents. AAAI press, Menlo Park, California (1997)
4. Genesereth, M., Ketchpel, S.: Software Agents. Communications of the ACM **37** (1994) 48–53
5. Labrou, Y.: Semantics for an Agent Communication Language KQML. PhD thesis, "University of Maryland" (1997)
6. FIPA: Fipa Agent Communication Language Specification. Technical report, Foundation for Intilligent Physical Agents (2002) URL, http://www.fipa.org/repository/aclspecs.html.
7. Lu, H.: Ontology based Agent Services Description and Matchmaking on the World Wide Web. In: Preceedings of the 9th Australian World Wide Web Conference, Queensland, Australia (2003) 110–117
8. Sycara, K., Wido, S., Klusch, M., Lu, J.: LARKS: Dynamic Matchmaking Among Heterogeneous Software Agent in Cyberspace. Autonomous Agents and Multi-Agent Systems **5** (2002) 173–203
9. Embley, D., Campbell, D., Jiang, Y., Liddle, S., Lonsdale, D., Ng, Y.K., Smith, R.: Conceptual-Model-Based Data Extraction from Multiple-Record web pages. Data & Knowledge Engineering **31** (1999) 227–251
10. Miller., G.A.: Wordnet:A Lexical Database for English. Communications of the ACM **38** (1995) 39–41
11. Bernstein, P., Rahm, E.: A Survey of Approaches to Automatic Schema Matching. The VLDB Journal **10** (2001) 334–350
12. Xu, L., Embley, D.: Using Domain Ontologies to Discover Direct and Indirect Matches for Schema Elements. In: Proceedings of the Workshop on Semantic Integration (WSI'03), Sanibel Island, Florida (2003) 105–110
13. Jennings, N.R., Jackson, A.J.: Agent-Based Meeting Scheduling: A Design and Implementation. IEEE Electronics Letters Journal **31** (1995) 350–352