Applying Semantic Web Technologies to Matchmaking and Web Service Descriptions

Amer Al Shaban and Volker Haarslev
Department of Computer Science and Software Engineering
Concordia University, Montreal, Canada

***Abstract*-The recent growth of using agents in representing web services is causing difficulties in finding specific types of services. This problem usually arises because matchmaking techniques for services are often based on string comparison and service providers might neglect to provide enough or appropriate keywords for the matchmaking process. In this paper we report on an approach that makes use of formal ontologies and automated reasoning services in order to improve the matchmaking process. Our approach is based on the Ontology Web Language (OWL), the OWL reasoner RACER, and the agent framework DECAF. Our use of OWL ontologies is two-fold. First, we use ontologies in order to express the particular knowledge of agents. These ontologies are grounded by referring to a so-called common upper ontology providing the necessary glue between the different agent domains. Second, with the help of OWL-S, a standard OWL ontology designed for specifying service descriptions, agents describe formally their offered web services. Our approach depends on a middle-ware agent called matchmaker which will be in charge of matching required services to proper provider agents. Due to the use of OWL ontologies the matchmaking process can be reduced to query processing and ontology reasoning implemented by the RACER system. Our approach has been demonstrated using an e-business scenario, where several buying and selling agents for various products are involved. The communication protocol is based on OWL-S and allows buying agents to adapt smoothly to dynamically changed web service descriptions of selling agents.**

I. INTRODUCTION

The recent expansion of representing Web Services using agents is causing difficulties in finding specific types of Web Services. The main reason for these problems is the employed matchmaking techniques. Most of the existing techniques are based on search using string comparison, so if service providers neglect to provide sufficient or appropriate terms for the matchmaking process, the search techniques will return incomplete results. This paper addresses the problem of matching requested services to proper provider agents by making use of OWL (Ontology Web Language) [11] ontologies and the OWL reasoner RACER [2]. In the following sections we will first describe the used tools, and then introduce an implemented prototype, where an agent (matchmaker) was added to an existing agent framework (DECAF) [1], where the new matchmaker employs OWL-S for matching requests to available services.

II. DECAF

Web Services are being used massively lately, and the fact of the high distribution of the web services is causing a problem of how to handle these web services. Agents could be considered a reasonable solution for that case. Agents have the advantage of breaking down the tasks into smaller subtasks. This feature makes it easier to handle, distribute, replicate and modify the agents since they are not dependant on each other anymore. Also developing a self-centered application that has all these characteristics could be very hard for the developer in case of having non-expert developers, that is why in case of agents the developer can create a plan file that will act as the path for the agent. All these facts show that the usage of agents in that field in specific is very important. *DECAF* (Distributed, Environment-Centered Agent Framework) is a toolkit which allows a well-defined software engineering approach to build multi *agent* systems. The toolkit provides a stable platform to design, rapidly develop and execute intelligent agents to achieve solutions in complex software systems. DECAF provides the necessary architectural services of a large-grained intelligent agent communication, planning, scheduling, execution monitoring, coordination and eventually learning and self-diagnosis [1]. DECAF has been used in the suggested prototype as the framework for agents. It can be considered as the Operating System of agents. It takes care of the internal, agent to agent, communication and the assurance of populating agents without any conflicts.

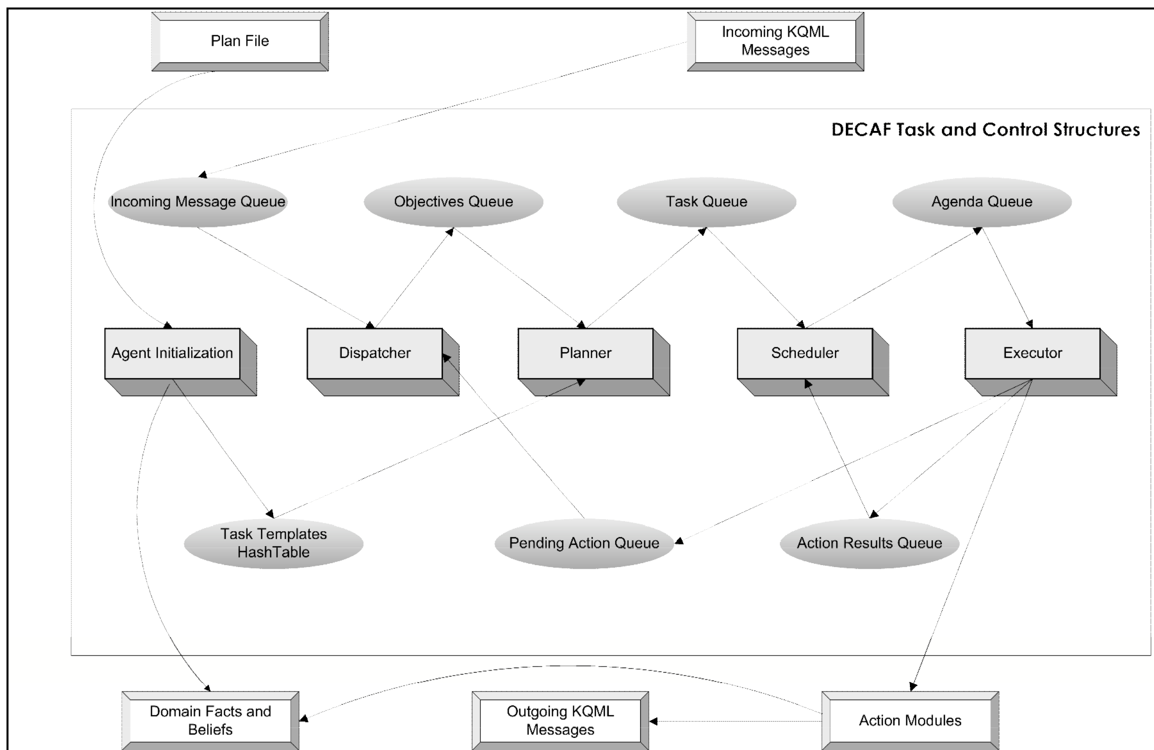

Figure 1. DECAF Task and Control Structures.

2.1 *DECAF INTERNAL ARCHITECTURE*

In this section, each module of DECAF will be discussed, elaborating on the use of that module and its function, see *Fig. 1*. This elaboration will give a clear picture of the functionality of DECAF. [6]

A. Agent Initialization

As soon as an agent starts, the first module that will run, in its own Java thread, is the initialization module and will run only once for that particular agent. The initialization module takes care of the plan file. The plan file is created in ASCII format and its main goal is to explicate the basic information flow relationships between tasks and other relationships that affect control decisions [7]. The initialization module reads the plan file, and it adds each task specified to the *Task Templates Hash table* (plan library).

B. Dispatcher

After the initialization module, the control will be passed to the dispatcher. The dispatcher will be running continuously in the background of the application waiting for an incoming KQML (Knowledge Query and Manipulation Language) message. As soon as any KQML message arrives, it will be queued in the *Incoming Message Queue*. Depending on the content of the KQML message the dispatcher will take one of two paths. In the first path, the KQML message will try to communicate as part of an existing conversation which will be recognized through the in-reply-to field. In that case the dispatcher will be searching for the equivalent action in the *Pending Action Queue,* then it continues the regular actions for that agent. The second path, by not using the in-reply-to field, will show that this KQML message is part of a non-existing conversation. If so, then the dispatcher will take care of creating a new objective and place it in the *Objectives Queue*.

C. Planner

Knowing that the objectives are now placed in the *Objectives Queue,* we then need a module that will monitor them and attach new tasks to the existing task template stored in the plan library. The planner module will perform that task. The instantiated plan will have a copy at the *Task Queue* area in the HTN format corresponding to that task, along with a unique ID and any passed provisions from the KQML messages. In the occurrence that a request comes for the same goal to be accomplished, automatically the plan template will be instantiated in the task networks with a new ID. All the plans/goal structures that have to be accomplished in response to an unsatisfied message will always remain in the *Task Queue*.

D. Scheduler

The scheduler module's main task complies with the *Task Queue* thus it is dormant until the *Task Queue* contains

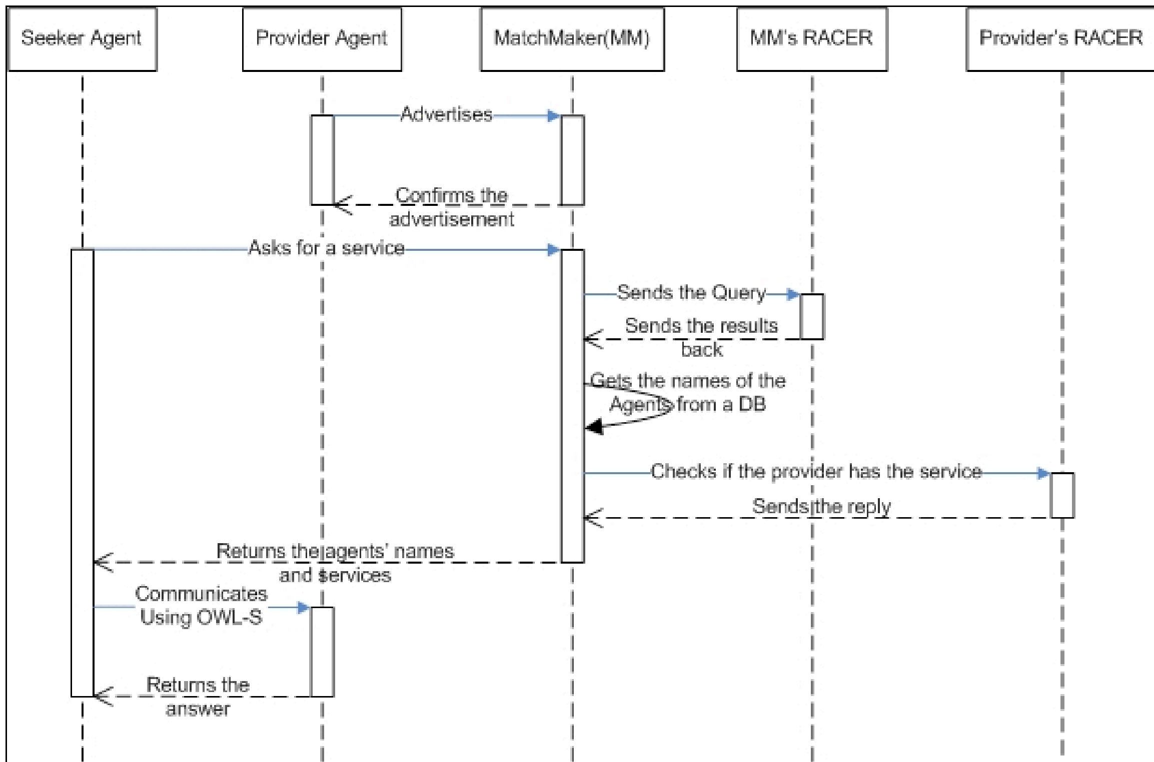

Figure 2. Sequence diagram illustrating the agent communication.

tasks. It then starts scheduling the tasks into which groups will be executed immediately, and in what order they should be executed. The execution element depends on the availability of the provisions for a particular module. Provisions could be available either from an incoming KQML message or, it could be given as an output from a different action. After that undertaking, we can tell that every time a certain provision is given, the *Task Queue* is checked in order to see if it is possible to execute any task or if there is a delay. This part of the DECAF is capturing the attention of many researchers; deciding where they want to add reasoning ability for scheduling in order to select the optimal path for task completion.

E. Executor

Finally, the executor module monitors the *Agenda Queue* and starts when it is nonempty. As soon as we have an action in the queue, the Executor module executes that task. After the execution two paths can occur. The first path would happen if everything runs smoothly and we get a result which will be put in the *Action Result Queue*. The framework will distribute the result to downstream actions which might be waiting in the *Task Queue*. After this is accomplished the executor will go back to the *Agenda Queue* and check if there are any other tasks that need to be executed or not. The executor will resolve the tasks by having a thread for each action. The second path would be performed if the action failed and did not return. In this case the framework will designate failure and shows it to the requester.

III. OTHER TOOLS

3.1 ONTOLOGIES

The definition of ontology differs from one context to the next. Generally an ontology can be considered as a set of formal descriptions of ***classes*** in a domain of discourse, where classes describe common characteristics of individuals. These descriptions specify ***properties*** of individuals, and ***value restrictions*** on properties. Ontologies together with a set of assertions about individuals or ***instances*** of classes usually define a ***knowledge base***. In reality, there is a fine line where an ontology ends and a knowledge base begins [3]. In general we can say that ontology is an explicit specification of a conceptualization where it makes classes more specific by using other classes and properties [4].

3.2 RACER

RACER (Renamed A-box and Concept Expression Reasoner) is a description logic reasoner for OWL DL that implements a highly optimized tableau calculus for very expressive description logics. It offers reasoning services for multiple T-boxes, which can be viewed as a set of OWL class declarations that specify common characteristics of individuals, and for multiple A-boxes, which represent assertional knowledge (facts) about particular individuals. A T-box and an A-box together are usually encoded as an OWL DL knowledge base [2].

The T-Box introduces the terminology, i.e., the vocabulary of an application domain, where the A-Box contains assertions about named individuals in terms of this vocabulary.

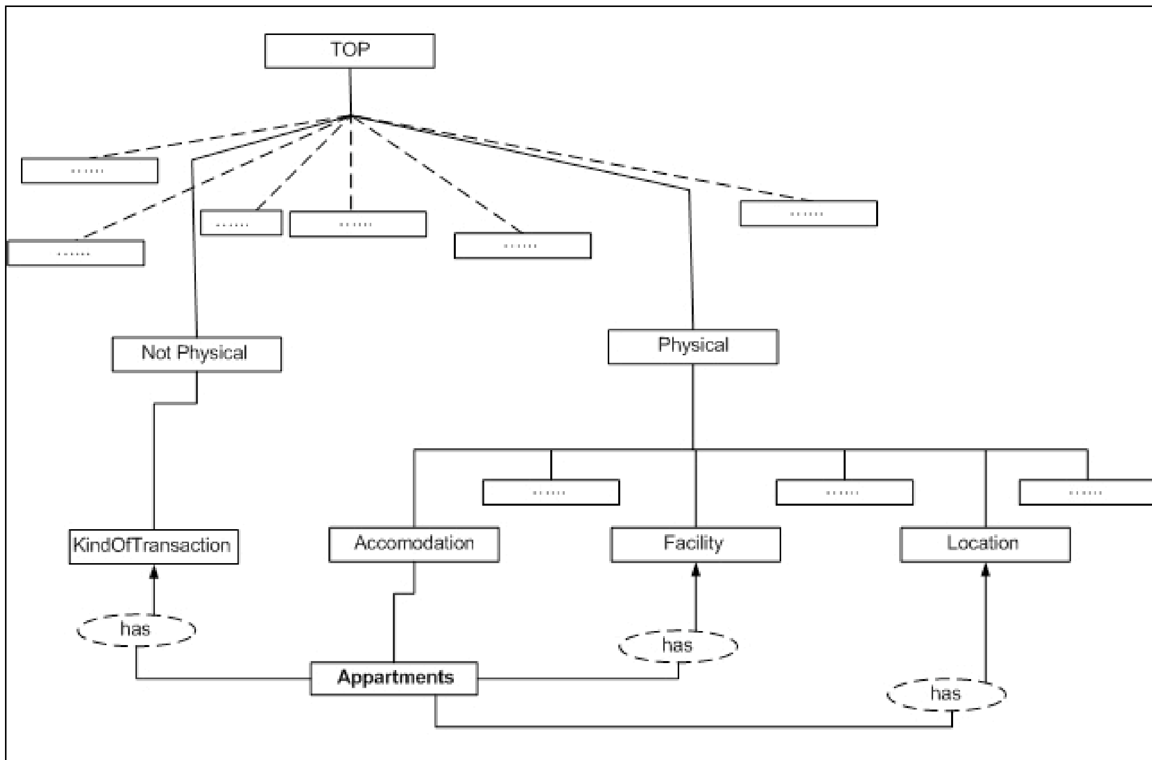

Figure 3. Upper Ontology Sketch.

Combining both the A-Box and the T-Box we get a knowledge base. [8]

RACER is used as a reasoner for provided OWL ontologies. RACER offers reasoning services such as consistency of class descriptions, subsumption between class descriptions, and retrieval of class names that instantiate known individuals. With the help of these services, seeker agents have the ability of querying the ontologies about provided Web Services. Ontologies are used to specify offered Web Services. If a certain agent is providing a service, it will specify the service using an ontology format such as OWL, when it is registering this service. In this way, the ontology specification will be the reference for the service when it is being retrieved.

3.3 OWL-S

OWL-S is an OWL-based Web Service ontology, which supplies Web Service providers with a core set of markup language, constructed for describing the properties and capabilities of their Web Services in unambiguous, computer-interpretable form. The OWL-S markup of Web Services will facilitate the automation of Web Service tasks, including automated Web Service discovery, execution, composition and interoperation [5]. The main characteristic of OWL-S that makes it interesting in our domain is the fact that it is based on OWL, which is the main language that is used in the proposed solution, which makes it coherent with the whole system. And since service descriptions are based on OWL, the ability of using RACER's inference services is an advantage. For instance, the reasoner might return the closest answer to a query in case of not having an exact answer by using subsumption techniques. Also, another advantage is the ability of dynamically changing the interface of any provided Web Service by just translating the WSDL (Web Services Description Language) into a regular OWL format that is readable by any other agent. On the other hand, when using the regular methods of publishing Web Services, the requester (client) might as well lose newly added services or parameters for existing services because its communication protocol will be based on an outdated published service and since OWL-S communicates with the WSDL at runtime then we might be avoiding the outdated Web Services' interfaces.

IV. SUGGESTED SOLUTION

After briefly mentioning the used tools and the main problem, the suggested solution (provided in the sequence diagram *Fig2*, see above) is discussed. The main scenario that was applied for this solution basically consisted of agents that represent different parties in the e-business world where a seeker would inform his/her agent to seek for a certain product or service, and that agent would be able to find the appropriate provider agent if available in the e-market through certain techniques and procedures that will be explained later on. In general, the presented solution can be divided into three main steps; an agent providing Web Services (*Provider Agent*), an agent requesting Web Services (*Seeker Agent*) and an agent which takes care of matching the seeker agent to correct provider agents (*matchmaker*).

4.1 MATCHMAKER

In order for a seeker agent to find a certain service from a provider agent, there should be a third party which takes care

of that specific issue. That third party is called the matchmaker. The matchmaker is a middle-ware agent that is provided with DECAF. It is considered as a middle-ware agent because it acts as a coordinator between the seeker and the provider agent. The matchmaker is in charge of matching requested services to proper provider agents. In the occurrence that a problem arises, the matchmaker's searching technique is based on string comparison which, as mentioned before, can cause performance problems and produce incomplete results for seeker agents. So the main part of the provided solution is to re-implement the matchmaker by replacing string comparison with reasoning based on ontologies.

The new matchmaker was re-implemented with several new features. The main feature of the new matchmaker is the ability of distinguishing the different domains that the agents belong to. That feature was accomplished by having an upper ontology; an ontology that will include everything related to the different agent domains and will connect them in a reasonable way. By any means this ontology will basically be universal where as all the matchmaker agents have to attain one in order to ground the domains and makes the matchmaking procedure easier for the matchmaker agents. Thus it will be easier for the seeker and provider agents to find their category without any conflicts or arising problems. That upper ontology acts as make-shift glue which connects all the agents' domains. For an example in Figure 3 we have a sketch of part of that upper ontology which basically emphasizes on the apartment concept. We can see the connection between the apartment and the other concepts in a very general and primitive way. It also shows the location of apartment in the hierarchy of the ontology. And by having apartment in such a general ontology, then it will be related to any other existing concept in ontology even if they were not related at all. This means that we can distinguish between related and non-related concepts under this upper ontology. As a result, when a seeker searches for a service, the upper ontology is used as a ground for communication and by getting only the related agents. What makes the general ontology a better technique than a regular database that includes all the concepts of all the domains and relate them to the agents is its ability to reason with ontologies. Another feature that was improved on the new matchmaker is the use of a database to store information about agents (e.g. their names and offered services). In comparison, the past models used to store the information in a text file instead of a database. The database is used also to keep track of the requests; while one is searching for matching agents, where for each request there is a unique ID number that is passed with the on-going events that are made to satisfy the seeker agent's request. At the end when the events are terminated and it returns to the matchmaker, the matchmaker will be able to define to which seeker agent this request belongs. Three basic services were implemented at the matchmaker in order to communicate with the other agents. These services are *Advertisement*, *asking, and deeper,* and they will be explained as follows:

A. ADVERTISEMENT

The provider agent has to register somewhere, and advertises itself in order to allow other agents to use its services, and to be known in the e-market. That is why the service Advertisement was implemented, and it expects two parameters. The first one is a set of **concepts**, which includes all the related concepts for that particular service. And these provided concepts must occur as leaves in the upper ontology (e.g., if an agent is providing apartments, and the upper ontology includes housing as a leaf, then the agent will provide housing as a concept for that service).

Obviously the agent category, which is called ontology in the DECAF world, should be already known by the provider agent, so it can know where it would belong, and where would be the best location for the file in order for some other agent that is looking for that service to encounter it. Consequently the concepts provided by the provider agent will be the key to its place in the taxonomy. The second parameter is the **category** which will be used in the protocol implementing the agent communication. The category will be used only for DECAF, instead of having two categories for the same agent, so when it is actually following the agent's plan it would know to which taxonomy that certain event belongs to.

B. ASKING

The second service is *asking*, which fundamentally addresses the search part in the matchmaker. Whenever a seeker agent starts searching for a certain service, it calls the asking service at the matchmaker. It expects two parameters, **ConceptBasedQuery** which is a general query that will be used as a filter on the upper ontology at the matchmaker's side to get all the related provider agents. The special advantage of this query is that it does not actually deal with the A-Box of the ontology; it basically queries only the T-Box. It executes the query based on subsumption, so if the same exact request was not available in the upper ontology it will return back the closest logical term based on the existing terms in the query. By these means the matchmaker will have a general picture of the request since we deal with the upper ontology at the matchmaker side only. The second parameter is named **RQL** (Racer Query Language) which contains a more specific query that will be used at the provider agent's side to check whether the filtered agents truly provide the requested service. What makes the RQL special is the ability of representing complex OWL queries, which is not possible in the case of regular queries or keyword matching techniques. The RQL deals directly with the instances in the A-Box which will give it the ability of finding the exact wanted service at the agent's side which satisfies all the requirements. On the other hand, using the ConceptBasedQuery might return the agents that do not have the exact requirements that the seeker agent was primarily seeking for. However at least it will return all the agents that could serve those services back to the originator. In other words we can consider the ConceptBasedQuery as a filter that retains only the related agents regardless of the details. Thus a lot of time will be saved seeing that the number of agents that have to be checked will decrease to the related agents only. Then the RQL can be considered as the main

query that gets only the wanted agents with all the details since it is more specific than the ConceptBasedQuery and furthermore because it deals with the A-Box at the agent's side.

C. DEEPER

As mentioned above, the Asking procedure has two parameters, one for the general filter, and another for the specific search. In order to have the specific search, the matchmaker has to take the list of agents from the first query, and go to each one of them, ask if that service exists or if it is not using the second query. This procedure is done through the service *deeper*. This service in particular will never be called by any agent but the matchmaker itself. The parameters of this service are **count** (which contains the number of the agents to be checked), **AgentsNames** (the list of filtered agents from the first query), **RQL** (which will be used in order to query the specific services in the search), **answer** (which what it will get back from the provider agent that the matchmaker was checking in the first place), **origiSender** (the name of the Agent that made the searching request), and **ID** (which will be the unique ID that will be explained later in this section). It takes the list of filtered agents from the *asking* service, and sends the RQL query to each of them and obtains the answer from them. Since the deeper searching procedure is centered at the matchmaker side, more than one agent can execute the service at the same time. Then we need to keep track of the request that goes to more than one agent by knowing to which agent it belongs. In order to solve this problem, a database is used that keeps track of the requests by giving each of them a unique number. It also provides a service that activates each time a service comes back with an answer from a filtered provider agent. It will be saved in the database in the same field of that unique number and the number will be passed on to the next agent. After checking on all the filtered agents, it returns the matched agents' names with their offered services to the seeker agent.

4.2 PROVIDER AGENT

The provider agent represents a Web Service, and makes it possible for any seeker agent to use that Web Service. It can be looked at as the supplier in the real market, but in this situation it is the supplier in the e-market where it can be used by any other agent as long as it satisfies the seeker's requirement. This agent was implemented with two main features, *deeperSearch* and *activateService*. The deeperSearch service is the service that communicates with the matchmaker, and to be more specific with the deeper service at the matchmaker side. It takes the RQL query, poses it to RACER by referring to the corresponding ontology. It replies back with the service name if available or with none if no matching services could be found on the basis of this ontology. It also takes four parameters. The first parameter is **count,** which as previously explained will be used when it sends the reply back to the matchmaker, in order to keep track of how many other agents are left in the deeper search procedure before terminating. The second parameter is **AgentsNames** where it contains the name of the other provider agents that might satisfy the requirements and have the right service as well. The third parameter is **ID** which is the identifier of this specific search. The fourth parameter is **origiSender** which has the name of the seeker agent. The other service, activateService, is used by the seeker agent after receiving the results from the matchmaker. This service makes use of OWL-S by representing the existing Web Service in an ontology format, so that a seeker agent can retrieve the parameter profile of the service and executes the Web Service call. Using OWL-S makes the representation of Web Services more flexible and dynamic because the interface of an agent might change at any given time. The service description is compatible with WSDL and agents can adapt to changed services at runtime without the need to code any of the involved agents or even to put any service down in order to change any kind of parameters. This procedure is done automatically as soon as it is changed from the provider's side. The OWL-S description specifies a communication protocol between the seeker agent and provider agent for executing the Web Services successfully.

4.3 SEEKER AGENT

The seeker agent in this scenario acts as the consumer in the real market, where it has two main functionalities. The first function is the need to communicate with the matchmaker, in order to retrieve the names of the matching provider agents that have the required service. In this service it passes the required parameters at the matchmaker side which are the RQL and the ConceptBasedQuery, and it expects the answer back through another service called getAnswer, where it expects one parameter only which is the parameter named answer. The other feature is named **activateService**, where it actually communicates directly with the Web Service of the matching agents in order to activate that specific service. The special characteristic about this is its usability of OWL-S. These parameters of the Service are captured instantly, as soon as it needs to activate the service. It pops up to the user to fill up the missing information, noting that if there is already provided information then the agent can use it without going back to the user. In the case of not provided parameters it will ask the user to fill it in, after that, the service will be activated and it will return a reply confirming that the request went through or an error if something wrong happened. Therefore this is a regular activity of an agent in the e-business using web services, but with the ability of instant and immediate changes on the web service, and a better matchmaking procedure.

To make the picture clearer, we will be illustrating a real time example that was already implemented in Java with the mentioned tools and framework. The scenario is about an agent providing a web service for renting apartments (apartment_Agent). It is providing apartments in the Downtown area of Montreal city, and they have two types of apartments, a one and half and a two and half. Another agent is providing a web service that sells bananas (banana_Agent). Finally we have a seeker agent that is searching for a one and half apartment for rental in Downtown Montreal. Initially the two provider agents register their web services with the

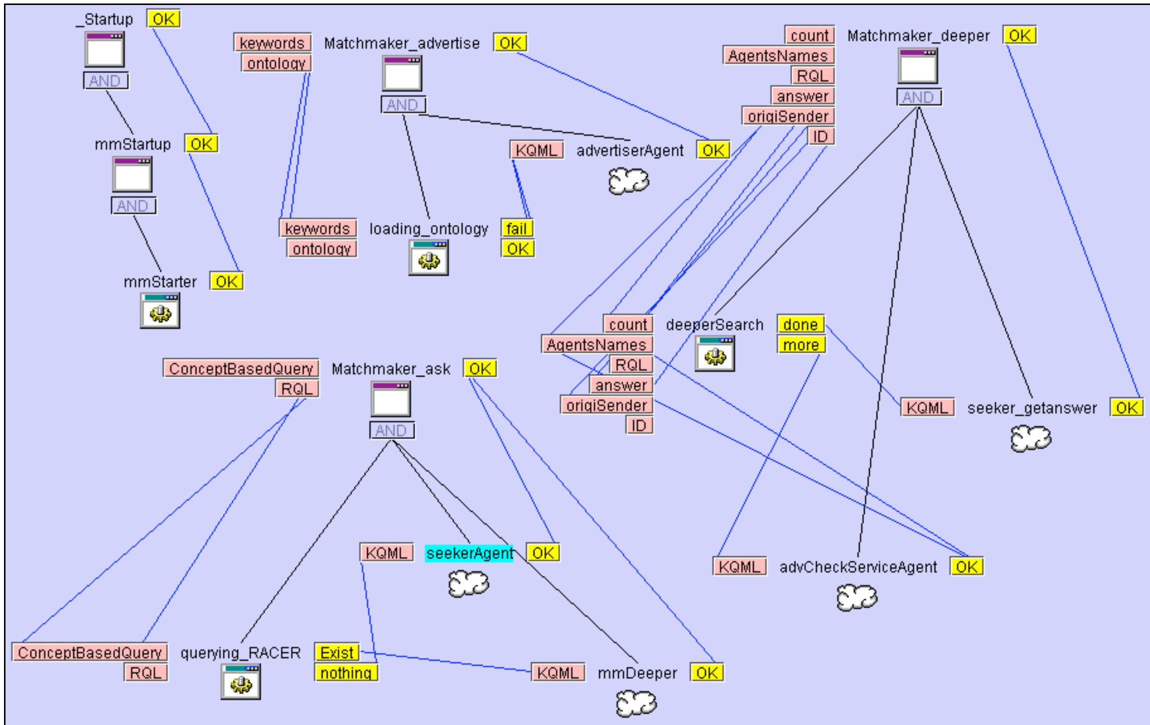

Figure 4. Plan File for the Matchmaker agent

matchmaker agent, where they provide concepts that are the guide for the matchmaker in order to fit this service in a certain category of the upper ontology with the same concepts, and ontology which is a term that is used for DECAF in order to be able to communicate with. After the registration procedure, the seeker agent will start communicating with the matchmaker by sending two parameters, RQL and ConceptBased, see Figure 4 for the plan file of the Matchmaker agent. The RQL will be used for the ABox query which will be used at the provider agent side, where the ConceptBasedQuery will be used at the matchmaker side for filtering purposes in querying the TBox as will be explained later on. The matchmaker will send the ConceptBasedQuery, which requires apartments in general, to the RACER which contains the upper ontology. So the RACER will query its TBox getting back all the registered agents that are related to apartments in general, which is apartment_Agent in our case. After filtering the registered agents and getting agents which provide the most compatible web services to the requested one, the matchmaker will send the RQL query to the apartment_Agent which will query the ABox of the apartment_Agent returning the name of the web service that exactly supports what the seeker agent is asking for. By having the name of the web service, the matchmaker returns the name of the service to the seeker agent. Afterward, the seeker agent communicated directly with the provider agent through the OWL-S interface for the web service, by getting the required parameters for activating the service, filling up these parameters from the user, then finally executing the web service.

V. RELATED WORK

There are several existing prototypes for matchmaking using different methods. In [9] they implemented a prototype for matchmaking based on Java. It uses the technology of Web Services for the communication part between agents. The server that takes care of the matchmaking procedure runs as a Web Service on top of SOAP-enabled application server. These web services use WSDL to describe its operation. The approach that was used to design such a framework was to declare concepts and functionalities that are common to different types of matchmaking servers, and to define a reference model capturing these commonalities. The developer specifies and creates a set of entity types and relationships, and by combining them we get the mentioned reference model. The reference model is created to fit the matchmaking modalities of a given e-marketplace. Beside the reference model, they created a processing model that defines the sequence of steps that the matchmaker will follow when it receives a trading intention. Another existing related approach is in [10] where they based their solution on the Semantic Web technologies. The language they used to apply the semantic web was the DAML+OIL where they used it to express service descriptions using description logic reasoners such as RACER and FaCT. Their concept of match was using subsumption in order to find the general description matches, the more specific description matches and the compatible services. This means that the approach is totally based on the TBoxes of the ontologies either in RACER or FaCT. The subsumption procedure return either equivalent concepts to the service description (S), sub-concepts of S, super-concepts of S, or sub-concepts of any direct super-concept of S. In [11]

they used the semantic description for matching the location based web services. The description logic language they used was OWL, and the reasoner was RACER. The methods of reasoning were as following: (1) first using concept-only conditions, (2) using concept-instance conditions and (3) using concept-instance conditions plus actual individuals. The reasoning was concentrated on the TBox through a re-classification by RACER and identifying the subsumption relations between the advertised service and the requested service description. Finally in [12] they also combined the semantic web with web services in the e-commerce, and they used matchmaking prototype using DAML-S based ontology and a description logic reasoner to compare the ontologies' descriptions.

VI. CONCLUSION

So far there has not been any convenient language or representation for building agents that depend on existing ontologies. All of the previous work done in those fields show that there might be a bright future pertaining to such an approach. But as we have seen, non of the related work have used the ABox in the searching technique like what we implemented in our prototype. And they did not have the idea of filtering the requests first through the upper ontology in order to avoid the overload traffic on the network by visiting each agent. In this paper, a brief description of the integrated tools was presented. Moreover, a new way to integrate ontologies with agents in DECAF has been suggested with the ability of reasoning the ontologies using RACER. The described prototype has been implemented in Java using the mentioned tools. We tested it using a selling/buying scenario with various configurations of agents. Hopefully this prototype can be a good starting point for switching to the emerging semantic web and using ontologies with mobile agents in a wider application area.

REFERENCES

[1] John R. Graham, Keith S. Decker, Michael Mersic, DECAF - A Flexible Multi Agent System Architecture, Autonomous Agents and Multi-Agent Systems. 2003.

[2]Volker Haarslev, Ralf Möller, RACER System Description. Proceedings of International Joint Conference on Automated Reasoning, IJCAR'2001, June 18-23, 2001, Siena, Italy.

[3] Thomas R. Gruber, A Translation Approach to Portable Ontology Specifications. Knowledge Acquisition, 1993.

[4] Holger Knublauch, Mark A. Musen, Natasha F. Noy. Creating Semantic Web (OWL) Ontologies with Protégé. International Semantic Web Conference, Sanibel Island, Florida, USA, October 20-23, 2003.

[5] OWL-S (OWL Web Service) 1.0 Release http://www.daml.org/services/owl-s/1.0/

[6] Keith Decker, Xiaojing Zheng, Carl Schmidt, A multi-agent system for automated genomic annotation, Proceedings of the fifth international conference on Autonomous agents, 2001

[7] M. Williamson, K. S. Decker, and K. Sycara. Unified information and control flow in hierarchical task networks. In *Proceedings of the AAAI-96 workshop on Theories of Planning, Action, and Control,* 1996.

[8] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation, and Applications.* Cambridge University Press, 2003.

[9] Marlon Dumas, Boualem Benatallah, Nick Russell and Murray Spork. *A Configurable Matchmaking Framework for Electronic Marketplaces*. Electronic Commerce Research and Applications, 3(1):95-106, Spring 2004. Copyright Elsevier Science Publications.

[10] Javier Gonzalez-Castillo, David Trastour and Claudio Bartolini. *Description Logics for Matchmaking of Services*. Trusted E-Services Laboratory, HP Laboratories Bristol, HPL-2001-265, October 30th, 2001. Also Proceedings of the 2003 ACM symposium on applied computing. Melbourne, Florida.

[11] http://www.w3.org/2004/OWL/

[12] Rob Lemmens, Marian de Vries. *Semantic Description of Location Based Web Services Using an Extensible Location Ontology.*

[13] Lei Li and Ian Horrocks. *A Software Framework For Matchmaking Based on Semantic Web Technology.*