ABSTRACT
In this paper, a new agent development framework and platform, which includes built-in features for semantic web based multi agent system development, is introduced. The first built-in support is that all agents and services in the platform use semantic web standards to represent their internal knowledge and semantic web query languages are used to query them. The second feature is the ability to discover and dynamically invoke semantic web services. Third, directory service is implemented in a way to support semantic matching of agent capabilities. Finally, an ontology service is defined to manage and translate ontologies. The ontology service allows defining mappings between platform ontologies and external ontologies and the ontology translation is done based on the defined mappings. With these features, which the existing agent development frameworks do not have, the agent development platform that we have developed simplifies the semantic web based multi-agent system development.

1. INTRODUCTION
Semantic web and agent research are evolving together. Semantic web research aims to transform the World Wide Web into a knowledge representation system in which the information provided by web pages is interpreted using ontologies. This gives the opportunity for autonomous computational entities - agents - to collect and interpret semantic content on the behalf of their users.

The idea of integrating the semantic web and agent research has already been realized and some systems have been developed. ITtalks [3] system offers access to information about activities such as talks, seminars related with information technology. ITtalks uses DAML+OIL for knowledge representation and lets agents to retrieve and manipulate information stored in the ITtalks knowledge base. The smart meeting room system [2] is a distributed system that consists of agents, services, devices and sensors that provide relevant services and information to the meeting participants based on their contexts. This system uses semantic web languages for representing context ontologies. Both the ITtalks and the smart meeting room system use a multi-agent development framework in their underlying infrastructure. For example, ITtalks uses Jackal [4] and smart meeting room system uses Jade [1]. In these systems, semantic web functionality is hard coded into the system together with the domain knowledge, because the agent frameworks used in the implementation of these systems do not have a built-in support for semantic web. For example, it is difficult for these systems’ developers to support basic semantic web functionalities such as discovering and dynamically invoking of semantic web services inside an agent or performing an ontology translation between different platform ontologies. Moreover, it requires knowledge for ordinary developers to handle the semantic web and agent technology details in addition to the application domain related knowledge. We can conclude from this discussion that there must be environments, which will simplify semantic web based multi agent system development for ordinary developers and which will support the basic semantic web functionalities.

In this paper, we introduce SEAGENT, which is a new agent development framework and platform that is specialized for semantic web based multi agent system development. The communication and plan execution infrastructure of SEAGENT looks like other existing agent development frameworks such as DECAF [8], JADE [1], RETSINA [16]. To support and ease semantic web based multi agent system development, SEAGENT includes the following built-in features that the existing agent frameworks and platforms do not have:

i) Agents created using SEAGENT handle their internal knowledge base using semantic web standards and the platform provides specifically designed interfaces to manage and query the internal knowledge without being dependent on a particular application programming interface.

ii) The directory service of SEAGENT is implemented in a way that the directory knowledge is held in semantic web standards and the directory service supports semantic matching of the agent capabilities to find the semantically similar agents.

iii) FIPA-RDF content language [6] has been used to trans-
The second layer includes packages, which provide the core functionality of the platform. The first package, called as Agency, handles the internal functionality of an agent. Agency package supports the creation of general purpose and goal directed agents. In this sense, Agency package provides a build-in ‘agent operating system’ that matches the goal(s) to defined plan(s), which are defined using HTN planning formalism [14]. It then schedules, executes and monitors the plan(s). From semantic web based development perspective, an agent’s internal architecture must support semantic web ontology standards for messaging and internal knowledge handling to simplify semantic based development. For this purpose, Agency package provides a build-in support to parse and interpret FIPA RDF content language to handle semantic web based messaging. On the other hand, Agency provides two interfaces for semantic knowledge handling, one for local ontology management and the other one for querying. Although the current version includes the JENA [9] based implementation of these interfaces, other semantic knowledge management environments and query engines can be integrated to the platform by implementing these interfaces.

Besides implementing standard services in a semantic way, SEAGENT platform provides two new services to simplify semantic web based MAS development. The first one is called as Semantic Service Matcher (SSM). SSM is responsible for connecting the platform to the semantic web services hosted in the outside of the platform. SSM uses ‘service profile’ construct of the Web Ontology Language for Semantic Web Services (OWL-S) standard for service advertisement and this knowledge is also used by the internal semantic matching engine for discovery of the service(s) upon a request. SSM and DF services are implemented by extending a generic semantic matching engine architecture, which is introduced in section 5.2 in detail. The second unique service is the Ontology Manager Service (OMS). It behaves as a central repository for the domain ontologies used within the platform and provides basic ontology management functionality such as ontology deployment, ontology updating, querying etc. The most critical support of the OMS is its translation support between the ontologies. OMS handles the translation request(s) using the pre-defined mapping knowledge which is introduced through a specific user.

The second package of the Core Functionality Layer includes service sub-packages, one for each service of the platform. SEAGENT provides all standard MAS services such as Directory Facilitator (DF) Service and Agent Management Service (AMS) following the previous platform implementations and FIPA standards. But these standard services are implemented differently using the capabilities of a semantic web infrastructure. For example, standard functionality of the DF is to store agent capabilities and return the matched agent(s) upon a capability-matching request. In SEAGENT implementation, DF uses an OWL [17] ontology to hold agent capabilities and includes a semantic matching engine to be able to return agent(s) with semantically similar capabilities to the requested ones. Similarly, AMS stores agents’ descriptions in OWL using FIPA Agent Management Ontology [6] and can be queried semantically to learn descriptions of any agent that is currently resident on the platform.

Besides implementing standard services in a semantic way, SEAGENT platform provides two new services to simplify semantic web based MAS development. The first one is called as Semantic Service Matcher (SSM). SSM is responsible for connecting the platform to the semantic web services hosted in the outside of the platform. SSM uses ‘service profile’ construct of the Web Ontology Language for Semantic Web Services (OWL-S) standard for service advertisement and this knowledge is also used by the internal semantic matching engine for discovery of the service(s) upon a request. SSM and DF services are implemented by extending a generic semantic matching engine architecture, which is introduced in section 5.2 in detail. The second unique service is the Ontology Manager Service (OMS). It behaves as a central repository for the domain ontologies used within the platform and provides basic ontology management functionality such as ontology deployment, ontology updating, querying etc. The most critical support of the OMS is its translation support between the ontologies. OMS handles the translation request(s) using the pre-defined mapping knowledge which is introduced through a specific user.
interface. Through the usage of the ontology translation support, any agent of the platform may communicate with MAS and/or services outside the platform even if they use different ontologies.

Third layer of the overall architecture includes pre-prepared generic agent plans. We have divided these generic plans into two packages. Generic Behavior package collects domain independent reusable behaviors that may be used by any MAS such as well known auction protocols (English, Dutch etc.). On the other hand, Generic Semantic Behaviors package includes only the semantic web related behaviors. In the current version, the most important generic semantic behavior is the one that executes dynamic discovery and invocation of the external services. This plan is defined as a pre-prepared HTN structure and during its execution, it uses SSM service to discover the desired service and then using OWL-S 'service grounding' construct it dynamically invokes the found atomic web service(s). Hence, developers may include dynamic external service discovery and invocation capability to their plan(s) by simply inserting this reusable behavior as an ordinary complex task to their HTN based plan definition(s). Details of this plan is not given in the paper because of space limitation.

3. COMMUNICATION INFRASTRUCTURE LAYER

Since we think that the interoperability between various multi agent systems is an important issue, we have used FIPA specifications [6] in the implementation of the communication infrastructure.

FIPA standards don't interfere with the message transport protocol inside an agent platform. Since the platform is being developed in Java language, implementing the transport protocol with a Java native API would be more advantageous. Therefore, agents in the platform communicate using Java Remote Method Invocation (RMI). Nevertheless, the integration of other transport protocols is not a complicated process once the agents use a standard interface to access the communication modules. Standard interfaces make the communication transparent of the protocol used. Additionally, to conform to the FIPA standards, Internet Inter ORB Protocol (IIOP) is used for communication between different agent platforms. The concrete realization of this protocol uses Java RMI Over IIOP.

ACL messages are represented in String as specified by the FIPA standards. Again by using generic encoder interface, encoders for different representations can be added without causing any integration problems. FIPA RDF content language is our choice for a content language. It is still in experimental status but to come up with a semantic web enabled platform, one must use the standards set by W3C. In FIPA RDF, which is a schema for FIPA-RDF content language, an action is represented by http://www.fipa.org/schemas/fipa-rdf#Action and it has three properties: act, actor and argument. Act identifies what is to be done by actor. Argument is the context specific part of an action. When the platform services are considered, queries are used significantly to question about agent and service capabilities. As a consequence, the argument of an action is generally a query when a capability is searched form DF and SSM services. Our choice for a query language is OWL-QL [5]. The choice of OWL-QL is the natural outcome of storing the agents' internal knowledge in OWL ontologies. But, OWL-QL is not yet sufficient for being used in semantic matching of agent or service capabilities. For example, a way of passing the matching degree (unit to measure the semantic relationship between searched concepts) [13] to the semantic matching engine must be found. Hence, we have extended OWL-QL to cope with this problem. The difference of extended OWL-QL is its capability of carrying semantic matching degree parameters such as exact-match, plug-in-match, subsume-match.

4. AGENCY PACKAGE

Although a SEAGENT agent’s internal architecture is similar to the previously developed goal-directed agent platforms like RETSINA [16], DECAF [8] and JADE [1], it has two differences. The first difference is that unlike others a SEAGENT agent represents all its internal knowledge using OWL language. Hence, SEAGENT provides standard interfaces to the classes of the internal architecture for handling and querying OWL ontologies. The first interface is called as OntologyManager which is defined for ontology parsing, concept and/or individual retrieving and updating. The second interface is defined for query management and called as QueryManager. In the current version, two implementations for QueryManager is provided, one for RDQL for JENA and the other one for OWL-QL standard. The concrete objects are passed to the plan code through a factory object and plan becomes independent of any specific parser and query engine implementation. The other difference is that the matching of incoming requests to agent plans is performed using a match ontology. In the current version the mapping knowledge is directly written in to the concepts. But, in future version, ontology capabilities may be used to infer the appropriate plan to satisfy the semantically defined goal request.

The overall structure of agent internal architecture is shown in Figure 2. As it can be seen from Figure 2, the internal architecture is composed of four functional modules: dispatcher, matcher, scheduler and executer. Each module runs concurrently in a separate Java thread and use the common data structures. All together, they match the goal extracted from the incoming FIPA-ACL message to an agent plan, schedule and execute the plan following the pre-defined plan structure. Plans are represented using the Hierarchical Task Network (HTN) formalism [14]. In HTN formalism a complex task is defined as compositions of primitive and possibly other complex tasks. We call the complex task, which matches to a goal as a plan. In the following, we briefly describe the role of each module within a general scenario.

When a FIPA-ACL message is put the incoming message queue by the communication infrastructure layer, the dispatcher is notified. Dispatcher then parses the message and checks whether it is reply of a previous message or not. If it is a reply message, then the dispatcher finds out the task waiting for that reply from the pending queue, sets the provision(s) for that task and puts the task to the ready queue if all the other provisions of task are set. If it is not a reply message, then the dispatcher creates a new objective, puts it to the objective queue and notifies the matcher.
Matcher is responsible for matching the incoming objective to a pre-defined plan by querying the 'Match Ontology'. The 'Match Ontology' is defined in OWL including Match and Template concepts and this knowledge is used to retrieve the MatchTemplate object. Then, the matcher creates a TaskTemplate object by setting its parameters using the returned template, puts it to the task queue and notifies the scheduler.

Scheduler gets the name of the task from the TaskTemplate and creates a ComplexTask object by getting its class definition from task structure library. The class definition takes includes the reduction schema which holds the sub-tasks of the task. Then, the scheduler interprets the reduction schema and puts the ready actions to the ready queue by creating a ReadyActionTemplate and notifies Executor. It also places the provision waiting action(s) into the pending queue, the complex task(s) to the task queue by creating a TaskTemplate object.

Executor first gets the name of the primitive task from the ReadyActionTemplate and creates an Action object using the class definition that it retrieved from the action library corresponding to the primitive task name. Secondly, it calls the Do() method of the action object. The result queue is updated using the outcome of the executed action. One important point is that if there are action(s) waiting for that outcome in the pending queue, then the related provisions of these actions are set based on the outcome. These actions are put into the ready queue if their all other provisions are already set, otherwise they continue to wait in the pending queue until all other provisions are set by different outcomes.

5. PLATFORM SERVICES
In this section, first the generic service definition interface from which all the services in the platform implement is introduced. Then, the main platform services are explained.

5.1 Generic Service Definition Interface
Each service in an agent platform must be able to handle agent communication language (ACL) messages to communicate with agents. Hence there must be a generic service interface in order to generalize this capability. In the platform, this is accomplished using an abstract service class. There may be two kinds of services, internal or external. Internal services are inside the platform and by default they communicate using RMI. The internal service logic is implemented in classes derived from an abstract class for internal services, which itself is derived from the main abstract service class. Services that are to be found in any multi agent platform such as agent management service, directory facilitator service, message transport service are examples for internal services. External services are services that are outside of any platform boundary and they communicate via Internet Inter ORB protocol (IIOP). The external service structure is realized in an abstract external service class, which is derived from abstract service class. An example for an external service is the 'Agent Federation Directory Service', which is used for providing the semantic interoperability between multi-agent systems providing services in similar domains. Agent federations and the federation directory service is not the main concern of this paper and they are discussed in detail in our other work.

A service understands ACL and FIPA RDF content. It then does the necessary actions indicated by the performative of the ACL. Extending a service’s capabilities should be easy for an agent developer, so command pattern [7] has been used. A message’s performative is the main point where a message is differentiated from another. The actor of the message (the service in this case) is another point of how a message is handled. So these two parameters form the behavioral difference in the command pattern that is implemented for message handling. IFIPAMsgCommand is the command interface. Each service has its own commands according to the performative of the message, like agent management service’s RequestCommand, directory facilitator’s RequestCommand, etc. The class model for this command interface is shown in Figure 3.

For example, when agent management service receives a message with request performative, it creates the request specific command for itself and passes the execution to it. How this is done is shown in the code fragment below:

```java
public void message(FIPATransportMsg msg) {
    // create a command according to performative
    String per = message.getPerformative();
    IFIPAMsgCommand cmd = commandFactory.getCommand(per);
    // execute the command to handle the message
    cmd.execute(message, this);
}
```

Within the concrete implementation of the 'execute' method, message is parsed and related method of the service is called passing the parsed knowledge as parameter. As a conclusion, any platform service is simply connected to the platform by implementing concrete command objects for any type of messages that may sent to the service.
5.2 General Matching Engine Architecture for Directory Facilitator and Semantic Service Matcher Services

Directory Facilitator (DF) and Semantic Service Matcher services have very similar functionality. They provide yellow-pages service to agents to find the agent(s) or service(s) with a desired capability. In this study, we applied the semantic matching concept, which is originally proposed in [13] for autonomous discovery of semantic web services into both DF and SSM services. But to apply semantic matching to both services, we have first defined a generic matcher architecture to facilitate capability matching both for semantic web services and agents. Considering platforms in which agents and autonomous services interact with each other; such a matching engine plays an important role into selecting ‘semantically right’ service or agent in a platform for a requestor. The benefits of semantic matching are shown in the case study which is applied in the tourism domain.

5.2.1 Software Architecture

The basic software interfaces of an abstract matching engine are Matcher, Ontolog, AdvertisementDB, Entity and MatchResult. We think that a capability matching engine should semantically match an Entity with other entities that are advertised. An Entity can be a service profile in a semantic web service domain or it can be a DF agent descriptor in a multiagent system. AdvertisementDB represents the group of the entities, which are semantically processed and compared to a given entity to determine if it supplies the desired capabilities. Ontolog is the generic interface that represents the primitive reasoner of a matching engine. Implementations of this component will determine ontology class relations and find superclass distances between the ontology classes with specified URIs. We implemented its various implementations in our studies and those were working on domain ontologies with OWL or DAML semantic markups.

Any implementation of the matching engine should implement the Matcher interface. This provides generalization of the different matching implementations and users of those engines will deal with only one service: ‘match’. Implementation of the Matcher will have an entity database and an ontolog. The result of each semantic match operation will be a MatchResult. A MatchResult contains the matched entity and its match degree.

We developed two concrete versions of our abstract general matching engine. As first we designed and implement an engine, which realizes capability matchmaking in semantic web services that are described by OWL-S [15] profile documents.

Object model of the matching engine is shown in Figure 4. The matcher component called SWSMatcher is an implementation of the Matcher which processes on OWL-S profile documents and perform a match operation between a request service profile and the advertised profiles. SWSMatcher works on completely over object interfaces of advertisement databases and ontolog implementations. This provides an abstraction to the matcher as it doesn’t care about the underlying semantic markup or service profile structures.

The component called OWLSAdvertisementDB is one of the implementations of the AdvertisementDB interface which parses OWL-S profile documents, creates ontology models and stores achieved profile data as Profile collection. As its name already defines, a Profile object is the object representation of a semantic web service’s OWL-S profile knowledge and it implements the Entity interface.

OwlOntolog is the reasoner of the concrete matching engine and designed to process on OWL ontologies. It parses OWL documents and finds superclass distances.

Each matched service profile is encapsulated in a MatchResult including its each parameter match degree. In software design, each match result implements a programming language specific Comparable interface to be easily and quickly sorted so service profile requester can retrieve profiles in order - from semantically most exact to least one.

Our second concrete matching engine implementation is developed for DF service. In a multiagent platform, we believe that yellow page capabilities of a directory facilitator can be significantly increased by integrating an agent description matching engine into the directory facilitator so it supports semiantically enriched service lookups for agents.

This implementation of the general matching engine -called DFMatcher- performs a semantic match on the agent services advertised on the directory facilitator. DFDescriptionDB is the matching engine’s database which stores DFDescriptions. Likewise the above concrete matching engine DFDescriptionDB is the implementation of the AdvertisementDB and each DFDescription implements the Entity interface. The reasoner OwlOntolog is same with the one in SWSMatcher and processes on platform dependent agent service concepts domain.

5.3 Ontology Manager Service

Multi Agent Systems (MAS) operating in semantic web domain naturally depend on the global system ontologies. These ontologies have to be known and to be used by the agents of MAS. Furthermore, some agents are initially dependent different ontologies and need ontology translation for being able to communicate with each other. So, it is very critical to provide a central ontology management and translation
service in the platform working on semantic web environment. OMS service of the SEAGENT platform plays this role within the platform.

OMS's internal architecture, especially ontology mapping architecture is similar previously developed MAFRA Framework which is operating within the KAON (Karlsruhe Ontology and Semantic Web Tool) environment. Of course KAON environment provides more sophisticated support for distributed ontology management and versioning than OMS implementation. But our main purpose is to couple the ontology management environment to the agent platform, not to develop general purpose semantic web tool suit. By this way platform administrator can add new ontologies and/or define ontology mappings at any time of the platform life cycle. Additionally agent of the platform can query the service at any time using the ACL semantic.

OMS provides tree types of functional support to the platform. These are management, mapping and translation supports. Each functional support is connected to the platform by implementing generic IFIPAMsgCommand interface.

Initial stage of the ontology management is the introduction of the platform ontologies to the OMS. This stage is executed by three main classes and their collaboration can be defined as follows. Ontologies are selected by platform administrator through an admin console. In SEAGENT platform, admin user interface console is again implemented as a service hosted in the admin server and it communicates with other services through the FIPA ACL. Implementing admin console as a service makes our platform very flexible in terms of adding new interface requirements and makes it possible to manage distributed platform services at run-time. Admin console passes the selected ontology uri(s) to OMS via communication infrastructure. This uri's are directed to PlatformOntologyManager object by the command object. PlatformOntologyManager is a general container that is responsible to manage the platform ontologies. To be able to monitor all ontologies, it creates a OntologyManagerOntology (OMO) object for each ontology to hold the metadata (identity, version, url etc.) of the ontologies and then store these OMO(s) using the KBSimulator. KBSimulator is capable to transfer the object(s) to their OWL representation and store this knowledge using the JENA API. After the ontologies are introduced, they can be managed by admin console via ontology management support in which new ontology can be inserted or previous ones can be updated or deleted.

To provide translation, administrator has to define necessary mappings between the required ontologies. Mapping knowledge is generated by the admin console and then sent to the OMS. To execute mapping, console requests the OMO's of the platform ontologies from OMS to present them. Admin chooses two ontologies for mapping and then console retrieves the actual ontologies using uri knowledge encapsulated in the related OMO object. To parse ontology, console uses OntologyManager object which is mainly responsible of creating the model of the ontology and provides methods to query the created model. To represent the ontology model as a user interface, OntologyManager is able to create Container object(s) that hold the meta knowledge of the ontology concepts. These Container objects are used to illustrate the ontology model in the user interface. Admin chooses the concepts and sub properties for mapping(s) and console creates OWL individuals for each mapping to send them to the OMS within a FIPA ACL message. When OMS' command receives this message, it passes it to MappingManager object which creates Map object(s) and store them using the KBSimulator.

Translation request comes to the related command of the OMS within a ACL which includes individuals for translation and the target ontology's uri. This knowledge is passed to OntologyTransfer object which uses the MappingManager to find the Map objects for each individual. It then creates new individuals from Map object to reply the translation request.

6. CASE STUDY AND CONCLUSION

The platform's overall architecture has been implemented and it is operational. To evaluate the platform, we have developed an experimental but a realistic application in tourism domain. The semantic features are tested in two scenarios that make up the application.

The tourism ontologies are defined consisting of the concepts in the domain: hotel, room, price, etc. before the platform is instantiated. The first scenario is finding the right hotel agent. The customer is interested in outdoor activities and specifically wants to have a vacation in a hotel with mountain biking activity. Mountain biking concept is declared as the sub-class of outdoor activity in the activities ontology and the hotel agents advertise their activities by the concepts from that ontology to the DF. The advertisements made by the hotels (e.g. mountain biking) are the values of service-type property in agent-descriptions stored by the DF. The agent-description and service-type are concepts defined in FIPA Agent Management Ontology.

The user agent gathers the requirements from the customer to contact the most suitable hotel agent. The service type of the agent to be searched is "mountain biking", however the customer also wants to find hotels offering any outdoor activity. Therefore the degree of match specified in the query is "plug-in" which in other words means that the advertised service type can be mountain biking and any of its super classes. Thus outdoor activity is implied by the degree of match in the query. When the DF is queried to find such a hotel agent, the results returned contain the matching degree. Agents giving services of type mountain biking are marked as exact matches and all others are plug-ins of it. As it is seen from the results, a general degree of match (here plug-in) subsumes a more specific one (here exact). The popular FIPA-compliant agent platforms do not implement a semantic matching capability in their DF's hence leaving it to the user to implement such a facility.

In the application, a hotel agent speaking in a different ontology than the customer agent is chosen on purpose therefore the second scenario is the ontology translation between the hotel agent and the customer agent. The customer agent wants to ask the price per night but the customer refers to it as "Expense#Cost" and the hotel as "Income#RoomPrice". Seeing the difference between their ontologies, the customer
agent requests for a mapping from its price concept (Cost) in its ontology (Expense) to the same concept (RoomPrice - yet not known by the customer agent) in the hotel agent’s ontology (Income) from the Ontology Management Service. Income#RoomPrice returns from the service thus letting the agent continue with its bargaining plan (if any).

In conclusion, a user can develop agent programs utilizing the power of semantic web with the facilities provided as default in SEAGENT Framework. This has been the most important goal in developing the framework from the beginning.

7. ADDITIONAL AUTHORS
Additional authors: Erdem Eser Ekinci (Ege University, email: erdemeserekinci@hotmail.com), Önder Gürcan (Ege University, email: gurcan@staff.ege.edu.tr), Geylani Karlas (Ege University, email: geylani@bornova.ege.edu.tr), İnanç Seylan (Ege University, email: seylan@staff.ege.edu.tr) and Ali Murat Tiryaki (Ege University, email: tiryaki@staff.ege.edu.tr).

8. REFERENCES