Engineering a Multi Agent Platform with Dynamic Semantic Service Discovery and Invocation Capability

Oguz Dikenelli[1], Özgür Gümüs[1], Ali Murat Tiryaki[1], and Geylani Kardas[2]

[1]Ege University, Department of Computer Engineering,
35100 Bornova, Izmir, Turkey
{oguzd,gumus,ali_tiryaki}@staff.ege.edu.tr
[2]Ege University, International Computer Institute,
35100 Bornova, Izmir, Turkey
geylani@bornova.ege.edu.tr

Abstract. In this paper, an agent framework, which provides a build in support for dynamic semantic service discovery and invocation within the agent's plan(s), is introduced. To provide such a support, a generic plan structure is defined for semantic service integration. Developer can reuse this generic plan and add it to any agent plan as a task to create semantic service enabled plan(s). The platform executes this kind of plan(s) with its build in support. Also, a case study is developed to show the effectiveness of this approach in terms of integrating agents with web services.

1 Introduction

Web Services can be considered as pluggable software components with language and platform independent interfaces. Hence, other components can use the web services dynamically through the published interfaces. This machine-readable interface description of the web services gives opportunity to autonomous agents to use them when they demand the functionality provided by the service. But, it is not clear how agents will decide to use a web service and how they will discover and invoke the right service in addition to its own duties.

In the literature, Semantic Markup for Web Services (OWL-S, formerly DAML-S) [16] has been extensively used to implement semantic based service discovery and execution of composite services. For example, some semantic service matching engines have been implemented based on the DAML-S profile ontology in [7] and [9]. Also, some works have been conducted to execute composite services using DAML-S process ontology in [11], [12] and [13]. Moreover, some integrated architectures have been proposed based on matchmaking in [10] and [15] and brokering in [14] to handle discovery and invocation together. However, the basic problem still remains: How can a developer build an agent system that uses web services and how he/she integrates and synchronizes these services' execution with other agent related task(s). To solve this problem, first of all agent platform must be developed that supports the execution of semantic service integration type of task(s). In this paper, we introduce an implemented agent platform that provides such a support. For this purpose, a generic plan structure is defined using the Hierarchical Task Network (HTN) formalism for semantic service discovery and dynamic invocation. Then, agent's internal architecture is specially designed to execute plan(s) that includes the task(s) derived from this generic plan. Also, a service is implemented within the platform to give the semantic service matching service to the agents in the platform.

The paper is organized in the following manner. In section 2, the general architecture of the platform is given. A design approach for the agent and semantic service integration is discussed in section 3. Section 4 introduces the internal architecture of the agent designed for execution of semantic service enabled plans. The generic plan structure for semantic service integration is explained in section 5 and the last section gives a case study and concludes the work.

2 Software Architecture of the Agent Platform for Semantic Service Integration

To be able to call a multi agent platform as semantic service enabled, agents of the platform must be capable of executing plan(s) that include specific task(s) for semantic service integration. We call this kind of plans as semantic service enabled plans. It is clear that agent(s) requires a specific support from the platform to execute this kind of plans. So, a conceptual architecture, that executes semantic service enabled plans, must be defined first. Fig. 1 illustrates packages of such a conceptual architecture in a layered style. Of course, a general purpose MAS platform requires additional services such as Directory Facilitator, Agent Management Service and such an abstract architecture is defined in FIPA's Abstract Architecture Specification [5]. But, our purpose is only defining and implementing packages which are responsible to execute semantic service enabled plans. So, any platform can be made semantic service enabled by just implementing package responsibilities defined in this paper.

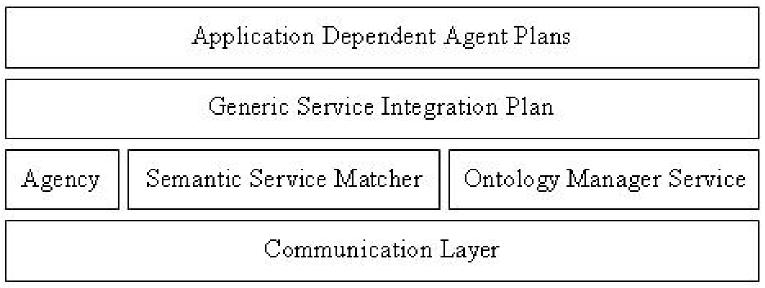

Fig. 1. Packages of platform's software architecture

Bottom layer includes the communication layer that is responsible of abstracting platform's communication infrastructure implementation. It implements FIPA's Agent Communication and Agent Message Transport specifications [5] to handle agent messaging. This layer has been developed and used as part of our FIPA compliant agent development framework [2], [3] and then reused in this implementation.

In our implementation, Agency package includes necessary infrastructure to generate general purpose and goal directed agents similar to JADE [1] and DECAF [6] platforms. It provides a build in agent operating system to schedule, execute and monitor agent plan(s) which are defined in HTN formalism [8]. To execute semantic service enabled plans, we have defined a generic HTN structure that is specialized based on the domain requirements. Naturally, this plan can be executed by the agency as the other HTN plans and it can be combined with other plan(s).

Semantic Service Matcher (SSM) can be considered as a bridge between platform and web services hosted outside of the platform. SSM uses service profile concept of the OWL-S ontology for service advertisement and this advertisement knowledge is used by internal semantic service matching engine for discovery of the services upon request. We have used the service capability matching algorithm originally proposed in [9] for semantic service matching engine implementation. Since our discussion on Generic Service Integration Plan in section 5 sometimes uses the concepts of this algorithm, we briefly introduce the concepts used in the algorithm in this section. Capability matching algorithm matches OWL-S profile's input and output concepts of the advertisement and request. Input and output concepts are taken values from specific domain ontology(ies) and the match degree is determined by the minimal distance between the concepts of these ontology(ies). Formally if out_{AD} and out_{REQ} represent the outputs of the advertisement and the request respectively, algorithm defines four types of match on outputs:

- *exact match* when out_{AD} and out_{REQ} are equal or out_{REQ} is subclass of out_{AD}

- *plug-in match* when out_{AD} is more generic than out_{REQ} (out_{AD} subsumes out_{REQ})

- *subsumes match* when out_{AD} is more specific than out_{REQ} (out_{REQ} subsumes out_{AD})

- *fail* when neither of the conditions above satisfies

The scoring function is ordered as exact $>$ plug-in $>$ subsumed $>$ fail. Same can be applied to inputs but matchmaker prefers output matches over input matches and input match scoring is used to sort equivalent output matches. In our implementation, SSM is queried by the platform's agent(s) with FIPA RDF [5] content language using OWL-QL [4] query syntax in argument part of the message. To be able to use the match degree within the QWL-QL, we have extended the QWL-QL for querying the matching of semantic capability. Details of this extension are discussed in section 5.

Ontology Manager Service (OMS) behaves mainly 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. But, the most critical support of the OMS for service integration is its translation support between the service or domain ontologies. OMS provides a user interface to define mappings between the selected ontologies and then handles the translation request(s) using the mapping knowledge. Through the usage of the ontology translation support, any agent of the platform may discover and/or invoke the services even if they use different ontologies.

"Generic Service Integration Plan" includes pre-defined tasks for dynamic semantic service discovery and invocation. This generic plan executes standard tasks such as service discovery based on the service capability, selection of matched services and invocation of selected service(s) in a pre-defined order and under the pre-set conditional assumptions. The details of this plan are discussed in section 5. But, it has to be emphasized that behavior of some task(s) may need to be modified depending to the application conditions. In this kind of situations, developers have to modify the specific tasks of this plan to satisfy the application requirements.

Top layer includes the application dependent plans that are defined by agent developers to satisfy the system's goal(s). To make these plans semantic service enabled, "Generic Service Integration Plan" can be added to the plan as service task.

3 A Design Approach for the Agent and Semantic Service Integration

To be able to integrate semantic web services with agents, some well defined activities are needed within the agent development methodology. These activities are conducted to build up the elements of the conceptual architecture defined in section 2. In the following, two activities are defined for this purpose.

Activity 1: *Define the OWL-S based service profiles of each domain specific services that may be used by the agents.*

In the design phase, we must know interface of external services to be able to write the actual agent plans that include service integration. So, an OWL-S service profile is defined for different types of domain specific services that agents may use. These profiles are stored in and build up the knowledge base of the SSM. External services are advertised themselves to the SSM using these pre-defined service profiles. The problem occurs when input and output parameters of external service interface and profile take values from different ontologies. In this case, service provider must define the mappings between ontologies of these parameters using the transition service of the OMS and agents translate profile values to the actual service interface values before the invocation. In this paper, we do not consider this case since we have not integrated the OMS implementation with the platform yet.

Activity 2: *Specialize the Generic Service Integration Plan for each required service(s).*

Application dependent agent plans may need one ore more external services whose profiles are defined in activity 1. So, developers first identify such services in this activity, then "Generic Service Integration Plan" is specialized for these services using the defined service profiles. Finally, specialized service plans are integrated with the actual application dependent plans.

4 Agent's Internal Architecture

As we stated in section 2, our agent's internal architecture executes plans represented with HTN formalism. HTN structure consists of two types of tasks. Complex task includes a "reduction schema" knowledge that defines the decomposition of the complex task to the sub tasks. The second type of tasks is the primitive tasks (actions) that can be executed by the internal architecture directly. Each task also has "provision/outcome links" that are used to propagate values between the tasks. So, internal architecture dynamically opens the complex task using the "reduction schema" knowledge, identifies input/output values of each task with "provision/outcome links", executes primitive tasks and propagates output values other dependent task(s).

In addition to complex and primitive tasks, we have defined a new task type called as "service" task to execute semantic service enabled plans. "Service" task is different because it always takes values from OWL-S profile concepts. Moreover, input and output concepts of the requested service(s) are mandatory values for "service" task to be able to discover the requested service(s). In HTN formalism, complex and primitive task types propagate data values through its provision link. Similarly, "service" task can take values through provision link if it is dependent to some task(s) which produce the required data. But if input and output of the requested service are not provided by other task(s), it must be provided as constant. So, "service" task is responsible to collect all of the values of input and output concepts from provision link and internal constant values to pass them to subtasks. Hence, this task is handled differently by the internal architecture when it is encountered.

The overall structure of agent's internal architecture is shown in Fig. 2. This architecture is specially designed to execute semantic service enabled plans. But, of course, it can also execute HTN structure(s) that includes only complex and primitive tasks. As it can be seen from Fig. 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 uses 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 predefined HTN structure. In the following, we briefly explain responsibilities of each module during plan execution with an emphasis on semantic service integration.

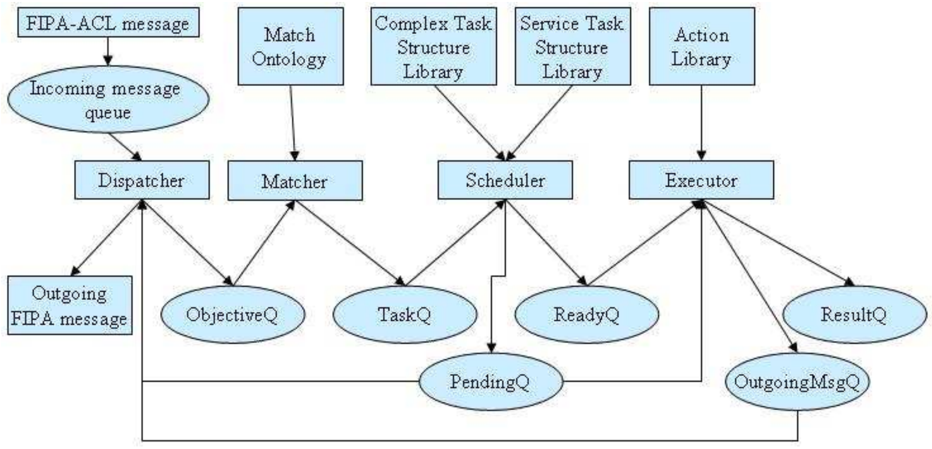

Fig. 2. Agent's internal architecture

When a FIPA-ACL message is put into 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". There can be two kinds of plans. They are called as service integration plan and ordinary plan. The service integration plan aims only semantic service discovery and invocation. The ordinary plan may include "service" task(s) and becomes semantic service enabled plan or it includes only complex and primitive tasks. The "Match Ontology" is defined in OWL including Match and Template concepts and the method of *QueryManager* interface returns the *MatchedTemplate* object to the matcher. Matcher identifies the type of the plan from *MatchedTemplate* object and creates a *ServiceTemplate* object for service plan or *TaskTemplate* object for ordinary plan by setting its parameters using the returned template. It then puts the created object to the task queue and notifies the scheduler.

Scheduler works differently for complex task and "service" task. If it gets a *TaskTemplate* from the task queue, it understands that it is a complex task. Then, it gets the name of the task from the *TaskTemplate* and creates a *ComplexTask* object by getting its class definition from task structure library. This *ComplexTask* may include "service" task(s). The class definition taken includes the reduction schema which holds the subtasks 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 and the complex task(s) to the task queue by creating a *TaskTemplate* object. If it finds a "service" task in the reduction schema, it creates a *ServiceTemplate* object and puts it into the task queue for execution.

If scheduler gets *ServiceTemplate* object from the task queue, it gets its task structure from service structure library. Service structure library holds only service integration plans that are derived by reusing our generic service integration plan structure. At this point, scheduler creates *ServiceTask* object, it gets OWL-S profiles input and output concepts from *ServiceTask*, and then it passes this knowledge as a parameter to the sub-task found in the reduction schema. Sub-tasks are handled in a same way of complex task scheduling.

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, 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 Generic Plan Structure for Semantic Service Integration

In this section, we introduce the structure of the generic plan that is specially designed for semantic service integration. The workflow of the plan can be described as follows: When an agent requires executing semantic service, it first must discover the desired service using SSM. After that, it must select the most suitable service among the discovered services. Finally, the selected service is invoked directly communicating with its providers. The HTN structure for this workflow is illustrated in Fig. 3. Each node in this HTN structure represents a task of HTN formalism. Provision links are located in the left side of the node and outcome links are in the right side. The sub-task(s) of a complex task is represented with a line drawn between them. The responsibility of each task is written inside of the node.

The top level task is called as "execute semantic web service". This task is a "service" task that may be included to any plan to make it semantic service enabled. Also, "execute semantic web service" tasks can be connected to each other to create composite semantic web services.

"Execute semantic web service" task must include input and output parameters of desired semantic web service and match degree to discover the service. These parameters can be passed through the provision link or defined as constant(s) during the creation of the real plan derived reusing the generic structure. "Execute semantic web service" task propagate the required parameters to its

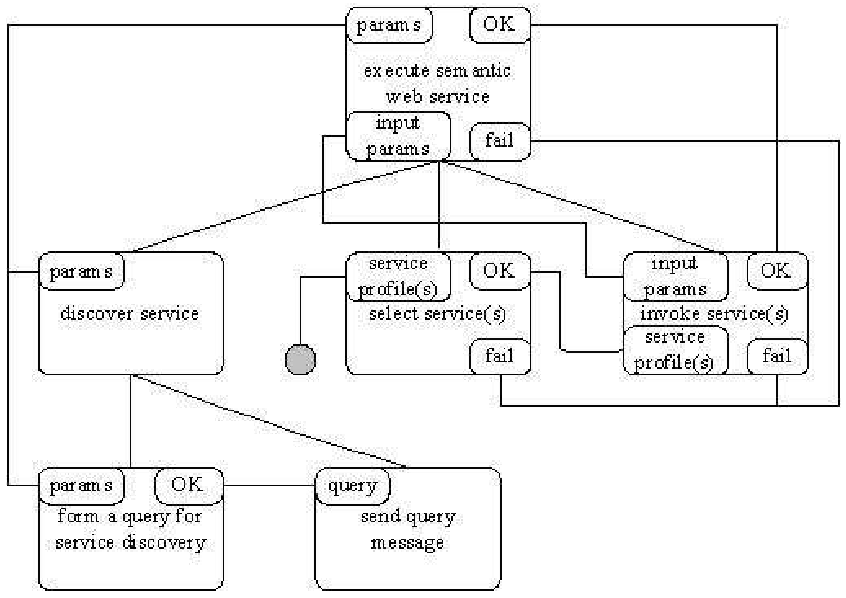

Fig. 3. Generic plan structure for semantic service integration

sub-tasks using the provision link structure. After the all tasks are executed, the result of the "execute semantic web service" task is propagated to other task(s) with OK outcome. If a desired service can not be found or any problem is occurred during the service invocation, it ends with fail outcome.

First sub-task of the "execute semantic web service" is called as "discover service" task which is responsible of discovering the service with the desired capability. It takes input and output parameters of desired semantic web service and match degree from "execute semantic web service" task. "Discover service" task includes two primitive tasks (actions) named as "form a query for service discovery" and "send query message" to inquire the SSM.

"Form a query for service discovery" action inherits the provisions of "discover service" task. This action forms a query using the parameters passed through its provision to discover the desired semantic web service. This query must include input and output parameters of the desired service and a match degree for each parameter since SSM uses this knowledge to semantically match the requested service with the advertised ones. As it is said before, our platform uses OWL-QL to query the SSM for service discovery. Since SSM knows only the OWL-S ontology and match degree is not defined in OWL-S ontology, it's not possible to specify match degree in OWL-QL for querying OWL-S profiles. So, we have extended OWL-QL to be able to prepare queries that include match degree for semantic service discovery. This extension is called as OWL-QL-S. An OWL-QL-S query may include an exact-match parameters list, a plug-in-match parameters list and a subsume-match parameters list. These lists contain URI references that occur in the query, and no URI reference can be an item of more than one of these lists. Thus, an OWL-QL-S query that prepared to discover OWL-S semantic web service(s) is capable of specifying the match degree that will be accepted for every input and output parameter.

"Send query message" action takes the OWL-QL-S query as provision and prepares a FIPA-ACL message to discover the desired service. Then it sends this message to the SSM. FIPA-RDF content language is used to transfer OWL-QL-S query. So, OWL-QL-S query is located in the argument property of the FIPA-RDF action.

"Select service(s)" action is executed when a reply message, which includes matched service profiles, is sent by the SSM. The matched service profiles are passed to this action as external provision. "Select service(s)" action is responsible of selecting the most appropriate service(s) among all the sent ones. In our implementation, SSM returns the matched services by sorting according to match degrees and "select service(s)" selects the first one. But, this task may vary depending to the overall requirements of the plan. For example, an application may require invoking all exactly matched services. In this kind of situation, plan developer has to modify the original action implementation according to the application requirements. At the end of the action, selected service profile(s) is sent with the OK outcome to the "invoke service(s)" action. If no service is selected, "select service(s)" action and consequently "execute semantic web service" task end with fail outcome.

"Invoke service(s)" action takes the selected service profile(s) and values of service's input parameters through the provision links and invokes selected service(s). This version of our implementation is capable to call atomic semantic web services. In other words, invocation of composite semantic web services is not supported yet. To invoke an atomic semantic web service, first of all, the URI of the WSDL document and operation name of service must be obtained from the corresponding OWL-S grounding document. WSDL document contains all the information that is required to invoke a web service dynamically such as network address, operation name, types of input and output parameters etc [17].

6 Case Study

To give ideas in a more concrete way, we designed an agent based information system prototype for tourism domain. We considered only a single scenario in which traveler tries to find and reserve a suitable hotel room for his/her holiday. The system includes a traveler agent that interacts with outside semantic web services to satisfy the requirements of this scenario.

Following our design approach, we first identified possible external domain service profiles that traveler agent can use to satisfy its goal. First of all, this system requires a service to find the hotels which satisfy travel preferences of the user. So, the first service profile is defined for this service and it takes "activity" and "location" as service input parameters and returns "hotel" individuals as output. We have implemented three actual services of this type and registered them to SSM by selecting different “activity”, “location” and “hotel” concepts from related domain ontologies. Second service is defined for querying room availability and it takes “date” as input parameter and returns “room availability” as output. We have also created three actual services of this type and registered them to SSM. It must be pointed out that knowledge about the “hotel” individual is stored in the “contactInformation” concept of OWL-S profile during registration of this service and then used in “find a room” service task as service selection criteria. Final service is designed to make reservation. It takes “date” as input parameter and returns “reservation result” individual as output.

After we have defined the service profiles, we try to model traveler agent’s plan that uses the defined services to satisfy the scenario at hand. This plan’s HTN structure is shown in Fig. 4. As shown in the figure; the plan includes “three” service tasks that are defined considering the service profiles identified as the first activity. At this point, all of these service tasks are created from the generic service integration plan by specializing its actions based on the requirements of the plan.

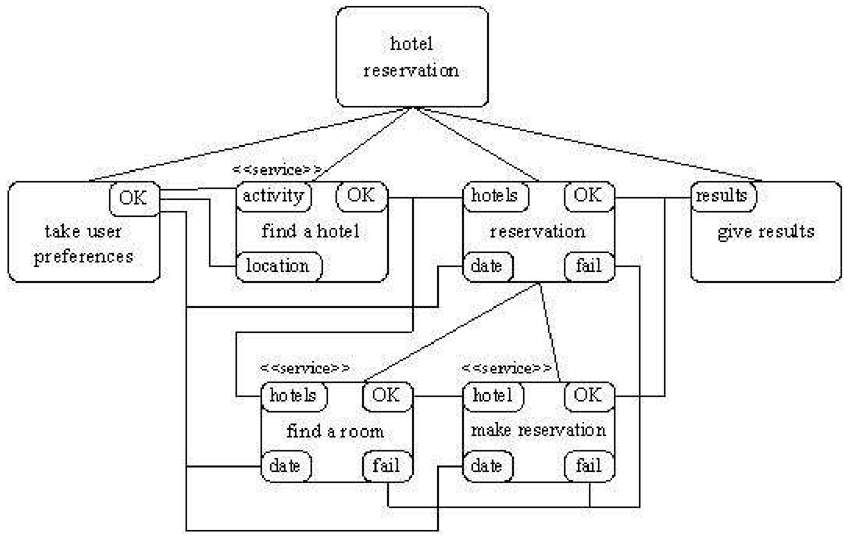

Fig. 4. Hotel reservation plan structure

First action of the plan finds the user’s activity and location preferences using the predefined preferences ontology and passes them to the “find a hotel” “service” task for the execution of the “service” task. Since this task reuses the structure of the generic service integration plan, it first executes the “discover service” complex task of the generic plan. Subtasks of “discover service” complex task are reused as is since they require only the input and output parameters of the searched service. “Select service(s)” action is used as is also since SMM sends the matched profiles as sorted. It takes the matched service profiles send by SMM and pass them to the “invoke service(s)” action. “Invoke service(s)” action specialized to invoke the services starting from the top of the list using the values of input parameters. If an invoked service returns a list of hotels successfully, it stops and passes the list of hotels to the next task.

The found hotels are passed to the “reservation” task which includes two “service” sub-tasks, one for finding an available room and one for making reservation. These sub-tasks also reuse the structure of the generic service integration plan. They use sub-tasks of “discover service” complex task as is. However, “find a room” “service” task specializes the “select service(s)” action to select the services that provided by one of the given hotels and “make reservation” “service” task specializes the same action to select the service provided by the given hotel that has an available room. “Find a room” “service” task specializes the invoke service(s) action that continues to invoke selected room availability services until it gets an output indicating an available room. Upon the completion of service invocation, “find a room” “service” task passes the found hotel that has an available room to the “make reservation” “service” task for realizing the reservation. This “service” task uses “invoke service(s)” action as is since only one service is passed to this action from “select service(s)” action. At the end, reservation details are given to the client.

7 Conclusion

Developers can create semantic web service enabled plans using the support provided by the architecture introduced in this paper. Two activities should be performed to develop plans with such a capability. First, service profile of each domain specific service should be defined with use of a service ontology. Those services will be used by platform’s agents in their plans. Second, for each required service, the plan called “Generic Service Integration Plan” should be specialized. We believe that the above mentioned support simplifies the semantic service based multi agent system development and bridges the gap between the agent and semantic service worlds.

References

1. Bellifemine, F., Poggi, A., and Rimassa, G.: Developing multi-agent systems with a FIPA-compliant agent framework. Software Practice and Experience, 31 (2001) 103-128.
2. Erdur, R.C. and Dikenelli, O.: A standards-based agent framework for instantiating adaptive agents. In Proceedings of The Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2003), pages 984-985, ACM Press, 2003.
3. Erdur, R.C. and Dikenelli, O.: A FIPA-Compliant Agent Framework with an Extra Layer for Ontology Dependent Reusable Behaviour. In Proceedings of Advances

in Information Systems, Second International Conference (ADVIS 2002), LNCS 2457, Springer, 2002.

4. Fikes, R., Hayes, P, Horrocks, I.: OWL-QL - A Language for Deductive Query Answering on the Semantic Web. Knowledge System Laboratory, Standford University, 2003, avail-able at http://ksl-web.standford.edu/KSL-Abstracts/KSL-03-14.html
5. FIPA: FIPA Specifications, http://www.fipa.org
6. Graham, J.R., Decker, K.S., Mersic, M.: DECAF - A Flexible Multi Agent System Architecture. Journal of Autonomous Agents and Multi-Agent Systems, 7, 7-27, 2003.
7. Li, L. and Horrocks, I.: A Software Framework for Matchmaking Based on Semantic Web Technology. In Proceedings of the Twelfth International Conference on World Wide Web, pages 331-339. ACM Press, 2003.
8. Paolucci, M. et al.: A Planning Component for RETSINA Agents, Intelligent Agents VI, LNAI 1757, N.R. Jennings and Y. Lesperance, eds., Springer Verlag, 2000.
9. Paolucci, M., Kawamura, T., Payne, T., R., Sycara, K.: Semantic Matching of Web Services Capabilities. In Proc. of the International Semantic Web Conference (ISWC'02), Springer Verlag, Sarddegna, Italy, June 2002.
10. Paolucci, M. and Sycara, K.: Autonomous Semantic Web Services. IEEE Internet Computing, September - October 2003, Published by the IEEE Computer Society
11. Sheshagiri, M., desJardins M., Finin T.: A Planner for Composing Services described in DAML-S. Workshop on Planning for Web Services, Trento, 2003.
12. Sirin, E., Hendler, J., Parsia B.: Semi-automatic Composition of Web Services using Semantic Descriptions. Web Services: Modeling, Architecture and Infrastructure workshop in conjunction with ICEIS2003, April 2003.
13. Sirin, E., Parsia, B., Wu, D., Hendler, J., Nau, D.: HTN planning for web service composition using SHOP2. Journal of Web Semantics, 1(4):377-396, 2004
14. Sycara, K., Paolucci, M., Soudry, J., Srinivasan, N.: Dynamic Discovery and Coordination of Agent-Based Semantic Web Services. IEEE Internet Computing 8(3): 66-73 (2004)
15. Sycara, K., Paolucci, M., Ankolekar, A., Srinivasan, N.: Automated discovery, interaction and composition of Semantic Web Services. Journal of Web Semantics, Elsevier, pp. 27-46, 2003
16. The OWL Services Coalition: Semantic Markup for Web Services (OWL-S), 2004, http://www.daml.org/services/owl-s/1.1/
17. W3C: Web Services Description Language (WSDL) 2.0, http://www.w3.org/TR/wsdl20