Semantic Web Services: Service Discovery and Invocation Planning

Kati Limapichat, **Sukasom Chaiyakul**,
Avani Dixit, and **Ekawit Nantajeewarawat**, Non-members

ABSTRACT

With the expanse of internet, web programmers have wide choice of web services available to them. A need arises for automatic discovery of required web services and construction of an appropriate sequence of invocation thereof. In this paper, we present a framework for automation of this task based on currently emerging technologies such as ontological knowledge bases, OWL, OWL-S, WSDL, Description Logic (DL), etc. Background-knowledge ontologies are created based on which semantic meanings of web services can be given through OWL-S. An agent employs OWL-S API to extract web service metadata, and applies a DL inference engine, called Racer, for reasoning with the metadata with respect to given background knowledge. Reasoning tasks performed by Racer include profile matchmaking, input/output subsumption testing, and preconditions/effects analysis, which are basic mechanisms for web services discovery and invocation planning. A prototype system has been implemented.

Keywords: OWL, OWL-S, Semantic Web Service

1. INTRODUCTION

There are a lot of web services being offered nowadays, and this trend is going to continue in the future. A demand increases consequently for a system for discovery of services that are highly relevant to the requirements of a local application, and for construction of potentially suitable sequences of service invocation. We aim at developing a framework for automation and integration of web service discovery and invocation planning. Web service discovery is performed by using three methods: (a) profile matchmaking, (b) comparing input/output parameter types semantically, and (c) analyzing the preconditions and effects of web services. Two knowledge bases are used: a background knowledge base and a web service knowledge base. The first knowledge base stores background information in an application domain and defines concepts and terms used for describing web services. The standard ontology language OWL [1] is used for knowledge representation in this component. The second knowledge base stores web service descriptions in the OWL-S form. Given an application requirement, a complex sequence of web services, rather than a single service, may need to be invoked. Invocation planning is employed for construction of such a sequence, based on analysis of the interconnection between web services, e.g. one service may require outputs produced by or effects caused by another service.

Based on existing standards for description of web services (e.g. WSDL, UDDI, and OWL-S), some works have been proposed towards development of such a framework [2–4]. Some basic ideas are similar to ours, but several implementation details are different. In [2], which is based on the DECAF architectural framework [5], for instance, invocation planning using preconditions and effects of services is not mentioned. In comparison with [3, 4], which investigated the overall framework, i.e., from discovering to execution, our work focuses specifically on discovering desired services and constructing possible sequences of service invocation so as to accomplish a given set of requirements. To the best of our knowledge, there currently exists no commonly accepted framework for integration of emerging web service technologies; besides, automation or semi-automation of web services discovery and invocation planning is still an open challenge.

The paper progresses from here as follows. Section 2 provides some basic background; it describes fundamental components used in the proposed framework. Sections 3 and 4 are concerned with service discovery and invocation planning, respectively, with working examples. Section 5 presents some quantitative evaluation and discusses limitations of the implemented prototype system. Section 6 concludes the paper.

2. BACKGROUND

An ontology provides a conceptual schema about a domain [6]. Typically, it can be seen as a hierarchical data structure containing relevant entities, relationships, and rules within the domain; and, based on some formal semantics, it serves as a machine-understandable dictionary. The Web Ontology Language (OWL) is a standard language, recommended

Manuscript received on January 8, 2006; revised on March 15, 2006.

The authors are with the School of ICT, Sirindhorn International Institute of Technology, Bangkadi Campus,Thammasat University, 131 Moo 5, Tiwanont Road, Bangkadi, Muang, Pathumthani 12000, Thailand; E-mail: kati.lim@gmail.com, schaiyakul@gmail.com, avanidixit@gmail.com, ekawit@siit.tu.ac.th

by the World Wide Web Consortium (W3C), for defining web-based ontologies [1]. An OWL ontology defines concepts by specifying sufficient conditions, along with necessary and sufficient conditions, for membership of the concepts, and populates a concept by describing its instances in terms of their relations with other individual instances in the domain. OWL is used to define the background knowledge base of our framework. Several OWL editing tools are currently available—Protégé [7] is used in this work.

A web service [8] provides a function on the internet that can be invoked by software agents and can be accessed by web programmers worldwide. Each web service is usually described using Web Service Description Language (WSDL) [9]. A WSDL document defines web services as a collection of network end points or ports. It defines functions, parameters, and location of web services. A WSDL document is usually generated automatically when a web service is deployed.

OWL-S [10] is a language for describing web services in terms of OWL-based ontologies. As illustrated in Figure 1, an OWL-S document has three main parts: *Profile*, *Process*, and *Grounding*. The Profile part specifies the service class to which a web service belongs and the input/output types of the service. The Process part describes execution of a web service by specifying the flow of data and control between methods of the service. The Grounding part provides details about how a software agent can access and interact with a web service.

OWL-S API [11] developed at the University of Maryland is used for implementing a prototype system in this work. It provides a Java API for programmatic access to read, execute, and write OWL-S service descriptions. The API transforms elements of OWL-S documents into Java objects to be used by a software agent. For semantically reasoning tasks in a service discovery process, the Racer engine [12]—a Description Logic (DL) reasoning system [13]—is employed. It imports OWL documents as a knowledge base, and provides reasoning services such as consistency checking, concept subsumption testing, and semantic instance retrieval with respect to the knowledge content.

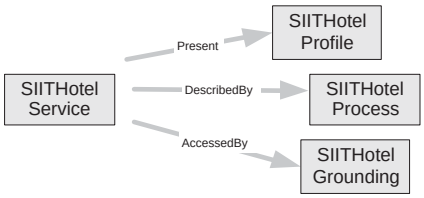


Fig.1: *SIITHotelService consisting of SIITHotelProfile (profile), SIITHotelProcess (process), and SIITHotelGrounding (grounding)*

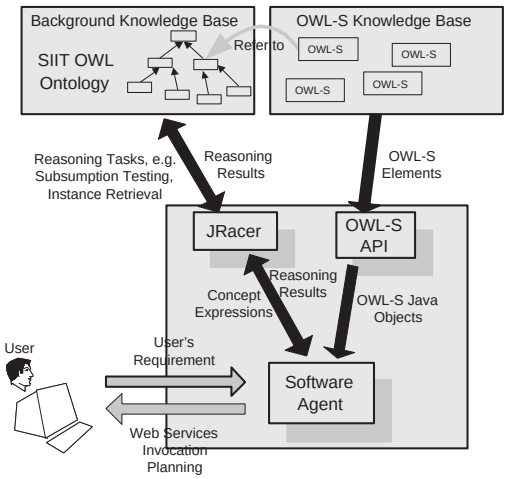


Fig.2: *The logical architecture of the framework*

3. THE PROPOSED FRAMEWORK

3.1 An Overview of the Framework

As outlined in the Figure 2, the proposed framework contains two knowledge bases: a background knowledge base, called SIIT OWL Ontology, and an OWL-S web service knowledge base. The background knowledge base provides an ontology of concepts and terms that are used by the OWL-S knowledge base for defining the profiles, the input/output parameter types, and the preconditions and effects of web services. To construct the OWL-S knowledge base, the WSDL2OWLS program provided by the OWL-S API is used for generating OWL-S descriptions from the WSDL descriptions associated with web services. On the generation process, semantic meanings are added to the OWL-S documents by replacing primitive data elements with concepts and terms defined by the background knowledge base.

A software agent retrieves metadata describing web services from the OWL-S knowledge base by using OWL-S API. The background knowledge base is loaded into the Racer reasoner. When a user requests for a service or a sequence of services that produces some desired results, the agent constructs a concept expression representing the user's requirement. This concept expression is then taken as the input of the service discovery process.

3.2 Service Discovery

Service discovery is a process of finding services that potentially satisfy a given requirement. Three methods are used for service discovery, i.e., profile matchmaking, input/output types matching, and precondition/effect analysis.

Profile Matchmaking

From a collection of available web services, matchmaking finds potential services with respect to a given

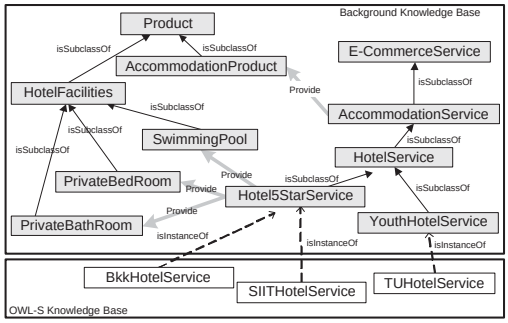


Fig.3: *Part of the background knowledge base and the OWL-S knowledge base*

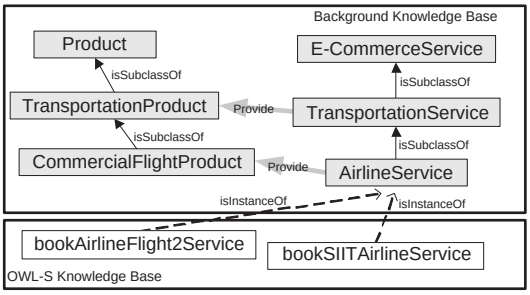


Fig.4: *Part of the background knowledge base and the OWL-S knowledge base*

requirement by semantic examination of their profiles. The input of this process is a DL-based concept expression constructed from the requirement. All concepts in the background knowledge that are subsumed by an input concept expression will be computed using a Java Racer API, called JRacer.

To illustrate, suppose that the user's requirement is "find every accommodation service that provides a swimming pool." The agent then constructs the DL-based concept expression "$AccommodationService \sqcap \exists Provide.SwimmingPool$." The background knowledge base is used for finding a subsumee of this concept, and, according to the part of the background knowledge shown in Figure 3, the concept *Hotel5StarService* is obtained. The agent can then start planning with web services that are instances of this obtained concept, i.e., *BkkHotelService* and *SIITHotelService*.

As another example, suppose that the user's requirement is "find all services that provide a commercial flight product (e.g. airline flight)." The agent then constructs the concept expression "$E\text{-}CommerceService \sqcap \exists Provide.CommercialFlightProduct$." Again the background knowledge base is used for finding a subsumee of this concept, and according to the part of the background knowledge shown in Figure 4, the concept *AirlineService* is obtained. The agent then considers web services that are instances of this obtained concept, i.e., *bookSIITAirlineService* and *bookAirlineFlight2Service*, in construction of an invocation plan. For instance, the

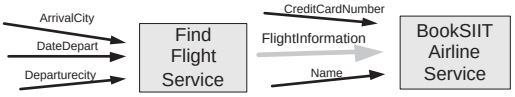


Fig.5: *An example of input/output types matching*

```
<profile:hasInput>
  <process:Input rdf:ID="flightinfo">
    <process:parameterType>
       SIITOntology.owl#FlightInformation
    </process:parameterType>
  </process:Input>
</profile:hasInput>
```

Fig.6: *Part of BookFlightService OWL-S file describing an input of BookSIITAirlineService with a parameter type FlightInformation*

agent may first start with the *bookSIITAirlineService*; then, it may try to discover other web services needed to be executed before this service through input/output types matching and precondition/effect analysis methods.

Input/Output Types Matching

The input/output parameter types of each service are represented as concept classes defined in the background knowledge base. Input/output types matching find services that produce an output required as an input of a given service. For example, consider the conceptual diagram in Figure 5, part of which is detailed by the metadata describing *BookSIITAirlineService* in Figure 6 and that describing *FindFlightService* in Figure 7. Parameter types matching reveals that the former service requires an input with the parameter type *FlightInformation*, which can be obtained from the latter service. (*FlightInformation* is a concept class defined in the background knowledge base.) In addition to exact matching, when an output type is more specific than a required input type, matching is also considered to succeed. Accordingly, the subsumption checking service provided by Racer is also used as a basic tool for this task.

Precondition/Effect Analysis

A precondition of a service specifies a condition that must be satisfied before the service can be invoked. By contrast, an effect of a service specifies a condition that necessarily holds as a result of executing the service. Given a web service, the agent retrieves its preconditions, and the background knowledge is used to determine an effect that can fulfill each obtained precondition. For example, based on the part of the background knowledge base shown in Figure 8, any web service having the effect *KnownCreditCardNumberValidity* may be chosen in order to fulfill the precondition *ValidCreditCardNumber*. As illustrated in Figures 9 and 10, we specify preconditions

```
<profile:hasOutput>
  <process:Output
    rdf:ID="findAirlineFlightResult">
    <process:parameterType>
       SIITOntology.owl#FlightInformation
    </process:parameterType>
  </process:Output>
</profile:hasOutput>
```

Fig.7: *Part of FindFlightService OWL-S file describing the output of FindFlightService with a parameter type FlightInformation*

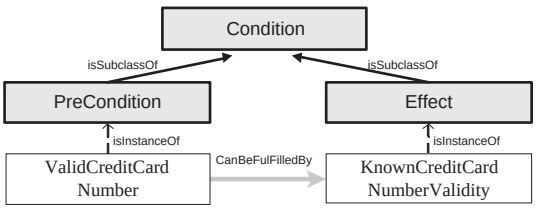


Fig.8: *CanBeFulfilledBy relation between the Effect and PreCondition concept classes in the background knowledge base*

and effects of a web service using the *textDescription* section in the profile part of an OWL-S document.

As another example, referring to the part of background knowledge shown in Figure 11, any web service having the effect *SIITHotelServiceAccountCreated* may be chosen in order to fulfill the precondition *SIITHotelServiceAccountExists*.

4. PLANNING

With the three discovery methods discussed in the preceding section, service planning can be achieved. The basic purpose of planning is to create a sequence of service invocation to achieve a given requirement. There can be numerous possible service invocation plans inasmuch as a service may require multiple inputs and/or multiple preconditions, each of which may be fulfilled by possibly several other services. A *service tree* is used to represent a potential invocation plan. Postorder traversal of a service tree determines a sequence of services to be successively invoked during a service invocation process.

In order to create a service tree, we use a temporary bipartite tree for storing the relationships between services and their requirements (i.e., input parameter types and/or preconditions). Such a bipartite tree contains nodes of two types: S-nodes ("Service"-nodes) and R-nodes ("Requirement"-nodes). An S-node represents a web service in the form of Java OWL-S object, while an R-node represents a requirement. Each R-node child of an S-node specifies an input or precondition required by the S-node. Each S-node child of an R-node specifies a service that can fulfill the requirement represented by the R-node. For construction of a service tree,

```
<profile:textDescription>
  (PreCondition = SIITOntology.owl
                  #ValidCreditCardNumber)
</profile:textDescription>
```

Fig.9: *Part of the Profile section of an OWL-S document, specifying the precondition of a web service*

```
<profile:textDescription>
  (Effect = SIITOntology.owl
            #KnownCreditCardNumberValidty)
</profile:textDescription>
```

Fig.10: *Part of the Profile section of an OWL-S document, specifying the effect of a web service*

an S-node is treated as an AND-node, while an R-node is treated as an OR-node. That is, an S-node requires all of its R-node children to be satisfied; by contrast, an R-node can be satisfied by invoking any of its S-node children.

Figure 12 demonstrates a process of constructing service trees. From the temporary bipartite tree T0 in the figure, the four service trees T4–T7 are obtained. Each of these trees represents a possible composition of services to satisfy a given requirement. Each tree will be traversed in postorder (left-right-root), considering that the input types and/or preconditions at the root can be satisfied by its left and right nodes. For example, from T6 the sequence [S3, S4, S1] is obtained.

Notice that in addition to the sequences obtained by postorder traversal mentioned above, an invocation sequence can also be derived from a subtree of an obtained service tree if some requirement can be given manually by a user. Referring to Figure 12, for example, the sequence [S1, S3] can also be obtained from T6 on the condition that a user must be able to provide information for satisfying the requirement R3 in place of the service S4.

An Example Experimental Scenario

To illustrate, suppose that the user's requirement is "find all services that provide a transportation product." *AirlineService* is discovered using profile matchmaking, as discussed in Section 3.2. Figure 13 demonstrates the relationship of all services and requirements involved in the scenario. One instance of *AirlineService* is *bookSIITAirlineService*, which is represented as S1 in Figure 12. It has one precondition, i.e., *ValidCreditCardNumber*, and requires three inputs, i.e., *CreditCardNumber*, *FlightInformation*, and *PersonName*. As the agent processes the precondition *validCreditCardNumber* by precondition/effect analysis (referring to Section 3.2 and Figure 8), it discovers that the condition can be fulfilled by *knownCreditCardNumberValidity*, which is an effect of two services, *validateCreditcardNumberService*

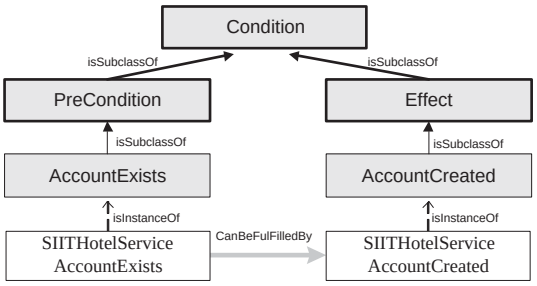


Fig.11: *CanBeFulfilledBy relation between the Effect and PreCondition concept classes in the background knowledge base*

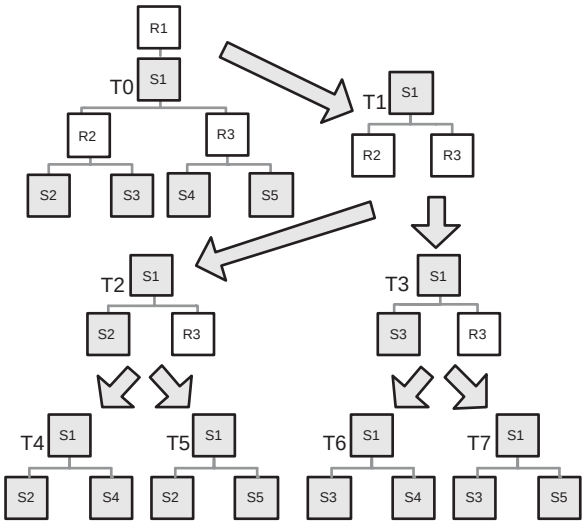


Fig.12: *Constructing service trees*

and *validateCreditcardNumber2Service*. These services are represented as S2 and S3, respectively, in Figure 12. The agent then considers *validateCreditcardNumberService* (S2), which has two inputs, *CreditcardNumber* and *CreditcardType*. These inputs are assumed to be provided directly by the user. By input/output types matching, the agent discovers that *findFlightKatiService* and *findAirlineFlightService* has *FlightInformation* as their output. They are represented by S4 and S5 in Figure 12, respectively. Next, the agent considers *findFlightKatiService* (S4), which has *City* (from), *City* (to) and *Datetime* as its inputs, which can be provided by the user. Assuming that the user can also provide other inputs of *bookSIITAirlineService*, the first service tree, T1, in Figure 14 is derived. The agent then considers *validateCreditcardNumber2Service* (S3) as an alternative of *validateCreditcardNumberService* (S2) and, likewise, *findAirlineFlightService* (S5) as an alternative of *findFlightKatiService* (S4). Figure 15 illustrates the scenario of the service tree T1. The other three service trees (T2, T3 and T4) in Figure 14 can be derived in a similar way; all of them are ready for invocation.

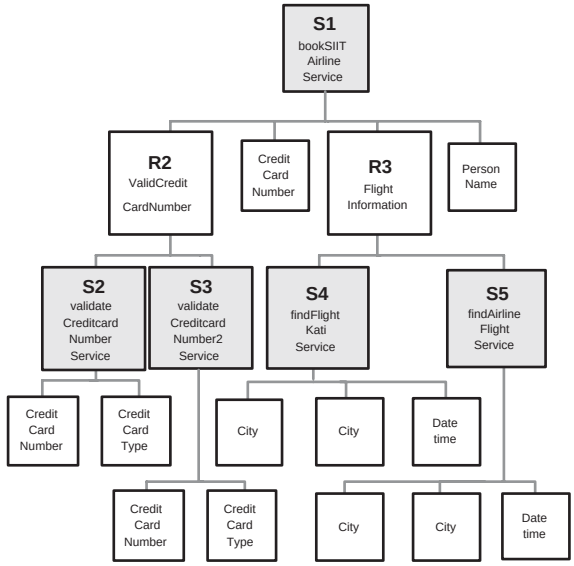


Fig.13: *Services and requirements*

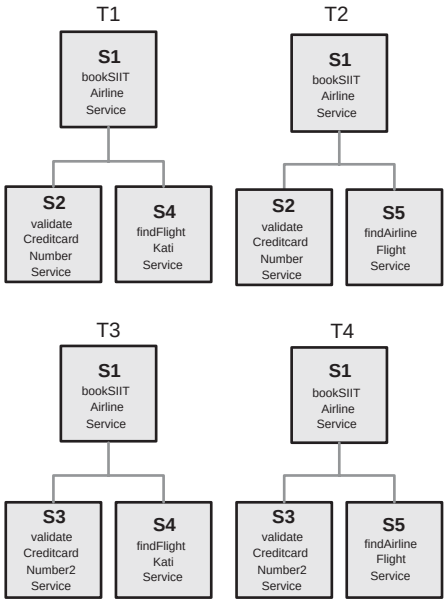


Fig.14: *Service trees*

5. QUANTITATIVE EVALUATION AND LIMITATIONS

In the experiment described previously, all services are assumed to already exist in the OWL-S knowledge base. In this section, services are added into the OWL-S knowledge base incrementally one at a time. Each time a new service is added, an experimental scenario is run in order to see service trees possibly obtained. Assuming the same user requirement as in the previous section, i.e., "find all services that provide a transportation product," the number of resulting service trees and execution time are shown in Table 1.

Initially, there exists no service in the OWL-S knowledge base. In Running #1, *bookSIITAirlineSer-*

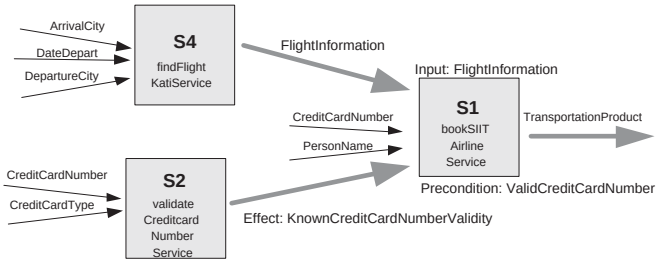


Fig.15: *T1 Service tree*

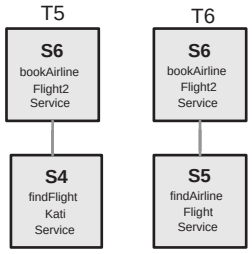


Fig.16: *Two additional service trees obtained after adding bookAirlineFlight2Service*

Table 1: *Experimental Scenario Results Summary*

#	Service added to the OWL-S knowledge base	No. of service trees	Run time (msec.)
1	*bookSIITAirlineService*	1	656
2	*findAirlineFlightService*	1	782
3	*validateCreditcardNumberService*	1	891
4	*findFlightKatiService*	2	1000
5	*validateCreditcardNumber2Service*	4	1109
6	*ITBookRoomService*	4	1281
7	*BookEddieHotel*	4	1344
8	*bookAirlineFlight2Service*	6	1625

vice is added. As a result, there is only one service tree, which contains only one service, *bookSIITAirlineService*. As explained in the previous section, *findAirlineFlightService* and *validateCreditcardNumberService* could produce an output and an effect, respectively, needed by *bookSIITAirlineService*. These two services are thus added in Runnings #2 and #3, respectively. The number of service trees obtained from Runnings #2 and #3 is still one, but the obtained trees consist of two services and three services, respectively.

In Running #4, *findFlightKatiService*, which can be used instead of *findAirlineFlightService*, is added. The addition results in two service trees: one uses *findFlightKatiService* whereas the other uses *findAirlineFlightService* in their invocation paths. In Running #5, *validateCreditcardNumber2Service* is added. This service can be a replacement for *validateCreditcardNumberService*, and the addition yields the four service trees in Figure 14.

In Runnings #6 and #7, *ITBookRoomService* and *BookEddieHotel*, which are irrelevant services, are added. These two services do not produce any outputs or effects needed by any service in a service tree previously obtained. The result of Runnings #6 and #7 are thus the same as that of Running #5.

In Running #8, *bookAirlineFlight2Service*, which can serve as a replacement for *bookSIITAirlineService*, is added. This service requires three inputs: *CreditCardNumber*, *FlightInformation*, and *PersonName*. Each of *findFlightKatiService* and *findAirlineFlightService* can produce *FlightInformation*. By addition of *bookAirlineFlight2Service*, the service trees shown in Figure 16 are additionally obtained.

The number of possible service trees increases as we add more relevant services to the OWL-S knowledge base. As shown in the last column of Table 1, addition of services also results in larger amount of execution time required for loading service's OWL-S files into the OWL-S knowledge base. Moreover, although not shown in the table, the size of the background knowledge base also affects the execution time. For example, when 100 new concepts are added to the background knowledge base used in Running #8, which initially contains 116 concepts, the execution time changes from 1625 milliseconds to 1953 milliseconds.

Some limitations of our prototype system are as follows. First of all, there is no mechanism for advertisement of services; our agents simply store all URIs of the OWL-S files within themselves. Secondly, the background knowledge base used in the current prototype system provides an ontology for a rather small domain. Using the proposed framework in a real world application would require more work on designing and constructing more complex and advanced ontologies, covering a far wider variety of domains. When there is a new web service with an input type that is not defined in the background knowledge base, semantic-based types matching may not succeed.

6. DISCUSSION AND CONCLUSION

With the growing number of web services being put up on the internet nowadays, tools for facilitation of automatic discovery and invocation of web services are obviously necessary. We propose a framework for integrating emerging technologies related to this process, based on currently existing standards. WSDL describes web services only in terms of primitive data types and therefore provides a limited means for service discovery. In order to realize the framework, semantic meanings are added to web services when OWL-S files are created. The framework employs two knowledge bases, i.e., a background knowledge base (a domain ontology) and a OWL-S web service knowledge base. The agent performs service discovery by extracting and/or deriving required information from these knowledge bases, and constructs plans for service invocation so as to meet given requirements. A prototype service discovery and invocation planning

system has been implemented. Further works include integration of an automatic service invocation module, and support for more complicated logic-based precondition/effect analysis.

ACKNOWLEDGMENTS

This work was partially supported by National Electronics and Computer Technology Center, under Grant No. NT-B-22-I4-38-49-11.

References

[1] G. Antoniou and F. van Harmelen, "Web Ontology Language: OWL," *Handbook on Ontologies*, Springer, 2004, pp. 67-92.

[2] A. Shaban and V. Haarslev, "Applying Semantic Web Technologies to Matchmaking and Web Service Descriptions," *Proceedings of the Montreal Conference on eTechnologies (MCeTech'05)*, Jan. 2005, Montreal, Canada, pp. 97-104.

[3] S. Balzer, T. Liebig, and M. Wagner, "Pitfalls of OWL-S—A Practical Semantic Web Use Case," *Proceedings of the 2nd International Conference on Service-Oriented Computing (ICSOC'04)*, Nov. 2004, New York, pp. 289-298.

[4] D. Martin, et. al., "Bringing Semantic to Web Services: The OWL-S Approach," *Proceedings of the 1st International Workshop on Semantic Web Services and Web Process Composition (SWSWPC'04)*, July 2004, San Diego, California.

[5] J. Graham, K. Decker, and M. Mersic, "DECAF–A Flexible Multi Agent System Architecture," *Autonomous Agents and Multi-Agent Systems*, Vol. 7, N0. 1-2, pp. 7-27, 2003.

[6] M. Uschold and M. Gruninger, "Ontologies: Principles, Methods and Applications," *Knowledge Engineering Review*, Vol. 11, No. 2, pp. 93-136, 1996.

[7] The Protégé Ontology Editor and Knowledge Acquisition System, http://protege.stanford.edu.

[8] Web Services Activity, World Wide Web Consortium, 2002, http://www.w3.org/2002/ws.

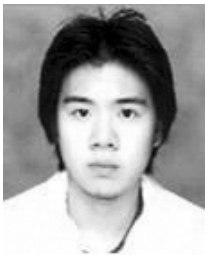
[9] E. Christensen, et. al., "Web Services Description Language (WSDL) Version 1.1," Technical report, World Wide Web Consortium, 2001, http://www.w3c.org/TR/wsdl.

[10] D. Martin, et. al., "OWL-S: Semantic Markup for Web Services," Technical report, The OWL Services Coalition, 2003, http://www.daml.org/services/owl-s/1.1/overview.

[11] The OWL-S Java API, Mindswap: Maryland Information and Network Dynamics Lab Semantic Web Agents Project, 2004, http://www.mindswap.org/2004/owl-s/api.

[12] V. Haarslev and R. Möller, "RACER System Description," *Lecture Notes in Artificial Intelligence*, Vol. 2083, pp. 701-705, 2001.

[13] F. Baader, et. al., *Description Logic Handbook: Theory, Implementation and Applications*, Cambridge University Press, 2002.



Kati Limapichat received the BS degree in Information Technology from Sirindhorn International Institute of Technology, Thammasat University, Thailand in 2006. He is currently a student in Master Program in Computer Science department at University of Southern California, United States. His areas of interest include semantic web and artificial intelligence.

Sukasom Chaiyakul received the BS degree in Information Technology from Sirindhorn International Institute of Technology, Thammasat University, Thailand in 2006. He is currently a student in European Master in Informatics majoring in Net Centric, a cooperation program between the Rheinisch-Westfälische Technische Hochschule (RWTH) Aachen, Germany and the University of Edinburgh, United Kingdom. His areas of interest include semantic web and web services.

Avani Dixit received the BS degree in Information Technology majoring in Information System from Sirindhorn International Institute of Technology, Thammasat University, Thailand in 2006. He is currently enrolled in Erasmus Mundus MSc in Network and e-Business Centred Computing, a program coordinated between Reading University, United Kingdom, Aristotle University of Thessaloniki, Greece, and University of Carlos III Madrid, Spain. He is expected to graduate in 2008. His interests are in the fields of web services and grid computing.

Ekawit Nantajeewarawat received the BEng degree in Computer Engineering from Chulalongkorn University, Thailand, in 1987; and the MEng and DEng degrees in Computer Science from the Asian Institute of Technology, Thailand, in 1991 and 1997, respectively. He is currently an associate professor of Computer Science at Sirindhorn International Institute of Technology, Thammasat University, Thailand. His research interests include knowledge representation, automated reasoning, rule-based equivalent transformation, program synthesis, formal ontologies, semantic web, and object-oriented modelling.