Using Semantic Web Services for Context-Aware Mobile Applications

Mithun Sheshagiri, Norman M. Sadeh, and Fabien Gandon
Mobile Commerce Laboratory
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891
sadeh@cs.cmu.edu

Abstract
One way of overcoming the challenges associated with mobile and pervasive computing environments involves providing users with higher levels of automation. This in turn requires capturing the context within which the user operates. In this paper, we describe ongoing research aimed leveraging Semantic Web Services in support of context awareness. This includes modeling sources of contextual information as web services that can be automatically discovered and accessed by agents that assist the user with different sets of tasks. By automatically discovering and accessing a variety of external web services, these agents can automatically develop and execute simple plans to assist the user (e.g. ordering a pizza, organizing an evening out, etc.). This research is being conducted in the context of *my*Campus, a prototype semantic web environment to enhance everyday campus life at Carnegie Mellon University.

Keywords
Intelligent Agents, Semantic Web, Context Awareness, Privacy, Web Services

1.0 Introduction

With hundreds of Internet-enabled mobile devices, the mobile Internet is opening the door to a slew of new mobile applications and services that will assist users as they engage in time-critical, goal-driven tasks [10]. Yet today, the mobile commerce landscape is dominated by relatively simple infotainment services. Moving beyond these simple services requires overcoming the inherent input/output limitations of mobile devices through higher degrees of automation and the development of services that understand the *context* within which their users operate – e.g. their locations, the activities they are engaged in, who their friends and colleagues are as well as a number of other contextual attributes and preferences. In this paper we look at ways of using contextual information using web services and automatically chaining together multiple services to achieve complex tasks using planning. Our usage of web services is discussed in the context of *my*Campus-a context-aware environment aimed at enhancing everyday campus life.

2.0 Overview of MyCampus

In *my*Campus, users can acquire (or subscribe) to different sets of task-specific agents that help them with different tasks. To properly operate these agents require knowledge of one or more contextual attributes about their users as well as possibly other users. These attributes can potentially be acquired from a number of possible resources, which typically vary from one user to another (e.g. not everyone uses Microsoft Outlook as their calendar) and may even vary over time for the same user. To overcome this challenge, sources of contextual information in *my*Campus are modeled as Semantic Web Services that can automatically be discovered and accessed by agents. Access to a user's contextual resource is controlled according to user-specified privacy preferences (including context-sensitive preferences such as colleagues have access to my location only if they have a meeting with me in the next hour).

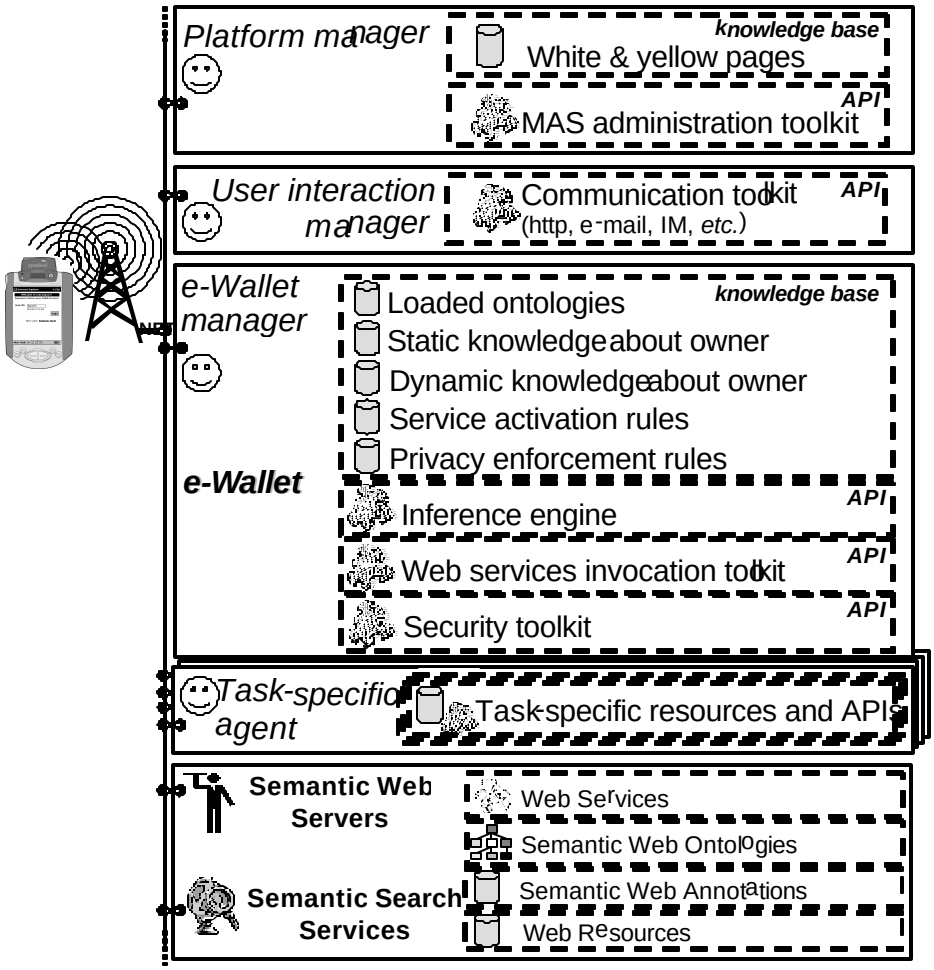

Figure 1. *my*Campus architecture: a user's perspective - the smiley faces represent agents

The e-Wallet Manager (or simply e-Wallet) serves as a repository of static knowledge about the user – just like.NET Passport, except that here knowledge is represented using OWL [14]. In addition, the e-Wallet contains knowledge about how to access more information about the user by invoking a variety of resources, each represented as a Web Service. This knowledge is stored in the form of rules that map different contextual attributes onto one or more possible service invocations, enabling the e-Wallet to automatically identify and activate the most relevant resources in response to queries about the user's context (e.g. accessing the user's calendar to find out about her availability, or consulting one or more location tracking applications in an attempt to find out about her current location). User-specified privacy rules, also stored in the e-Wallet, ensure that information about the user is only disclosed to authorized parties, taking into account the context of the query. They further adjust the accuracy of the information provided in accordance with the user's obfuscation preferences rules. (For example, I am willing to disclose my location in the building to Bob but I only want to reveal the city information to Mary).

Figure 1 provides an overview of *my*Campus. It illustrates a situation where access is from a PDA over a wireless network. However, our architecture extends to fixed Internet scenarios and more generally to environments where users can connect to the infrastructure through a number of access channels and devices – information about the particular access device and channel can actually be treated as part of the user's context and be made available through her e-Wallet.

Clearly, agents are not limited to accessing information about users in the environment. Instead, they also typically access public Web

Services, Semantic Web annotations, public ontologies and other public resources. On CMU's campus, where we have deployed *my*Campus, this includes access to a variety of services such as 23 restaurant web services or a pubic weather forecasting web service.
In the following sections, we focus on the use of web services in our system. Additional details on *my*Campus and some of the agents we have deployed can be found in [4, 5, 9].

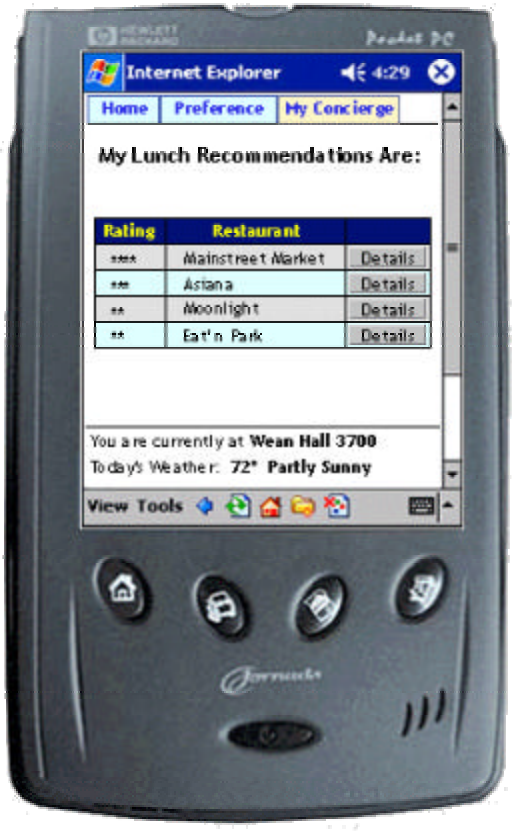

Figure 2: This Restaurant Concierge is an example of a *my*Campus agent.

3.0 Using Web Services for Contextual Information

This section discusses in detail, how web services are used as sources of contextual information. As explained in an earlier section, contextual attributes can potentially be acquired from a number of possible resources, which typically vary from one user to another and may even vary over time for the same user. In *my*Campus, sources of contextual information are wrapped as Semantic Web Services [1]. This means that each source of contextual information is described by a profile that describes its functional properties in relation to one or more ontologies [13]. For instance, Microsoft Outlook Calendar is an instance of a resource that provides both calendar activity information and user location information. Service descriptions also include details about how to invoke a service (e.g. input, output and preconditions). Thanks to these profiles, relevant sources of contextual information can be automatically discovered and accessed.

3.1 Service Invocation Rules

One particularly efficient way of identifying a service that can provide information about a given contextual attribute (e.g. user's location) is by using a set of rules. Service invocation rules provide a mapping between contextual attributes and personal resources available to access these attributes, viewing each personal resource as a Semantic Web Service. For instance, the location of the user is provided from a service wrapped around the user's Microsoft Outlook calendar. Service invocation rules are not limited to providing a one-to-one mapping between contextual attributes and personal resources. Instead, they can leverage rich ontologies of personal resources, enabling the e-Wallet to select among a number of possible personal resources based on availability, accuracy and other relevant considerations. For instance, in response to a query about the user's location, the rules can specify that, when the user is driving, the best method available is the GPS in her car. If she is at work and her wireless-enabled PDA is on, her location can be obtained using location tracking functionality running over the enterprise's wireless LAN. If everything else fails, her calendar might have some information about her location.

3.2 Semantic Web Services Using OWL-S

Binding contextual attributes to services through rules is a very efficient mechanism, especially for time critical applications. However, this is not always practical, especially considering that, over time, users may acquire new task-specific agents and new sources of contextual information. Maintaining detailed service invocation rules that cover all possible situations is simply impractical. Instead, a more flexible (albeit more computationally demanding) approach involves relying on automated service discovery. In general, contextual information about a given user is obtained by sending a query to her eWallet. As detailed in the following subsections, the eWallet then relies on a combination of local service identification rules and local and global service discovery

mechanisms to identify one or more relevant sources of contextual information. In the process, it also ensures that the request is compatible with the user's privacy rules, as detailed in [3]. Service discovery is done as follows. Services are registered in directories along with profiles that describe their various relevant capabilities and characteristics. Unfortunately, current Web Services standards for discovery such as UDDI [12] are not sufficient. UDDI does not describe a service in terms of the functionalities it offers but provides information about the entity that owns it and provides mechanisms to classify the service in terms of standard taxonomies such as North American Industry Classification System (NAICS) [11]. Moreover it supports only syntactic matching. A directory that advertises the functional attributes of the service is a far better alternative. Moreover if these descriptions are represented in Semantic Web Languages then subsumption-based reasoning can be used for semantic matching of service functionalities. The OWL ontology for services (OWL-S) [1] framework can be used to build directories that contain service descriptions (functional attributes expressed in Semantic Web Languages). More information on semantic discovery of services can be found in [6]. Information in the eWallet is stored as semantic annotations, therefore; the eWallet can readily make use of OWL-S (see appendix) based semantic service discovery. To further clarify the need for semantic discovery let us use the example of a service that provides a list of Italian restaurants using zip code as input. The user of the eWallet could be looking for an *food:eatery* using a popular ontology called *food*. Let us assume that OWL-S is used to represent the service as an atomic service with *loc:zipcode* as the input and *food:ItalianRestaurant* as the output. Also, assume that the *food* ontology categorizes *ItalianRestaurant* as a *subclassOf* of *eatery*. By using semantic inferencing one can conclude that all instances of *ItalianRestaurant* are instances of *eatery* and therefore the service in question is returned as a match to the query. This kind of matching is not possible in UDDI because firstly, inputs and outputs are not specified and secondly, the UDDI framework is not capable of semantic reasoning which we used to make match in the above example.

4.0 Composition of Services

Composition is defined as the task of putting together atomic/basic services to perform complex tasks. To start the discussion on composition, let us first consider an example. When a user wants to find about the local weather forecast, the system needs to first find the current location of the user before it invokes the weather service. The location of the user is in turn obtained by the eWallet of the user which actually invokes a web service to obtain this information. This in itself is a primitive form of composition where the weather service requires the invocation of the location service. This composition was possible using input and output **"type"** matching. One could think of a sequence of two invocations in which the second service can be invoked only when the invocation of the first operator produce a particular **"value"**. Such cases cannot be handled by type matching. To capture this ordering constraint we make use of preconditions. Preconditions are interpreted as: when a precondition is associated with a service, the service can be invoked only when the precondition is satisfied. A precondition is satisfied based on the value of output with which it is associated. We use both, type and value matching for composition.

Services are described using inputs, outputs and preconditions (IOPs). We model services in our system using the OWL-S *process* ontology. More specifically, all services are described using OWL-S *atomic processes*. We also propose the addition of a new construct - *realizedBy* which connects preconditions to outputs. Whether the output actually realizes the precondition can be found out only during service invocation. We use this construct to first compose and invoke the services involved later. This technique of disconnecting composition and invocation enables us to find out whether the existing set of services can satisfy the user' goal before we invoke a single service. If one were to implement simultaneous composition and invocation, the search for operators would have to be in the forward direction i.e., from the initial state to the goal. The main problem with *forward-chaining* arises due to high branching factor which leads to a huge search space. This also leads to other complications. One has to invoke services to move towards the goal. In the end, if the goal cannot be achieved, the service invocations achieved nothing but the waste of computation and network bandwidth. The situation is potentially worse if some if some of these services are *non-idempotent*. Thus our approach of first using *backward-chaining* to build the plan and then invoking the services is efficient and avoids the complications caused by *non-idempotent* services.

4.1 Automatic Operator Extraction and Planning

The composition of atomic services into complex services can be viewed as planning. This is done by mapping atomic services into planning operators and running a planning algorithm to link these operators. The plan generated using these operators constitutes a complex service. As mentioned before, *my*Campus environment has various specific agents. Some of these perform simple tasks like providing weather information to the user whereas others can do more complex tasks like ordering food. Some of these complex agents make use of the planner. The user specifies the goal using an interface designed for each task-specific agent. On receiving the goal, the task-specific agent checks whether the information resides in the eWallet of the user or if it can be provided by services known to the agent. If both these checks fail, the discovery phase is initiated. If an appropriate service(s) is found, the task of converting atomic services into planning operators is done as follows. A service description consists of one or more atomic processes. The service descriptions are first transformed into Predicate-Subject-Object (PSO). For each atomic service, we build *STRIPS*-style [3] planning operators by querying the triples. The actual composition is done by a simple *backward-chaining* planning algorithm.

The composition problem can be transformed into a planning problem given by the tuple <***Op, G***> where ***Op*** is the set of operators obtained from service descriptions and ***G*** is the user's goal(s). Composition is done using the following algorithm:

```
Compose (Goals G, Operators Op)
   0. If G is empty
        a. Return Success
   1. Search Op for operators
       that satisfy all the goals G.
   2. IF no operator is found
        a. Return Fail
   3. ELSE
        a. Delete G
        b. Add unavailable inputs of
            operator to G'
        c. Add unavailable outputs (using
            realizedBy)
        d. Compose(G', Op)
```

The first call to the Compose routine consists of all available operators and the user's goal. The algorithm looks for services that satisfy the goal. If no such service is found, composition fails and discovery can be re-initiated to look for more operators. If on the other hand, all goals in G are satisfied by operator(s), then the system checks to find if inputs to these operators are available. All unavailable inputs are transformed into goals for the next iteration. A similar availability check is run on outputs associated to preconditions via the *realizedBy* construct. All unavailable outputs are added as goals for the next iteration. A recursive call is made to the compose algorithm with the new set of goals. Composition is successful if all goals are satisfied. Composition fails if the system is unable to find services that achieve some goal. Use of planning for composition has been discussed in detail in [2, 7, 8].

It is possible that the composer finds multiple operators/services that achieve the same goals/sub-goals. These are taken into account as contingencies. If the first choice fails to produce the intended result, the alternative services are tried. The first choice is determined using the user's preference (for example, the user prefers services that accept debit cards) or her context (for example, the user would prefer the fastest pizza delivery service when she is at work).

To further clarify our ideas, we illustrate the use of composition using a scenarios described below.

4.2 Scenario

The scenario involves finding a pizza delivery service and ordering a pizza. To explain the discovery process let us assume that the semantic directory uses an ontology based on the NAICS. We query the main directory using *Pizza_Delivery_Shops*. *Pizza_Delivery_Shops* is a *subClassOf Limited_Service_Restaurants* in the NAICS based ontology. The directory returns a large number of ranked matches. Services classified as *Pizza_Delivery_Shops* get a higher ranking compared to services classified as *Limited_Service_Restaurants*. Options are further narrowed down using the user's context (in this case the location) ascertained from the eWallet. Assume that agent finds a single pizza delivery shop near its current location.

The main service description returned by the directory look-up, points to the *Process* that contains the atomic processes. We synthesize planning operators from these descriptions and try to build a plan that will enable the user to order a pizza and get it delivered to his current

```
;Pizza Delivery Service
; s: service name
; i: input
; o: ouput
; p: precondition

(s LocationCheck
(i pin)
(o RangeCheck))

(s CreditCardInfo
(i CreditCardNumber)
(i CreditCardType)
(i CreditCardExpDate)
(o CreditCardStatus))

(s PizzaBuy
(p CCAuth)
(p DeliveryAddReq)
(i PizzaType)
(o PizzaBought))

(s DeliveryAdd
(p InRange)
(i Add1)
(i Add2)
(o AddReq))

(realizedBy InRange RangeCheck)
(realizedBy CCAuth CreditCardStatus)
(realizedBy DeliveryAddReq AddReq)
```

Figure 3 Pizza Delivery Operators

location. Figure 3 shows the operators for the pizza delivery service.

The LocationCheck service ensures that the pizza service delivers to the use's location. CreditCardInfo service validates the credit card. DeliveryAdd service obtains the user's street address. Finally, the PizzaBuy service lets the user buy the pizza.

The user request for getting the pizza delivered is transformed into a goal- PizzaBought. The PizzaBuy service achieves this goal and is therefore included in the plan. The PizzaBuy service has two preconditions and an input. The preconditions are associated to the corresponding ouputs using the *realizedBy* predicate. These outputs along with the single input- PizzaType are made the goals for the next iteration. Other operators are included in the plan in a similar fashion. In the end, all goals are checked against the user's eWallet. Missing information like PizzaType is captured by prompting the user.

Figure 4. shows the plan generated by the planner. Once all the information is available, the plan can be invoked by accessing the WSDL descriptions via the OWL-S grounding.

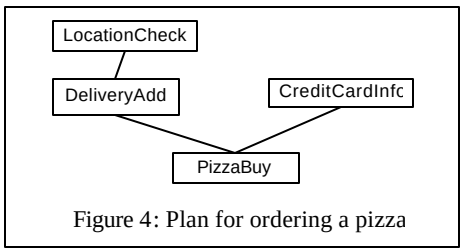

Figure 4: Plan for ordering a pizza

5.0 Conclusion

In this paper, we provided an overview of *my*Campus, a semantic web environment aimed at enhancing everyday campus life. Within *my*Campus, users over time acquire or subscribe to a variety of task-specific agents that assist them in the context of different tasks (e.g. scheduling meetings, sharing documents, organizing evenings out, filtering and routing incoming messages, etc.). Many of these agents require knowledge of the context within which their user operates as well as possibly information about the context of other users. In *my*Campus, sources of contextual information (e.g. calendar, location tracking functionality, organizational information, etc.) are represented as Semantic Web Services. These are described by profiles that can refer to any number of relevant ontologies. Service descriptions also include information about how to invoke a service. The end result is an environment, where relevant sources of contextual information about a user can automatically be discovered and accessed in support of different queries. This approach makes it possible to accommodate users that rely on different sets of contextual resources (e.g. different calendar systems, different sources of location information, etc.) and to adapt to situations where sources of contextual information for a user may change over time (e.g. different location tracking services depending on where the user is). Queries about a given user's context are submitted to that user's e-Wallet, which acts as a gatekeeper and a clearinghouse for the user's personal information, enforcing user-specified privacy rules. As users subscribe to or acquire new task-specific agents, these agents can find relevant contextual information about users by querying their e-Wallets.

*my*Campus agents can range from simple agents that rely on one or more sources of contextual information about their users to more complex agents that are capable of dynamically building plans in response to requests from their users. In this paper, we detailed how Semantic Web Services in

*my*Campus can be used to support: (1) The dynamic discovery and access of contextual sources of information about a user via her e-Wallet (2) The automated generation of plans by task-specific agents through the discovery of services modeled as planning operators that can be dynamically composed to satisfy one or more user-goals

References

[1] DAML Services Coalition: *DAML-S: Web Service Description for the Semantic Web*, First International Semantic Web Conference, ISWC'02, Sardinia, Italy, LNCS 2342, (2002) 348-363

[2] Dan Wu, Evren Sirin, James Hendler, Dana Nau & Bijan Parsia, *Automatic Web Service Composition Using SHOP2*, Workshop on Planning for Web Services, Trento, 2003

[3] Fikes, R. E. and Nilsson, N. J., STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4): pages 189-208, 1971.

[4] Gandon, F. and Sadeh, N., *Semantic Web Technology to Reconcile Privacy and Context Awareness*, Web Semantics Journal. Vol. 1, No. 3, 2004.

[5] Gandon, F. and Sadeh, N., "A Semantic eWallet to Reconcile Privacy and Context Awareness", Second International Semantic Web Conference, Florida, October 2003.

[6] Lei Li and Ian Horrocks. A software framework for matchmaking based on semantic web technology. In *Proc. of the Twelfth International World Wide Web Conference (WWW 2003)*, pages 331-339. ACM, 2003.

[7] Mithun Sheshagiri, Marie desJardins, and Timothy Finin, *A Planner for Composing Services described in DAML-S*, Workshop on Planning for Web Services, Trento, 2003

[8] Mithun Sheshagiri, *Automatic Service Composition and Invocation for Semantic Web Services*, University of Maryland, Baltimore County, May 2004.

[9] Norman M. Sadeh, Ting-Chak Chan, Linh Van, OhByung Kwon and Kazuaki Takizawa. "Creating an Open Agent Environment for Context-aware M-Commerce", in "Agentcities: Challenges in Open Agent Environments", Ed. by Burg, Dale, Finin, Nakashima, Padgham, Sierra, and Willmott, LNAI, Springer Verlag, pp.152-158, 2003

[10] Norman M. Sadeh, m-Commerce: Technologies, Services and Business Models, Wiley, 2002

[11] North American Industry Classification System , `http://www.census.gov/epcd/www/naics.html`

[12] OASIS: Universal Description, Discovery and Integration standard, `http://www.uddi.org`

[13] T. R. Gruber. A translation approach to portable ontologies. *Knowledge Acquisition*, 5(2):199-220, 1993

[14] W3C: OWL Web Ontology Language Reference, Working Draft 31 March 2003, http://www.w3.org/TR/owl-ref/

Acknowledgements

This material is based on research conducted at Carnegie Mellon University's Mobile Commerce Lab. (School of Computer Science) as part of the DAML initiative. This work has been sponsored by the Air Force Research Laboratory under contract F30602-02-2-0035 and by the Defense Advanced Research Project Agency under contract F30602-98-2-0135. The US Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. This work was also in part supported by grants from HP, Symbol, Boeing, Fujitsu and the IST Program (SWAP project). The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory or the U.S. Government.

Appendix

OWL-S: A Brief Overview

OWL-S is an ontology in OWL for describing services. The aim of OWL-S is to help service providers describe services to enable automatic discovery, composition and execution monitoring. OWL-S consists of the following ontologies:

1. The topmost level consists of a *Service* ontology. The Service is described in terms of a *ServiceProfile*, *ServiceModel* and a *ServiceGrounding* ontology.
2. The service presents a *ServiceProfile* which has a subclass *Profile*. The *Profile* provides a vocabulary to characterize properties of the service provider, functional properties of the service like Inputs, Outputs, Effects and Preconditions (IOPEs) and non-functional properties of the service. The Profile is used for discovering the service. The service provider provides this description to the directory service.
3. The service is *describedBy* a *ServiceModel* which has a subclass Process. The Process consists of all the functional properties of the service; the Profile on the other hand need not include all the functional properties. The service could be a collection of atomic services, composite services or a combination of both. The Process lets the service provider describe services in terms of IOPEs and is used for composition.
4. The service supports a *ServiceGrounding* which has a subclass Grounding. The Grounding provides an interface to plug-in WSDL descriptions. Grounding indicates how each atomic service can be invoked using a WSDL operation. As per OWL-S 1.0 specs, the service descriptions are instances of the Profile and Process ontologies.