Journal of Web Engineering, Vol. 8, No.2 (2009) 182-210

A SEMANTIC WEB SERVICES-BASED INFRASTRUCTURE FOR UBIQUITOUS SERVICE SYSTEMS

YOUNGGUK HA

Konkuk University, Seoul, Korea
ygha@konkuk.ac.kr

CHEONSHU PARK and SANGSEUNG KANG

ETRI, Daejeon, Korea
bettle@etri.re.kr, sskang@etri.re.kr

Received August 13, 2008
Revised January 27, 2009

The recent emergence of ubiquitous computing is rapidly changing computing environments and technologies. Based on the ubiquitous computing technologies, users can be provided with the services they need, anytime and anywhere through not only common computing devices but ubiquitous computing devices such as wireless sensor networks and embedded computers in their daily environments. There are requirements to be met for implementing such ubiquitous service systems. One of the essential requirements is that service applications must provide services dynamically based on the awareness of the current service environments, rather than statically for pre-programmed service environments. That is, service applications need to be aware of feasible service devices and sensors based on the user's current location, and then interoperable with them automatically. In this paper, we present design and implementation of a service infrastructure for ubiquitous services by the use of Semantic Web Services technology.

Key words: Semantic Web Services, service ontology, semantic service discovery, service composition, location-aware service, ubiquitous service
Communicated by: D. Schwabe & A. Ginige

1 Introduction

In recent years, motivated by the emergence of ubiquitous computing [42] technology as the next generation computing paradigm, a new class of computing service systems called "ubiquitous service systems" is introduced. Actually, ubiquitous service systems are service systems incorporating ubiquitous computing technologies (e.g., ubiquitous home network service systems and ubiquitous healthcare service systems) where networked computing devices can always communicate not only with each other but with ubiquitous computing resources in our daily environments (e.g., wireless sensor networks, small embedded computers and distributed information contents). Thus, users can be provided with the service they need, anytime and anywhere through the ubiquitous service systems. For implementing such ubiquitous service systems, there are requirements to be met. One of the essential requirements is that service applications must provide services dynamically based on the

awareness of the current service environments, rather than statically for some pre-programmed environments.

Here is a typical example of ubiquitous services, the "illumination control service." This service system can control indoor brightness according to the current service environments such as user's current location and available service devices in the vicinity of the user. For instance, it raises the window blind if the user is in the living room, or turn on the room lamp if the user is in the bedroom through a PLC (Power Line Communication) controller. Consider another service environment which has lightings with IR (Infrared) switches, and a mobile service robot controllable through a WLAN (Wireless Local Area Network) communication and equipped with an IR remote controller. Even in this case, the service system can also control indoor brightness by controlling the mobile robot to move toward the lighting switch, and then using the infrared controller on the mobile robot. Thus, to implement and provide ubiquitous services efficiently in the real service environments, service applications need to be aware of available service devices and sensors in the user's current location, and then interoperable with them automatically.

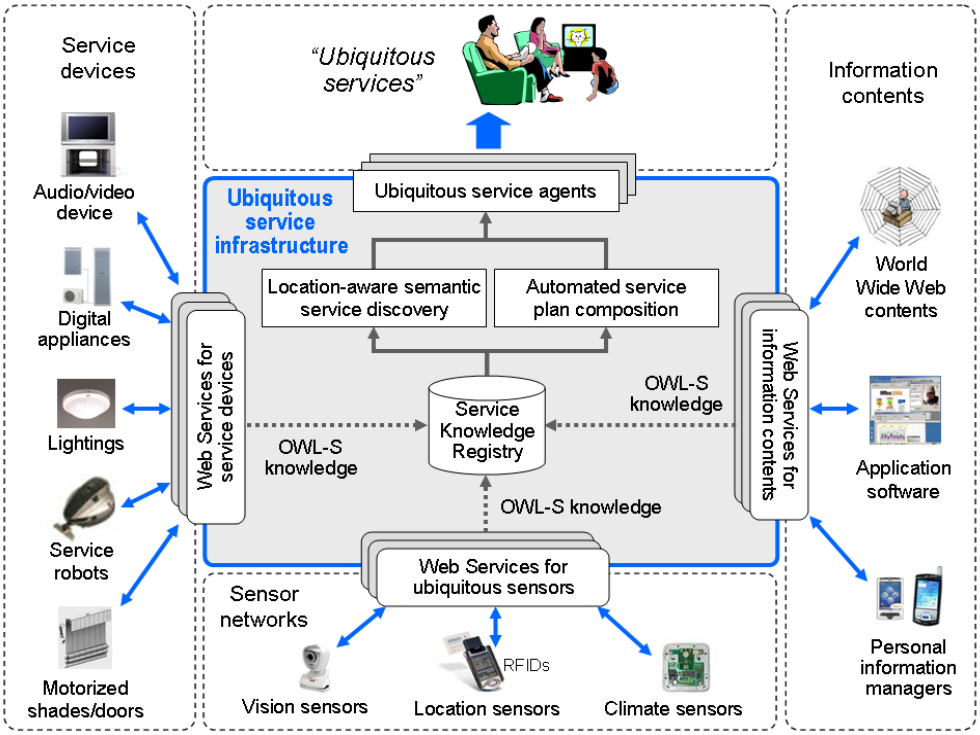

Figure 1 Conceptual architecture of the proposed service infrastructure.

In this paper, we present design and implementation of an infrastructure for ubiquitous service systems. The proposed service infrastructure enables service agents to automatically integrate feasible service devices, wireless sensors and other ubiquitous computing resources considering the user's current location by the use of Semantic Web Services technology [19]. As shown in figure 1, we implement Web Services for various service devices, ubiquitous sensors and information contents as a

unified interface method for accessing them. Then, we describe knowledge about capabilities, interfaces and effective locations of the Web Services in OWL-S (Web Ontology Language for Services) [7], the semantic description language for Web Services. And we register the knowledge to the service knowledge registry so that a ubiquitous service agent can discover the required knowledge to dynamically compose a feasible service execution plan for the current service environments. Finally, the service agent provides the service by automatically interacting with the service devices, ubiquitous sensors and contents through the SOAP (Simple Object Access Protocol) [30], which is a Web Services execution protocol, according to the service plan.

This paper is organized as follows. Section 2 briefly introduces related works and section 3 explains the Semantic Web Services technology as the fundamental background for the paper. Section 4 describes the detailed design of the proposed infrastructure and section 5 explains the prototype implementation and experiments of the proposed infrastructure in our ubiquitous service test bed. As conclusions, section 6 summarizes the paper and discusses some works in progress to extend the proposed service infrastructure.

2 Related Works

In this section, we briefly explore current researches and developments in the ubiquitous services area. And then discuss how the proposed ubiquitous services infrastructure is mainly different from those works and interoperable with some other service platforms accepted as de facto standards.

2.1 Current Researches and Developments

To begin with, we classify related researches and developments into the following three groups according to their basic architectures: OSGi-based (Open Service Gateway Initiative) works, multi-agent-based works, and Web-Services-based works.

One of the recent approaches to ubiquitous service systems is to build them based on the OSGi service platform [26]. OSGi service platform provides a standard Java-based service-hosting environment as well as a set of common APIs (Application Programming Interfaces) for residential gateways to develop dynamically downloadable service bundles. So, OSGi-based works, such as SOCAM [10], D-OSGi [14], and the work of Xiaohu et al. [43] are commonly focused on ubiquitous services in home network environments. Specifically, each work extends the OSGi service platform by supporting ontology-based context models [10], mobility and distribution of services [14], and interoperability with ad-hoc network devices [43] respectively.

Some recent examples of multi-agent-based researches on ubiquitous services include the work of Soldatos et al. [28], the work of Choi et al. [5], the work of Lee et al. [18], and Microsoft's EasyLiving [21]. The core component of those multi-agent-based works is an agent middleware or framework on which each of ubiquitous service provider and perception components is implemented as a distributed agent. Thus, various service provider and perception agents can discover and communicate with each other in a unified way supported by the agent middleware or framework.

Another state-of-the-art approach to a ubiquitous services infrastructure is to apply Web Services architecture. Such Web-Services-based works include Sensor Web [3], Sens-ation [9], and TOPAZ [17], as well as the work proposed in this paper. The main benefit of using Web Services technology is

that it is composed of open Web standards, such as XML (eXtensible Markup Language) [38], WSDL (Web Services Description Language) [32], and SOAP, which provide actually "ubiquitous" interfaces for today's distributed computing and service systems.

To meet the previously mentioned requirement for ubiquitous services, service applications must provide services dynamically based on the awareness of the current service environments, rather than statically pre-programmed for specific environments. However, those existing approaches to ubiquitous service systems are commonly based on proprietary service applications pre-programmed for some specific service environments. For instance, OSGi service bundles [10, 14, 43], JADE (Java Agent DEvelopment framework) agents [28, 5, 18], .NET [21], and SOAP applications [3, 9, 17] for specific service environments (locations, sensors, and devices) are used in the related works. Surely, they can provide users with some level of ubiquitous services in the specific experimental service environments. However, they have practical limitations to provide ubiquitous services the real service environments, which are dynamic, ad hoc and heterogeneous as introduced in the illumination control service example.

2.2 Proposed Infrastructure and Related Works

The fundamental difference between the proposed work and the related works comes from that the proposed work incorporates the Semantic Web Services technology. That is, the proposed infrastructure can effectively support the requirement for ubiquitous services by providing ubiquitous service applications with the location-aware semantic service discovery, automated service plan generation, and execution functionalities based on OWL (Web Ontology Language) [36] and OWL-S [7] knowledge about service locations, ubiquitous sensors and devices. For reference, there are some previous ubiquitous computing researches based on ontology such as CONON (CONtext ONtology) [41], SOUPA (Standard Ontology for Ubiquitous and Pervasive Applications) [4], and myCampus [6]. However, in those works, only OWL is used for modeling and reasoning about contexts in ubiquitous computing environments. On the other hand, in the proposed infrastructure, OWL together with OWL-S is used not only for modeling contexts but for describing, discovering and composing services in ubiquitous computing environments.

In addition, currently OSGi [26] and OGC (Open Geospatial Consortium) Sensor Web [3] provide commonly accepted standard service platforms for heterogeneous devices and sensors. Thus interoperation with ubiquitous sensors and devices deployed with these platforms will be more beneficial. Figure 2 illustrates how the service applications over the proposed services infrastructure can interoperate with devices and sensors deployed with OSGi and Sensor Web platforms. At first, OWL-S knowledge descriptions for all the deployed devices and sensors need to be registered to the service knowledge registry of the proposed infrastructure. Then, as illustrated in the figure, gateway modules for each platform can be used for the interoperation. Ubiquitous service applications can access OSGi devices and sensors using OSGi Gateway Servlets implemented on top of each device and sensor service bundle. An OSGi Gateway Servlet receives SOAP RPC (Remote Procedure Call) request messages for devices and sensors through the HTTP (Hypertext Transfer Protocol) service. It also parses the request messages into appropriate API calls of the corresponding device or sensor service bundle. Ubiquitous service applications can also access sensors on the Sensor Web platform using Sensor Web Gateway Services implemented on top of the Sensor Observation Service. A Sensor

Web Gateway Service also receives SOAP RPC request messages from the service applications, translates the request messages into Sensor Observation Service messages for requested sensors and invokes the Sensor Observation Service with the translated request messages.

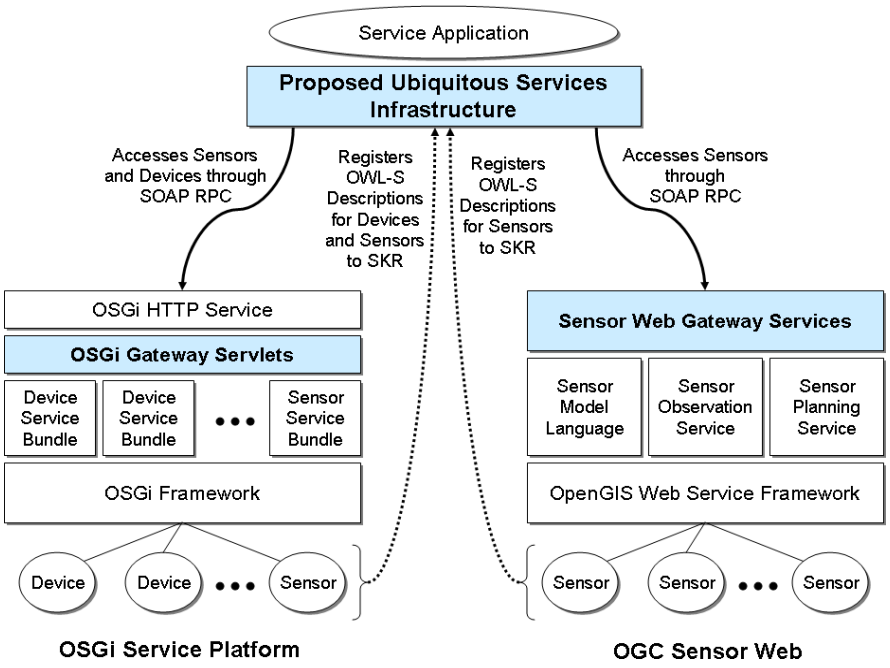

Figure 2 Interoperation between the proposed infrastructure and standard service frameworks.

3 Semantic Web Services Technology

The Web, once a repository of text and images, is evolving into a provider of services: information-providing services, such as Internet information providers and portals; and world-altering services, such as online reservation services and e-commerce applications. Web-accessible programs and databases implement these services through the use of CGI (Common Gateway Interface), Java, ActiveX or Web Services [31]. To have computer programs or agents implement reliable and automated interoperation of such services, the need to make the service computer interpretable is fundamental. That is, to create a Semantic Web [2] of services whose semantics, such as properties, capabilities and interfaces, are encoded in an unambiguous, machine-understandable form that are based on RDF (Resource Description Framework) [33] or OWL [36]. The Semantic Web Services technology is developed to meet this need by describing Web Services with OWL ontology, namely OWL-S. As shown in figure 3, OWL-S service descriptions are founded on the standard Semantic Web and Web Services technology layers including OWL, RDF, WSDL, SOAP and XML.

To specify a richer-level of service semantics, OWL-S provides AI-inspired markups [7] with well-defined meanings. As shown in figure 4, OWL-S markups are grouped into three essential parts for describing the service profile, the service model and the service grounding. An OWL-S description can be encoded with XML, and the following example shows the top level of an OWL-S

service description in XML. Note that in this paper, the semantics of each vocabulary is based on the formal semantics of RDF [34], RDFS (RDF Schema) [35], OWL [37], the service, profile, process and grounding ontologies [22] of OWL-S as it is prefixed by the namespace.

```
<service:Service rdf:ID="ServiceName">
    <service:presents rdf:resource="ServiceProfile"/>
    <service:describedBy rdf:resource="ServiceModel"/>
    <service:supports rdf:resource="ServiceGrounding"/>
</service:Service>
```

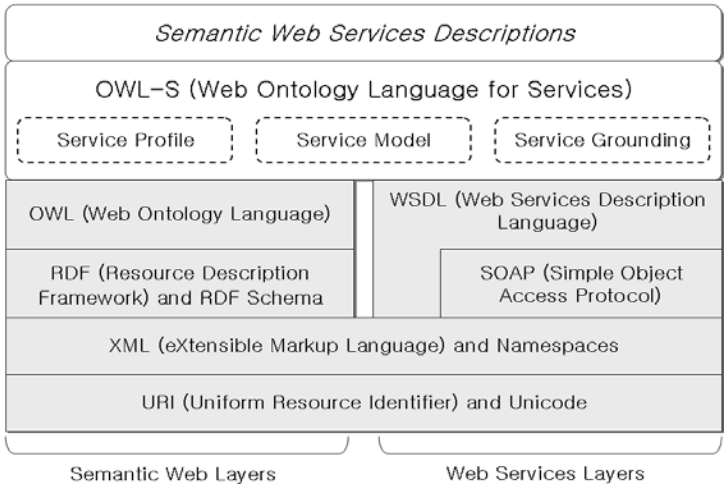

Figure 3 Semantic Web Services Technology layers.

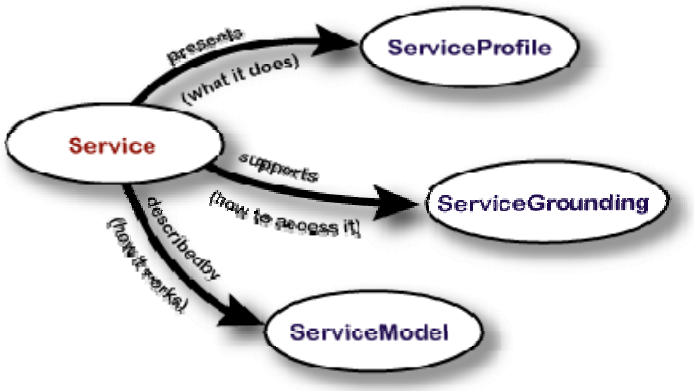

Figure 4 Top level of OWL-S service ontology.

The service profile tells "what the service does," that is, it gives the type of information needed by a service agent to determine whether the service meets its needs. In addition to representing the

capabilities of a service, the service profile can be used to express the needs of the service agent so that a matchmaker has a convenient dual-purpose representation upon which to base its operations. In the proposed service infrastructure, an OWL-S service profile is used to explicitly represent a service request from a user so that a service agent can understand the semantics of the user's service request.

The service model tells "how the service works," that is, it describes what happens when a service is carried out. This description may often be used by a service agent in the following ways: to perform a more in-depth analysis of whether the service meets its needs; to compose service descriptions from multiple services to achieve a specific goal; to coordinate the activities of the different participants during the course of the service enactment; and to monitor the execution of the service. In the proposed service infrastructure, OWL-S service models are used to represent knowledge for composing a plan for the requested service.

The service grounding specifies the details of how an agent can access a service. Typically, a service grounding will specify service operations, message formats for inputs/outputs and other interface-specific details such as port types used in contacting the service. In addition, the service grounding must specify, for each abstract type specified in the service model, an unambiguous way of exchanging data elements of that type with the service. In the proposed service infrastructure, an OWL-S service grounding is used to represent knowledge about how to access various ubiquitous computing resources.

4 Design of the Service Infrastructure

4.1 The Architecture

Figure 5 illustrates the architecture of the proposed ubiquitous service infrastructure based on Semantic Web Services technology. As illustrated in figure 5, the architecture of the proposed infrastructure consists of the following major components: a SAP (Service Agent Platform) that provides service agents with a software platform for automated service discovery and composition; a SA (Service Agent) that is a service application on top of the SAP; a URS (Ubiquitous Resource Service) that is a Web Service for ubiquitous computing resources such as wireless sensors and service devices; and a SKR (Service Knowledge Registry) that is used for registration and discovery of service knowledge including OWL-S descriptions for each URS in service environments.

As shown in the figure, the architecture of the proposed infrastructure is designed so that the major components can interface with each other using the Web Services as an integration bus for the flexibility, extensibility and universal connectivity of the entire service system. For instance, the SAP includes a SAP access object to work as a Web Service itself, that is, the SAP allows every SA with just a Web Service protocol stack to access its essential functionalities. And the SAP can access the SKR and URS's through its Web Service protocol stack even though they are on other sites. In the same manner, newly deployed ubiquitous sensors and devices can be flexibly integrated into the infrastructure by implementing Web Service interfaces for them. And the infrastructure itself can be easily extended with supplemental components by implementing Web Service interfaces for those components. The following subsections will explain the architecture and major components of the proposed ubiquitous service infrastructure in more detail.

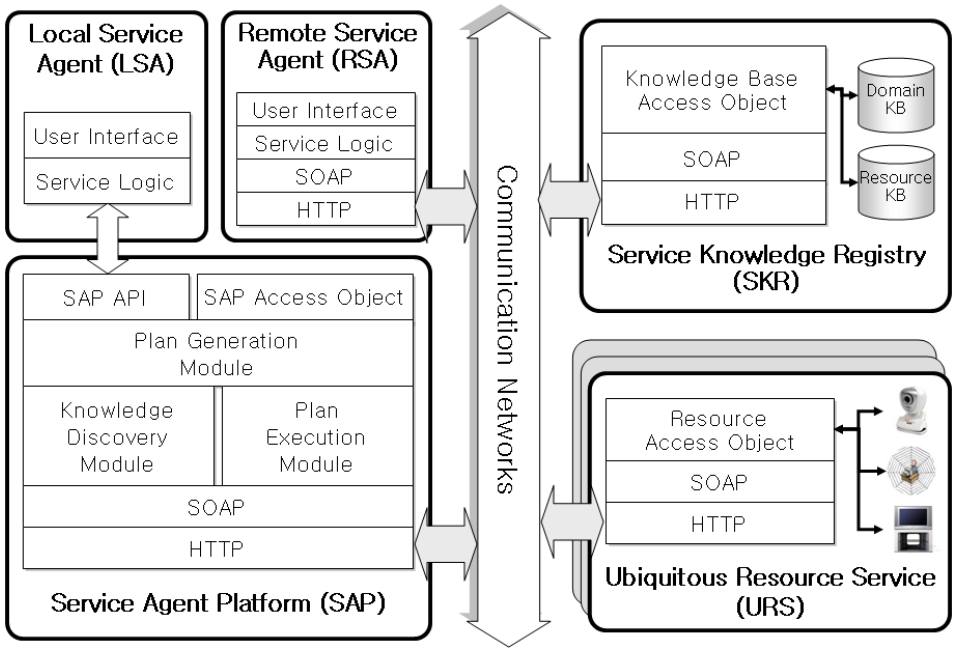

Figure 5 Architecture of the proposed ubiquitous service infrastructure.

4.1.1 SA (Service Agent)

As shown in figure 5, the proposed infrastructure has two types of SA's, one is LSA (Local SA) and the other is RSA (Remote SA). The LSA is implemented directly on top of the SAP API (Application Programming Interface), that is, it is implemented on the same device as SAP (e.g., devices with sufficient computing power such as a PC). On the other hand, the RSA is implemented on a different device from the SAP and communicate with the SAP through the SAP access object using the Web service protocol stack including SOAP and HTTP. Generally, the RSA is implemented on a limited device which has not enough computing resources to run the SAP, such as a PDA (Personal Data Assistant) or a cellular phone. Both types of SA's have a user interface and a service logic that consists of a set of application procedures for ubiquitous services. Once a user inputs a service command with the user interface, the SA encodes the user's command into an OWL-S service request profile and submits the generated request profile to the SAP through the SAP API or the SAP access object.

4.1.2 SAP (Service Agent Platform)

The SAP consists of a SAP API, a SAP access object, a plan generation module, a knowledge discovery module, a plan execution module and a Web service protocol stack. As described above, the SAP (the plan generation module of the SAP precisely) receives a service request in forms of OWL-S service profile through the SAP API or the SAP access object. Then, it discovers required OWL-S knowledge for the requested service from the SKR with the knowledge discovery module. More specifically, to search KB's (Knowledge Bases) in the SKR for the knowledge about a required service model and URS's, the knowledge discovery module creates discovery queries encoded in RDQL [20]

the RDF data query language. One purpose of the knowledge discovery is to gather essential context data for the requested service. For instance, to get the user's current location, it is required to discover knowledge about a URS that provides the user's location data. Another purpose is to perform a feasible service composition. Especially for this purpose, the knowledge discovery module uses the semantic service discovery algorithm (refer to the section entitled "Knowledge Discovery Phase" for details). After the discovery of all the required knowledge, the plan generation module starts to generate a plan for the requested service using an AI-based planning methodology (refer to the section entitled "Plan Generation Phase" for details). Then, the plan execution module translates the service plan into an executable format based on the discovered service grounding knowledge, and executes the plan by accessing the URS's through the Web service protocol stack (refer to the section "Plan Execution Phase" for details).

4.1.3 URS (Ubiquitous Resource Service)

URS's are actually concrete implementations of Web Services for ubiquitous computing resources such as a lighting device, a mobile service robot and RFID (Radio Frequency Identification) sensors (i.e., a light control service, a robot navigation service and a user localization service respectively). Each URS has a resource access object and a WSDL description to represent how to bind to the access object with SOAP. The resource access object can be associated with either one resource or multiple resources that work cooperatively such as a group of RFID sensors to detect where the user is currently. And the URS includes a Web service protocol stack to communicate with the SAP.

4.1.4 SKR (Service Knowledge Registry)

The SKR is a permanent storage of knowledge necessary to provide ubiquitous services. The SKR contains two kinds of KB's: a domain KB and a resource KB. The domain KB stores OWL-S composite process models, each of which represents a generic service model for a certain domain called domain service model. The domain KB also stores OWL ontology of general concepts and properties, called concept ontology, to describe IOPE (Input, Output, Precondition and Effects) and location of URS's. The resource KB stores OWL-S atomic services with corresponding groundings, each of which represents a URS for specific service environments. Some specific examples of such service knowledge will be given in the experiments section. In addition to the knowledge bases, the SKR includes the knowledge base access object to handle RDQL queries and a Web Service protocol stack to communicate with the SAP.

Figure 6 illustrates the overall service procedure of the proposed infrastructure and the activities of each component during the procedure, which is divided into the context data gathering and the service composition sub-procedures (for reference, the dashed arrows in the figure mean SOAP messaging). As illustrated in the figure, the service procedure starts with submitting the service request profile to the plan generation module of the SAP through the SAP API or the SAP access object. Especially, the service composition is broken down into three phases: knowledge discovery phase, plan generation phase and plan execution phase. The following sections will explain each service composition phase in more detail.

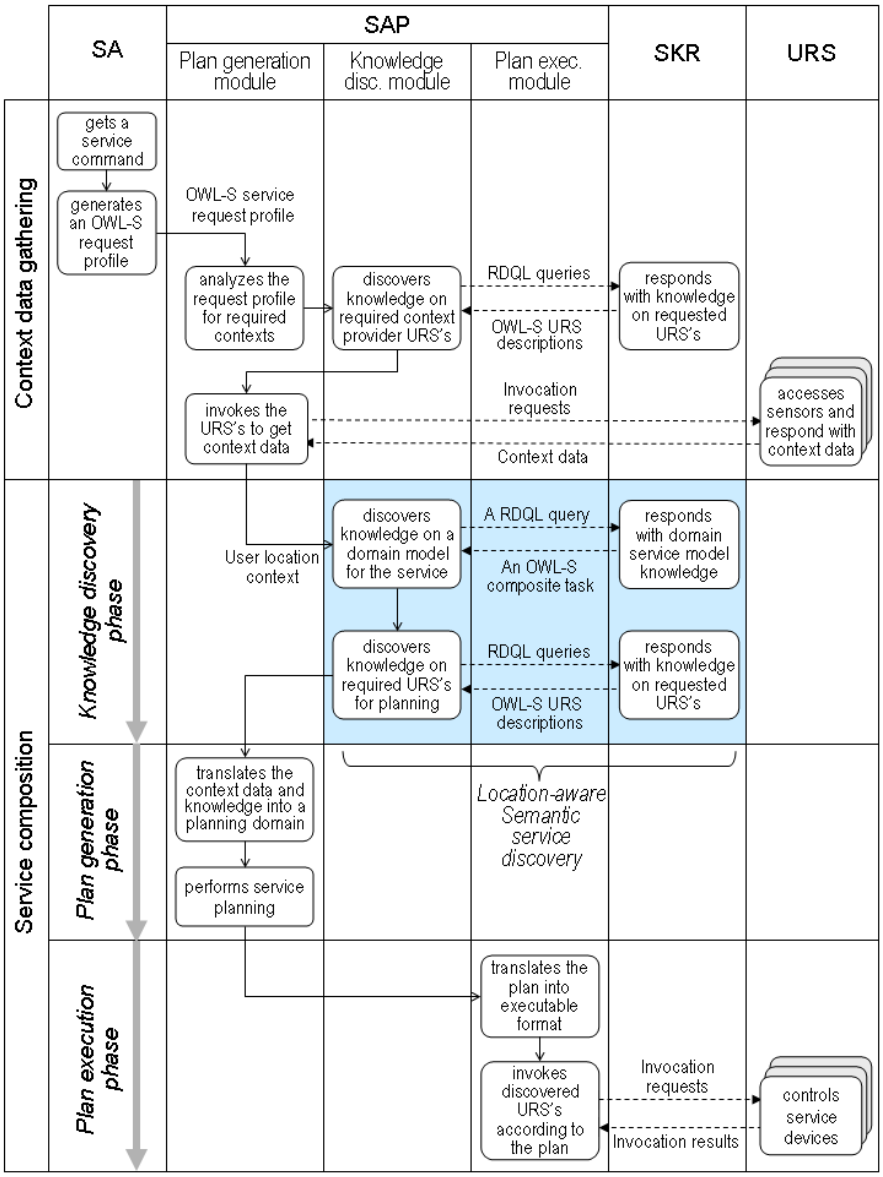

Figure 6 Overall procedure of the proposed service infrastructure.

4.2 Knowledge Discovery Phase

As mentioned above, knowledge about domain service models and URS's are described with OWL-S so that the SAP can automatically discover the required knowledge to compose a feasible service plan for a user request. Knowledge about URS's is used to generate the space of feasible actions in specific service environments during the plan generation phase and knowledge about domain service models is used to constrain how the services are to be planned independently of specific service environments.

As shown in figure 6, the knowledge discovery is actually performed by sending discovery queries to the SKR and receiving the responses through SOAP messages. In the course of query messaging, the knowledge about a domain service model is found from the domain KB, and then the knowledge about required URS's is found from the resource KB based on the user's location context. In the knowledge discovery module, we implement the location-aware semantic service discovery algorithm to generate a feasible plan for a requested service. Prior to explaining the details of the proposed service discovery algorithm, we will shortly introduce existing service discovery approach and discuss its limitation in ubiquitous service environments.

4.2.1 Existing Service Discovery Approaches

Currently existing service discovery technologies use interface-based, attribute-based or unique-identifier-based query matching algorithms at a syntactic level (i.e., matching the name of services and the type of arguments). For instance, Sun Microsystem's Jini [16], a service is described with a Java interface definition and registered to a service lookup server. Then a Jini client can discover a service from the lookup server by syntactically matching with the interface description of the service. Microsoft's UPnP (Universal Plug and Play) [29] uses a syntactic service attribute matching, based on service descriptions in XML. And other existing service discovery technologies such as SLP (Service Location Protocol) [25], DEAPspace [24], UDDI (Universal Description, Discovery, and Integration) for Web Services [31], OSGi Service Registry [26], and most of the agent directory services also use syntactic level matching mechanism.

However, the existing service discovery approaches based on syntactic matching are inefficient for ubiquitous computing environments where the variety and heterogeneity of service implementations and their interfaces exist. For instance, we can have the same service implement different interfaces (i.e., arguments with different names or types), which could result in the failure of a syntactic match if the service query does not match with any interface. Suppose that the SKR have knowledge of a user localization service that has an "IndoorLocation" as its output. If a service agent tries to discover a user localization service that has a "Location" as its output using a syntactic level matching algorithm, it will fail because there is no exact match for the requested output name. In addition, there are no methods to discover a service for a specific location in the existing service discovery system. For instance, a service agent can not distinguish a service for the living room from one for the bedroom. And if a service discovery fails for the user's current location, an agent can not decide to discover alternatively a service for a location including the user's location. So, we need a more expressive method to describe and discover services in a semantic and location-aware manner for ubiquitous service environments.

4.2.2 Location-aware Semantic Service Discovery

The proposed location-aware semantic service discovery algorithm can overcome the limitations of the existing service discovery approach by matching with a semantically compatible (replaceable) service for every URS that is not discovered by exact query matching. For instance, the previously mentioned problem of syntactic service discovery can be resolved by matching with a service that has an output which is equivalent to or a subclass of "Location" (i.e., IndoorLocation) by reasoning about the concept ontology. And the proposed service discovery algorithm can also overcome the location-

awareness limitation of the existing service discovery approach by using the "effectiveLocation" property to describe and reason about the location where a URS has effect. For example, if a discovery of a URS having effect in the living room fails, a service agent can retry to discover an alternative URS having effect in an indoor location including the living room. The following OWL code shows a part of the concept ontology that defines a "Location" concept hierarchy and the "effectiveLocation" property.

```
<!-- Partial definition of "Location" concept hierarchy. -->
<owl:Class rdf:ID="Location"/>
    <rdfs:subClassOf  rdf:resource="&process;#Output"/>
</owl:Class>
<owl:Class rdf:ID="IndoorLocation">
    <rdfs:subClassOf  rdf:resource="Location"/>
</owl:Class>
<owl:Class rdf:ID="OutdoorLocation">
    <rdfs:subClassOf  rdf:resource="Location"/>
</owl:Class>
<owl:Class rdf:ID="Bedroom">
    <rdfs:subClassOf  rdf:resource="IndoorLocation"/>
</owl:Class>
<owl:Class rdf:ID="LivingRoom">
    <rdfs:subClassOf  rdf:resource="IndoorLocation"/>
</owl:Class>
<owl:Class rdf:ID="Kitchen">
    <rdfs:subClassOf  rdf:resource="IndoorLocation"/>
</owl:Class>
                    …

<!-- Definition of "effectiveLocation" property. -->
<owl:ObjectProperty rdf:ID="effectiveLocation">
    <rdfs:domain rdf:resource="&process;#Process" />
    <rdfs:range rdf:resource="#Location" />
</owl:ObjectProperty>
```

As formerly explained, knowledge about domain service models and URS's are described as OWL-S composite and atomic processes respectively, which have OWL individuals [36] as their IOPE property [7] and "effectiveLocation" property values. For instance, an atomic process "GetUserLocationService" in the example of the experiments section has an OWL individual "UserLocation_Output" (an instance of "IndoorLocation" class) as a value of its "hasOutput" property, and an OWL individual "MyRooms" (an instance of "IndoorLocation" class) as a value of its "effectiveLocation" property. Therefore, we need to define some semantic relations between OWL individuals and OWL-S processes or profiles based on a predicate logic to describe the service discovery algorithm formally: that is, $\boldsymbol{p}(\boldsymbol{s}, \boldsymbol{o})$, where $\boldsymbol{p}$ is a predicate, $\boldsymbol{s}$ is a subject of $\boldsymbol{p}$, and $\boldsymbol{o}$ is an

object of ***p***. An OWL and OWL-S statement can be translated into a predicate logic representation by mapping a property into ***p***, a domain value of the property into ***s***, and a range value of the property into ***o***. To help understand the formal definitions, predicate logic representations for OWL-S atomic processes, "GetUserLocationService" and "TurnOnLightService" in the experiments section are given as follows.

// Predicate logic representation of "GetUserLocationService."
rdf:type(UserLocation_Output, IndoorLocation)
rdf:type(MyRooms, IndoorLocation)
process:hasOutput(GetUserLocationService, UserLocation_Output)
concepts:effectiveLocation(GetUserLocationService, MyRooms)

// Predicate logic representation of "TurnOnLightService."
rdf:type(LightOn_Effect, LightOn)
rdf:type(MyBedroom, Bedroom)
process:hasEffect(TurnOnLightService, LightOn_Effect)
concepts:effectiveLocation(TurnOnLightService, MyBedroom)

Definition 1 (Individual subsumption relation, $x \supseteq y$): An individual ***x*** of OWL Class ***a*** subsumes an individual ***y*** of OWL Class ***b*** iff rdfs:subClassOf(***b***, ***a***) $\vee$ owl:equivalentClass(***a***, ***b***) holds.

Definition 2 (Individual subsumption relation on a property, $x \supseteq y / p$): An individual ***x*** of an OWL Class subsumes an individual ***y*** of an OWL Class on an OWL Property ***p*** iff for each OWL individual $a \in \{v \mid p(x, v)\}$, there exists a distinct OWL individual $b \in \{u \mid p(y, u)\}$ s.t. $a \supseteq b$.

Definition 3 (A service, *x*): A service ***x*** is described with an individual of OWL-S Process or Profile. If any of its hasInput, hasOutput, hasPrecondition and hasEffect property is not specified, a service has a default value of *No_Input*, *No_Output*, *No_Precond* or *No_Effect* accordingly for that property.

Definition 4 (Service compatibility relation, $x \approx y$): A service ***x*** is compatible with a service ***y*** iff ($x \supseteq y$ / hasInput) $\wedge$ ($x \supseteq y$ / hasPrecondition) $\wedge$ ($y \supseteq x$ / hasOutput) $\wedge$ ($y \supseteq x$ / hasEffect) holds.

Definition 5 (Service equivalence relation $x = y$): A service ***x*** is equivalent to a service ***y*** iff ***x*** and ***y*** have the same set of IOPE property values.

Definition 6 (Location-based service effect relation, $x \parallel l$): A service ***x*** has effect in a location ***l***, i.e., effectiveLocation(***x***, ***l***), where ***l*** is an individual of OWL Class Location. And a service ***x*** satisfying $x \parallel l_s$ also satisfies $x \parallel l_u$ iff ($l_s \supseteq l_u$) holds, where both l_s and l_u are individuals of OWL Class Location.

And the location-aware semantic service discovery algorithm (Location_Aware_SSD) is given as follows.

```
Procedure Location_Aware_SSD (R, L, M, U)
  Input: a service request profile R, location of the user L;
  Outputs: a domain service model M, a set of discovered URS's U;
{
  U ← Ø;

  Discover every domain service model from the SKR whose composite process = R;
  For each discovered domain service model M {
        Call URS_Discovery(L, M, U);
        If (the URS discovery succeeds) return with SSD_SUCCESS;
  }
  Discover every domain service model from the SKR whose composite process ≈ R;
  For each discovered domain service model M {
        Call URS_Discovery(L, M, U);
        If (the URS discovery succeeds) return wirh SSD_SUCCESS;
  }

  return with SSD_FAILURE;
} // End of procedure Location_Aware_SSD

Procedure URS_Discovery(L, M, U)
  Input: location of the user L, a domain service model M;
  Output: a set of discovered URS's U;
{
  For each component process P in M {
        If (P is a composite process) call URS_Discovery(L, P, U);

        Discover a URS S from the SKR with atomic process A s.t. (A = P) ∧ (A || L);
        If (the discovery succeeds) add S to U;
        Else {
              Discover a URS S from the SKR with atomic process A s.t. (A ≈ P) ∧ (A || L);
              If (the discovery succeeds) {
                    Add S to U;
                    Replace P in the service model M with A;
              } Else, return with URS_DISCOVERY_FAILURE;
        }
  }

  return with URS_DISCOVERY_SUCCESS
} // End of procedure URS_Discovery
```

The inputs of the algorithm are a service request profile ***R*** and the location of the user ***L***, and the outputs of the algorithm are a discovered domain service model ***M*** and a set of discovered URS's ***U***. To begin with, the algorithm discovers every domain service model whose composite process is equivalent to the service request profile ***R*** in accordance with the service equivalence relation (definition 5). If the discovery fails, the algorithm tries to discover every domain service model that has a semantically compatible composite process with ***R*** in accordance with the service compatibility relation (definition 4). On success of the discovery, the URS_Discovery algorithm is invoked to discover URS's that have effect in ***L*** and have an equivalent atomic process to each component atomic process of the domain service model ***M*** in accordance with the service equivalence relation and location-based service effect relation (definition 5 and 6). When a match is found, the algorithm adds the matched service to the set of URS's ***U***. On a failure, the algorithm tries to discover a URS that has effect in ***L*** and has a semantically compatible atomic process for the failure in accordance with the service compatibility relation and location-based service effect relation (definition 4 and 6). If the discovery succeeds, the algorithm adds the discovered service to the set of discovered URS's ***U*** and extends the domain service model ***M*** by replacing the corresponding component process with the atomic process of the newly added URS. If every required URS is discovered, the algorithm terminates successfully.

4.3 Plan Generation Phase

In the plan generation phase, a feasible service plan for the user request is automatically generated using HTN (Hierarchical Task Network) planning [8]. HTN planning is an AI planning methodology to resolve planning problems by a task decomposition process in which the planner decomposes tasks into smaller subtasks until primitive tasks, which can be performed directly, are found. The entire task decomposition process is based on planning operators and methods called planning domain. Such task decomposition concept and modularity of HTN planning is very similar to the concept of composite and atomic processes in OWL-S. In addition, HTN planning supports expressive precondition representation to solve the complex planning problems efficiently. To perform HTN planning, it is required to translate OWL-S knowledge found in the discovery phase into the HTN planning domain. That is, translate a domain service model ***M*** into the planning methods and a set of discovered URS's ***U*** into the planning operators (note that ***M*** and ***U*** are outputs from the service discovery algorithm). We programmed such translation algorithm into the plan generation module of the SAP. Because we use a domain-independent HTN planning system, SHOP2 (Simple Hierarchical Ordered Planner) for the implementation, generating a planning domain in terms of SHOP2 domain is required. Inheriting a generic HTN domain, each operator of SHOP2 describes what needs to be done to accomplish some primitive task and each method tells how to decompose some compound task into partially ordered subtasks [23].

A SHOP2 operator is an expression of the form (:operator (!***p v***) (***Pre***) (***Del***) (***Add***)), where ***p*** is a primitive task with a list of input parameters ***v***, ***Pre*** represents the operator's preconditions, ***Del*** represents the operator's delete list that includes a list of things that will become false after an operator's execution, and ***Add*** represents the operator's add list that includes a list of things that will become true after the operator's execution. Knowledge about a URS, described with an OWL-S atomic process ***A***, is translated into a SHOP2 planning operator by replacing ***p*** with ***A***, ***v*** with a set of inputs of ***A***, ***Pre*** with conjunction of all the preconditions of ***A***, and ***Add*** with conjunction of all the effects of ***A***.

A SHOP2 method is an expression of the form (:method (***c v***) (***Pre*$_1$**) (***T*$_1$**) (***Pre*$_2$**) (***T*$_2$**) …), where ***c*** is a compound task with a list of input parameters ***v***, each ***Pre*$_i$** is a precondition expression, and each ***T*$_i$** is a partially ordered set of subtasks with input parameters for ***Pre*$_i$**. Knowledge about a domain service model, described with an OWL-S composite process ***C*** with component processes $\boldsymbol{P}_{1 \le i \le n}$, is translated into a set of SHOP2 planning methods according to the control construct of ***C***. For example, the specific form (:method (***c v***) (***Pre***) (***T***)) is used to express a method for a composite process with a "Sequence" control construct, where ***c*** is replaced with ***C***, ***v*** with a set of inputs of ***C***, ***Pre*** with conjunction of all the preconditions of ***C***, and ***T*** with an ordered list of ($\boldsymbol{P}_1$ … $\boldsymbol{P}_n$).

Consequently, SHOP2 planner generates a total ordered set of operators based on the planning methods. Before the plan execution phase, the output of SHOP2 planner is encoded with XML for the future interoperability with other service agents or applications. Some running examples of XML-encoded service plans will be given in the experiments section. For more references to SHOP2, see [23, 27].

4.4 Plan Execution Phase

The result of the plan composition phase is a sequence of primitive tasks encoded in XML, that is, a sequence of OWL-S atomic processes of discovered services without any access information. To be executed, a service plan must be translated into an executable format such as a process of concrete Web Services operations and messages defined in WSDL descriptions. For the implementation of the proposed service infrastructure, BPEL4WS (Business Process Execution Language for Web Services) [12] is used to create the executable Web Service processes at the plan execution phase. During the creation of the executable process, service groundings for discovered URS's are used to generate a BPEL4WS process from the output of the plan composition phase. The following example shows a part of a BPEL4WS process generated during the third experiment in the experiments section to move the mobile service robot to a specific location (i.e., the kitchen) and turn on the IR-controlled light switch using the IR remote controller equipped on the robot. It includes a sequence of two "invoke" statements of concrete Web Services operations and messages: the first one is for the "moveToKitchen" operation of the Web Service "NavigationService" and the second one is for the "turnOnLight" operation of the Web Service "LightControlService." Note that such access information about concrete Web Services comes from the relevant service groundings.

```
<process  xmlns:ws1="http://urshost.etri.re.kr:8080/axis/NavigationService"
          xmlns:ws2="http://urshost.etri.re.kr:8080/axis/LightControlService">
    …
  <sequence>
    …
    <invoke  portType="ws1:navigationServicePort"
             operation="moveToKitchen"  …  />

    <invoke  portType="ws2:lightControlServicePort"
             operation="turnOnLight"  …  />
    …
  </sequence>
</process>
```

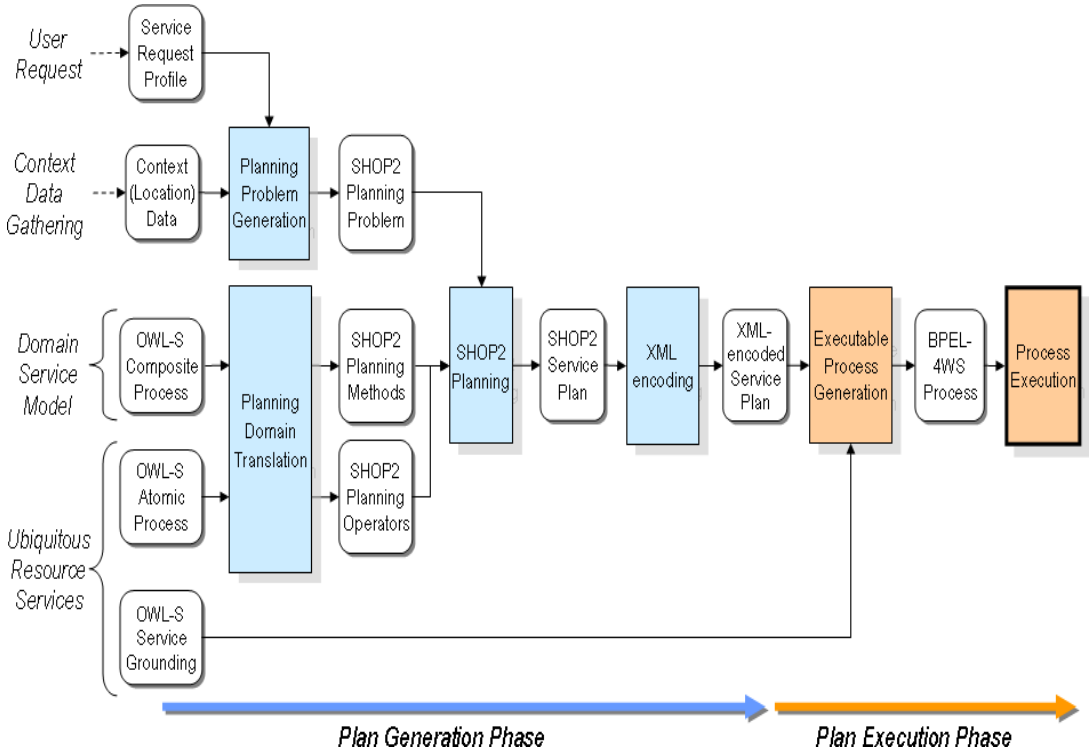

Figure 7 Data flow diagram for generating and executing a service plan.

Figure 7 illustrates a data flow diagram for generating an executable BPEL4WS process from a user request profile, user location context, discovered OWL-S knowledge about a domain service model and URS's during the service composition procedure. For reference, colored rectangles in the diagram represent major processes of the plan composition and execution phases, and rounded rectangles in the diagram represent data (inputs and outputs) for each process.

5 Implementation and Experiments

5.1 Experimental Ubiquitous Service System

We implemented a prototype ubiquitous service system over the proposed service infrastructure and made experiments in our ubiquitous service test bed, which has a bedroom, kitchen and living room as shown in figure 8. The test bed includes IR-controlled devices such as lighting switches in the kitchen, a PLC controller and PLC-controlled devices such as a bedroom lamp, an air conditioner and a motorized window blind in the living room. The test bed also includes a WLAN-enabled mobile service robot which is equipped with an IR remote controller. In addition, there are three RFID readers in each place of the test bed to sense the location of the user who carries a RFID tag. The implementation of the proposed service infrastructure consists of a URS host, an SKR server and a SAP system and a laptop with a remote SA. They are connected to each other via a wired and wireless LAN (Local Area Network) as illustrated in figure 9.

Figure 8 Ground plan and snapshot of the test bed for experiments.

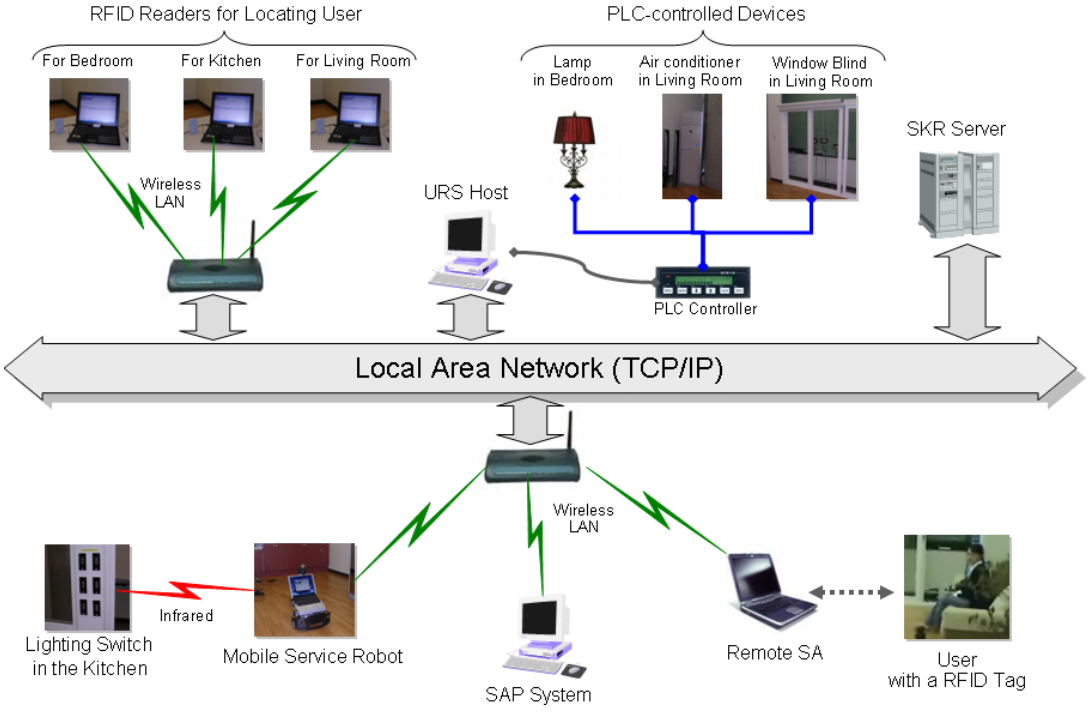

Figure 9 Network configuration of the test bed for experiments.

We use the Java2 SDK (Software Development Kit) as the development platform for the prototype implementation. We also use HP's Jena Semantic Web framework [11] to implement OWL ontology processing and reasoning functionality, and IBM's BPWS4J (BPEL4WS for Java) platform [13] to

implement BPEL4WS processing functionality. The Apache Axis Web Services toolkit [1] is used to implement Web Services and SOAP communications between the major components of the prototype implementation. Table 1 summarizes concrete Web Services and OWL-S descriptions for the URS's, which access the PLC controller, the RFID readers, the mobile service robot and the IR remote controller equipped on the robot in the test bed. Each Web Service is implemented in the URS host.

Table 1 Web Service implementations and OWL-S description for URS's.

Web Service	***Operation***	***Devices***	***OWL-S process***	***Effective location***	***IOPE***
User Location Service	getUser Location	RFID readers	GetUser Location Service	MyRooms	O = UserLocation_Output
Air Conditioner Control Service	turnOn AirConditioner	PLC controller	TurnOnAir Conditioner Service	MyRooms	E = AirConditionerOn_Effect
	turnOff AirConditioner	PLC controller	TurnOffAir Conditioner Service	MyRooms	E = AirConditionerOff_Effect
Window Blind Control Service	raise WindowBlind	PLC controller	RaiseWidow BlindService	MyLiving Room	E = BlindRaised_Effect
	pullDown Window Blind	PLC controller	PullDown WidowBlind Service	MyLiving Room	E = BlindDown_Effect
Lamp Control Service	turnOn Lamp	PLC controller	TurnOnLamp Service	My Bedroom	E = LampOn_Effect
	turnOff Lamp	PLC controller	TurnOffLamp Service	My Bedroom	E = LampOff_Effect
Light Control Service	turnOn Light	IR remote controller	TurnOnLight Service	MyKitchen	P = RobotAtKitchen_Precond E = LightOn_Effect
	turnOff Light	IR remote controller	TurnOffLight Service	MyKitchen	P = RobotAtKitchen_Precond E = LightOff_Effect
Navigation Service	moveTo Kitchen	Mobile robot	MoveTo Kitchen Service	MyKitchen	E = RobotToKitchen_Effect

Table 2 lists OWL-S composite processes of domain service models for illumination control and climate control service experiments. As shown in the table, the first two OWL-S processes are domain service models to control indoor brightness (i.e., having BecomeBrighter_Effect), but have different component processes and control structures. The last OWL-S process is a domain service model to control indoor temperature (i.e., having BecomeCooler_Effect).

Table 2 OWL-S composite processes for domain service models.

OWL-S process	***IOPE***	***Description***
RaiseBrightness ServiceModel	E = BecomeBrighter_Effect	A domain service model to raise indoor brightness
RaiseBrightness RobotServiceModel	E = BecomeBrighter_Effect	Another domain service model to raise indoor brightness using a mobile service robot
LowerTemperature ServiceModel	E = BecomeCooler_Effect	A domain service model to lower indoor temperature

5.2 Experimental Service Knowledge Bases

The experimental domain KB contains OWL-S descriptions of domain service models and the concept ontology for the ubiquitous service test bed. The following example knowledge shows a portion of hierarchies of "BecomeBrighter" and "BecomeCooler" concepts defined in the concept ontology which are actually used in the experiments.

```
<owl:Class rdf:ID="BecomeBrighter">
    <rdfs:subClassOf rdf:resource="&process;UnConditionalEffect"/>
</owl:Class>
<owl:Class rdf:ID="LampOn">
    <rdfs:subClassOf rdf:resource="#BecomeBrighter"/>
</owl:Class>
<owl:Class rdf:ID="BlindRaised">
    <rdfs:subClassOf rdf:resource="#BecomeBrighter"/>
</owl:Class>
        …
<owl:Class rdf:ID="BecomeCooler">
    <rdfs:subClassOf rdf:resource="&process;UnConditionalEffect"/>
</owl:Class>
<owl:Class rdf:ID="AirConditionerOn">
    <rdfs:subClassOf rdf:resource="#BecomeCooler"/>
</owl:Class>
```

And the following example knowledge shows two different OWL-S composite processes of domain service models to control brightness of the experimental service environment. Especially, the "RaiseBrightnessRobotServiceModel" incorporates a mobile service robot to control indoor brightness. It is composed of a sequence of two atomic processes, one for navigating the robot to the user's current location, and another for raising the brightness of the location. In more detail, the first atomic process has an instance of "RobotToLocation" concept as its effect, and the second one has an instance of "RobotAtLocation" concept as its precondition and an instance of "BecomeBrighter" concept as its

effect. Note that "BecomeBrighter" is a super-concept of "LampOn," "LightOn" and "BlindRaised" as defined in the concept hierarchy.

```
<!-- Domain service model to control indoor brightness -->
<concepts:BecomeBrighter rdf:ID="BecomeBrighter_Effect"/>

<process:ProcessModel rdf:ID="RaiseBrightnessProcessModel">
    <process:hasProcess>
        <process:CompositeProcess rdf:ID="RaiseBrightnessServiceModel">
            <process:hasEffect rdf:resource="#BecomeBrighter_Effect"/>
            <process:composedOf>
                <process:Sequence>
                    <process:components rdf:parseType="Collection">
                        <process:AtomicProcess>
                            <process:hasEffect rdf:resource="#BecomeBrighter_Effect"/>
                        </process:AtomicProcess>
                    </process:components>
                </process:Sequence>
            </process:composedOf>
        </process:CompositeProcess>
    </process:hasProcess>
</process:ProcessModel>

<!-- Another domain service model to control indoor brightness using a service  robot -->
<concepts:RobotToLocation rdf:ID="RobotToLocation_Effect"/>
<concepts:RobotAtLocation rdf:ID="RobotAtLocation_Precond"/>

<process:ProcessModel rdf:ID="RaiseBrightnessRobotProcessModel">
    <process:hasProcess>
        <process:CompositeProcess rdf:ID="RaiseBrightnessRobotServiceModel">
            <process:hasEffect rdf:resource="#BecomeBrighter_Effect"/>
            <process:composedOf>
                <process:Sequence>
                    <process:components rdf:parseType="Collection">
                      <process:AtomicProcess>
                        <process:hasEffect rdf:resource="#RobotToLocation_Effect"/>
                      </process:AtomicProcess>
                      <process:AtomicProcess>
                        <process:hasPrecondition rdf:resource="#RobotAtLocation_Precond"/>
                        <process:hasEffect rdf:resource="#BecomeBrighter_Effect"/>
                      </process:AtomicProcess>
                    </process:components>
                </process:Sequence>
            </process:composedOf>
        </process:CompositeProcess>
    </process:hasProcess>
</process:ProcessModel>
```

The following example knowledge shows OWL-S composite process of a domain service model to control temperature of the experimental service environment.

```
<!-- Domain service model to control indoor temperature -->
<concepts:BecomeCooler rdf:ID="BecomeCooler_Effect"/>

<process:ProcessModel rdf:ID="LowerTemperatureProcessModel">
    <process:hasProcess>
        <process:CompositeProcess rdf:ID="LowerTemperatureServiceModel">
            <process:hasEffect rdf:resource="#BecomeCooler_Effect"/>
            <process:composedOf>
                <process:Sequence>
                    <process:components rdf:parseType="Collection">
                        <process:AtomicProcess>
                            <process:hasEffect rdf:resource="#BecomeCooler_Effect"/>
                        </process:AtomicProcess>
                    </process:components>
                </process:Sequence>
            </process:composedOf>
        </process:CompositeProcess>
    </process:hasProcess>
</process:ProcessModel>
```

The experimental resource KB contains OWL-S knowledge about the URS's listed in table 1. The following example shows OWL-S knowledge about "turnOnLamp" operation of Web Service "LampControlService" that turns on the bedroom lamp, and "turnOnAirConditioner" operation of Web Service "AirConditionerControlService" that turns on the air conditioner in the living room. As described in the example, the knowledge about "turnOnLamp" operation consists of an atomic process "TurnOnLampService" and its service grounding "TurnOnLampServiceGrounding." The "TurnOnLampService" has an instance of "LampOn" concept as its effect, which is a sub-concept of the "BecomeBrighter." And it also has "MyBedroom" an instance of "Bedroom" concept as its effective location, which is a sub-concept of the "IndoorLocation." And the service grounding provides information to access the "turnOnLamp" service operation of the Web Service "http://urshost.etri.re.kr:8080/axis/LampControlService."

```
<!-- OWL-S atomic process for "turnOnLamp" operation of "LampControlService" -->
<concepts:LampOn rdf:ID="LampOn_Effect"/>
<concepts:Bedroom rdf:ID="MyBedroom"/>

<process:ProcessModel rdf:ID="TurnOnLampProcessModel">
    <process:hasProcess>
        <process:AtomicProcess rdf:ID="TurnOnLampService">
            <process:hasEffect rdf:resource="LampOn_Effect"/>
            <concepts:effectiveLocation rdf:resource="MyBedroom"/>
        </process:AtomicProcess>
    </process:hasProcess>
</process:ProcessModel>
```

```
<!-- OWL-S service grounding for "turnOnLamp" operation -->
<grounding:WsdlGrounding rdf:ID="TurnOnLampServiceGrounding">
    <grounding:hasAtomicProcessGrounding>
        <grounding:WsdlAtomicProcessGrounding>
            <grounding:wsdlDocument>
                http://urshost.etri.re.kr:8080/axis/LampControlService?wsdl
            </grounding:wsdlDocument>
            <grounding:wsdlOperation>
                <grounding:WsdlOperationRef>
                    <grounding:portType>LampControlService</grounding:portType>
                    <grounding:operation>turnOnLamp</grounding:operation>
                </grounding:WsdlOperationRef>
            </grounding:wsdlOperation>
        </grounding:WsdlAtomicProcessGrounding>
    </grounding:hasAtomicProcessGrounding>
</grounding:WsdlGrounding>
```

And the knowledge about "turnOnAirconditioner" operation consists of an atomic process "TurnOnAirConditionerService" and "TurnOnAirConditionerServiceGrounding." Note that "TurnOnAirConditionerService" has an instance of "IndoorLocation" concept "MyRooms" as its effective location.

```
<!-- OWL-S atomic process for "turnOnAirConditioner" operation -->
<concepts:AirConditionerOn rdf:ID="AirConditionerOn_Effect"/>
<concepts:IndoorLocation rdf:ID="MyRooms"/>

<process:ProcessModel rdf:ID="TurnOnAirConditionerProcessModel">
    <process:hasProcess>
        <process:AtomicProcess rdf:ID="TurnOnAirConditionerService">
            <process:hasEffect rdf:resource="AirConditionerOn_Effect"/>
            <concepts:effectiveLocation rdf:resource=" MyRooms"/>
        </process:AtomicProcess>
    </process:hasProcess>
</process:ProcessModel>

<!-- OWL-S service grounding for "turnOnAirConditioner" operation is omitted -->
                    …
```

5.3 Experimental Results

In the experiments, we will demonstrate how a ubiquitous service system built on the proposed service infrastructure can automatically integrate ubiquitous service devices and sensors based on the user's current location in the test bed. In the beginning of the experiment, the user is in the living room carrying her RFID tag. For the first experiment, the user commands the SA to "Make it brighter." Once inputted, the user's command is encoded into the following OWL-S service request profile that has

"BecomeBrighter_Effect," an instance of "BecomeBrighter" concept, as its effect. And then the service request profile is submitted to the SAP.

```
<profile:Profile rdf:ID="ServiceRequest">
    <profile:hasEffect rdf:resource="BecomeBrighter_Effect"/>
</profile:Profile>
```

As explained above, to compose a feasible service plan for the requested service, the SAP must know the current location of the user as an essential context for services. So, the SAP sends the SKR server a discovery query for a service that has an instance of "IndoorLocation" concept as its output, and has effect in every indoor location. As the result of querying, the SAP receives "GetUserLocationService," OWL-S knowledge about the "getUserLocation" service (refer to table 1), and executes it based on the service grounding "GetUserLocationServiceGrounding." When executed, the "getUserLocation" service responds to the SAP with "MyLivingRoom" as user's current location by scanning the RFID readers installed in each place of the test bed.

After knowing the current location of the user, the SAP discovers every domain service model that has an instance of "BecomeBrighter" concept or its sub-concept as its effect. As the result of discovery, the SAP receives OWL-S knowledge about the indoor brightness control service models (refer to table 2), and continues to discover URS's for each domain service model until the URS discovery for the service model is successfully completed. In the first experiment, the URS discovery succeeds with the "RaiseBrightnessServiceModel" in table 2. During the URS discovery, the SAP gets knowledge about feasible URS's by using the proposed location-aware semantic service discovery algorithm. At this time, the SAP tries to discover a URS that has effect in the living room, and has an instance of "BecomeBrighter" concept as its effect by sending an exact matching query according to the domain service model. When it fails, the SAP retries to discover a URS that has effect in the living room and has an instance of "LampOn" or "BlindRaised" concept as its effect, which is a sub-concept of the "BecomeBrighter" concept. As the result of querying, "RaiseWindowBlindService," OWL-S knowledge about the "raiseWindowBlind" service (refer to table 1), is discovered. And then the following plan is generated after performing the plan generation phase. Consequently, the plan was translated into the BPEL4WS process and successfully executed during the plan execution phase as shown in figure 10.

```
<sequence>
    <invoke service="RaiseWindowBlindService"/>
</sequence>
```

The second experiment is the same as the first one, but the user's location changed to the bedroom. That is, from the service agent's point of view, the service environments changed. After knowing the change of the user's current location to "MyBedroom" through the "GetUserLocationService," the SAP discovers an appropriate URS "TurnOnLampService" for the bedroom by using the location-aware semantic service discovery algorithm. The "TurnOnLampService" has an instance of "LampOn" concept, a sub-concept of "BecomeBrighter," as its effect. And the following service plan is automatically generated and executed for the user in the bedroom.

```
<sequence>
    <invoke service="TurnOnLampService"/>
</sequence>
```

Figure 10 Snapshots from the experiments.

For the next experiment, the user moves to the kitchen and commands the SA to "Make it brighter" again. At first, the SAP gets the current user's location with the "getUserLocation" service and tries to discover required URS's with the "RaiseBrightnessServiceModel" as in the previous experiments. But the URS discovery fails at this time because there are no URS's that have no precondition and a "BecomeBrighter" effect for the kitchen. Then, the SAP continues to discover URS's with another domain service model "RaiseBrightnessRobotServiceModel" and successfully discover appropriate URS's for the kitchen, that is, "MoveToKitchenService" and "TurnOnLightService" listed in table 1. And the following service plan is generated.

```
<sequence>
    <invoke service="MoveToKitchenService"/>
    <invoke service="TurnOnLightService"/>
</sequence>
```

As a result of the third plan execution, the mobile service robot successfully moved to the kitchen and turned on the lighting in the kitchen with the IR remote controller equipped on it as shown in figure 11. The last experiment is to control indoor climate. The user commands the SA to "Make it cooler" in the kitchen, and the SA generates the following OWL-S service request profile that has "BecomeCooler_Effect," an instance of "BecomeCooler" concept, as its effect.

```
<profile:Profile rdf:ID="ServiceRequest">
    <profile:hasEffect rdf:resource="BecomeCooler_Effect"/>
</profile:Profile>
```

After knowing the current location of the user "MyKitchen" by invoking "getUserLocation" service, the SAP successfully discovers "LowerTemperatureServiceModel" a domain service model that has an instance of "BecomeCooler" concept as its effect (refer to table 2). And the SAP continues to discover required URS's for the "LowerTemperatureServiceModel." Even though there are no URS with "BecomeCooler" effect for the kitchen as shown in table 1, the SAP can discover an alternative URS "TurnOnAirConditionerService" that has an instance of "AirConditionerOn" (a sub-concept of the "BecomeCooler") as its effect, and has an instance of "IndoorLocation" (a super-concept of the "Kitchen") as its effective location with the location-aware semantic service discovery algorithm. Finally, the following service plan is automatically generated and executed, and the air conditioner is turned on through the "turnOnAirConditioner" service as shown in the last photo of figure 11.

```
<sequence>
    <invoke service="TurnOnAirConditionerService"/>
</sequence>
```

Figure 11 Snapshots from the experiments.

6 Conclusions

In this paper, an architecture of ubiquitous service infrastructure based on Semantic Web Services is proposed and its working prototype system is demonstrated in our ubiquitous service test bed. The proposed service infrastructure enables service agents to automatically integrate feasible service devices, wireless sensors and other ubiquitous computing resources considering the user's current location by the use of Semantic Web Services technology.

Currently, the proposed ubiquitous service infrastructure has some limitations in supporting the robustness, scalability and security of ubiquitous service systems. Thus, works to overcome those limitations are in progress. At first, for the robustness of service systems, the proposed ubiquitous service infrastructure will incorporate execution monitoring and real-time execution functionalities for URS's. The execution monitoring module monitors whether the execution of each URS composing a service plan is properly completed or fails by checking the described effect of each URS is actually effective after the execution of it. For example of the "turnOnLamp" service, the execution monitoring module will check the brightness of the bedroom after the execution of the service through a brightness sensor deployed in the bedroom. When a service plan is not properly completed for some reasons, the plan generation module will generate an alternative service plan using other URS's except the failed URS. The prototype system used for the experiments does not incorporate any QoS (Quality of Service) mechanism for Web Services execution at a framework level because real-time support for URS's does not matter in the LAN-based experimental environments. But it is essential to ensure real-time invocations of Web Services with QoS mechanism in actual service environments including WAN (Wide Area Network) which often suffer from unpredictable communication delay (e.g., accessing a weather forecasting Web Service through the public Internet).

The second work in progress is to overcome the scalability problem of ubiquitous service systems. In the proposed ubiquitous service infrastructure, required service knowledge is discovered from the SKR, a centralized knowledge base server that is assumed to be well-known and always available. However, real ubiquitous service environments will be mostly based on ad hoc networks that are completely distributed and dynamic in nature. And they will surely consist of a very large number of ubiquitous sensors and service devices. This means that the centralized discovery approach is likely to suffer from a serious scalability problem in actual service environments. So, this work includes how to effectively distribute the knowledge to ad hoc sensors and devices in the service environments, and how to discover and plan with such distributed knowledge. It also includes how to cooperate and share the knowledge between multiple service systems under the same or relevant service environments (e.g., between a service agent in home and a service agent in office).

The final work is to support the security and privacy of ubiquitous service systems. Security and privacy issues of the proposed infrastructure arise owing to the use of the Web technologies and protocols as the communication method between the major components, which are open standards (e.g., HTTP, SOAP and XML). Basically, such issues are being approached from the XML encryption [39] and digital signature [40] techniques based on cryptography algorithms and the PKI (Public-Key Infrastructure) [15].

Acknowledgements

This work was supported by the faculty research fund of Konkuk University in 2008.

References

1. Axis Development Team, Axis Project Homepage, http://ws.apache.org/axis/.
2. Berners-Lee, T., Hendler, J. and Lassila, O., The Semantic Web. Scientific American, 2001, http://www.sciam.com/article.cfm?id=the-semantic-web&catID=2.
3. Botts, M., Percivall, G., Reed, C. and Davidson, J., OGC Sensor Web Enablement: Overview and High Level Architecture. OGC White Paper, Dec. 2007.
4. Chen, H., Finin, T. and Joshi, A., Ontologies for Agent Systems. Birkhauser Publishing, 2004.
5. Choi, K.H., Shin, H. J. and Shin, D. R., Service Discovery Supporting Open Scalability Using FIPA-Compliant Agent Platform for Ubiquitous Networks. Lecture Notes in Computer Science, LNCS 3482. 2005, 99-108.
6. CMU, myCampus Project Homepage, http://www.cs.cmu.edu/~sadeh/mycampus.htm.
7. DAML.org, OWL-S 1.0 Release. 2004, http://www.daml.org/services/owl-s/1.0/.
8. Erol, K., Nau, D. and Hendler, J., UMCP: a sound and complete planning procedure for HTN planning. in Proceedings of 2nd International Conferences on AI Planning Systems (AIPS-94), (1994), 249-254.
9. Gross, T., Egla, T. and Marquardt, N., Sens-ation: a service-oriented platform for developing sensor-based infrastructures. Int'l Journal of Internet Protocol Technology, 1 (3). 2006, 159-167.
10. Gu, T., Pung, H. K. and Zhang, D. Q., Toward an OSGi-based infrastructure for context-aware applications. IEEE Pervasive Computing, Dec. 2004, 66-74.
11. HP, Jena Project Homepage, http://jena.sourceforge.net/.
12. IBM, BPEL4WS (Business Process Execution Language for Web Services) version 1.1. 2003, http://www-128.ibm.com/developerworks/library/specification/ws-bpel/.
13. IBM, BPWS4J (BPEL 4 Java) Homepage, http://www.alphaworks.ibm.com/tech/bpws4j/.
14. Ibrahim, A. and Zhao, L., Supporting the OSGi Service Platform with Mobility and Service Distribution in Ubiquitous Home Environments. The Computer Journal (Oxford Univ. Press). 2008.
15. IETF, Public-Key Infrastructure (X.509). 2008, http://www.ietf.org/html.charters/pkix-charter.html.
16. Jini.org, Jini Homepage, http://www.jini.org/.
17. Kim J. H., Lee, W. I., Munson, J. and Tak, Y. J., Services-Oriented Computing in a Ubiquitous Computing Platform. Lecture Notes in Computer Science, LNCS 4294. 2006, 601-612.
18. Lee, K. M., Kim, H. J., Shin, H. J. and Shin, D. R., Design and Implementation of Middleware for Context-Aware Service Discovery in Ubiquitous Computing Environments. Lecture Notes in Computer Science, LNCS 3983. 2006, 483-490.
19. McIlraith, S., Son, T. C. and Zeng, H., The Semantic Web Services. IEEE Intelligent Systems, 16 (2). 2001, 46-53.
20. Miller, L., Seaborne, A. and Reggiori, A., Three implementations of SquishQL, a Simple RDF Query Language. in Proceedings of 1st International Semantic Web Conferences (ISWC 2002), (2002), 423-435.
21. MS Research, EasyLiving Project Homepage, 2001, http://research.microsoft.com/easyliving/.
22. Narayanan, S. and McIlraith, S., Simulation, verification and automated composition of Web services. in Proceedings of International WWW Conferences (WWW-11), (2002), 77-88.
23. Nau, D., Au, T., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D. and Yaman, F., SHOP2: an HTN planning system. Journal of Artificial Intelligence Research, 20, 2003, 379-404.

24. Nidd, M. Service discovery in DEAPspace. IEEE Personal Communications, 8(4), 2001, 39-45.
25. OpenSLP.org, OpenSLP Project Homepage, http://www.openslp.org/.
26. OSGi Alliance, OSGi Service Platform Core Specification Release 4. OSGi Specification, Apr. 2007.
27. Sirin, E., Parsia, B., Wu, D., Hendler, J. and Nau, D., HTN planning for Web service composition using SHOP2. Web Semantics, 1 (4), http://www.mindswap.org/papers/SHOP-JWS.pdf.
28. Soldatos, J., Pandis, I., Stamatis, K., Polymenakos, L. and Crowley, J., Agent based middleware infrastructure for autonomous context-aware ubiquitous computing services. Computer Communications, 30. 2007, 577-591.
29. UPnP Forum, UPnP Project Homepage, http://www.upnp.org/.
30. W3C Web Services Activity, SOAP Version 1.2 Part 0: Primer. 2007, http://www.w3c.org/TR/soap12-part0/.
31. W3C Web Services Activity, Web Services Architecture. 2004, http://www.w3c.org/TR/ws-arch/.
32. W3C Web Services Activity, Web Services Description Language (WSDL) Version 2.0 Primer. 2007, http://www.w3.org/TR/wsdl20-primer/.
33. W3C Semantic Web Activity, RDF Primer. 2004, http://www.w3c.org/TR/rdf-primer/.
34. W3C Semantic Web Activity, RDF Semantics. 2004, http://www.w3c.org/TR/rdf-mt/.
35. W3C Semantic Web Activity, RDF Vocabulary Description Language 1.0: RDF Schema. 2004, http://www.w3c.org/TR/rdf-schema/.
36. W3C Semantic Web Activity, OWL Web Ontology Language Guide. 2004, http://www.w3c.org/TR/owl-guide/.
37. W3C Semantic Web Activity, OWL Web Ontology Language Semantics and Abstract Syntax. 2004, http://www.w3c.org/TR/owl-absyn/.
38. W3C XML Activity, Extensible Markup Language (XML). 2008, http://www.w3.org/XML/.
39. W3C XML Security WG, XML Encryption Syntax and Processing. 2002, http://www.w3.org/TR/xmlenc-core/.
40. W3C XML Security WG, XML-Signature Syntax and Processing. 2002, http://www.w3.org/TR/xmldsig-core/.
41. Wang, X. H., Zhang, D. Q., Gu, T. and Pung, H. K., Ontology Based Context Modeling and Reasoning using OWL. in Proceedings of 2nd IEEE Annual Conference on Pervasive Computing and Communications Workshops (PERCOMW'04), (2004), 18-22.
42. Weiser, M., The Computer for the Twenty-First Century. Scientific American, 1991, 94-100.
43. Xiaohu, G. and Guangxi, Z., Empowering Ubiquitous Services in Next-generation Smart Home. Information Technology Journal, 5 (1). 2006, 64-69.