Available online at www.sciencedirect.com
ScienceDirect
Information Sciences 177 (2007) 657–679
INFORMATION SCIENCES
AN INTERNATIONAL JOURNAL
www.elsevier.com/locate/ins

A robotic service framework supporting automated integration of ubiquitous sensors and devices

Young-Guk Ha [a,*], Joo-Chan Sohn [b], Young-Jo Cho [b], Hyunsoo Yoon [a]

[a] *Computer Science Division, Korea Advanced Institute of Science and Technology, 373-1 Guseong-dong, Yuseong-gu, Daejeon 305-701, Republic of Korea*

[b] *Intelligent Robot Research Division, Electronics and Telecommunications Research Institute, 161 Gajeong-dong, Yuseong-gu, Daejeon 305-700, Republic of Korea*

Received 6 July 2005; received in revised form 23 May 2006; accepted 5 July 2006

Abstract

In recent years, due to the emergence of ubiquitous computing technology, a new class of networked robots called ubiquitous robots has been introduced. The Ubiquitous Robotic Companion (URC) is our conceptual vision of ubiquitous service robots that provides its user with the services the user needs, anytime and anywhere, in the ubiquitous computing environments. There are requirements to be met for the vision of URC. One of the essential requirements is that the robotic systems must support ubiquity of services. This means that a robot service must always be available even though there are changes in the service environment. More specifically, a robotic system needs to be interoperable with sensors and devices in its current service environments automatically, rather than statically pre-programmed for its environment. In this paper, the design and implementation of an infrastructure for URC called Ubiquitous Robotic Service Framework (URSF) is presented. URSF enables automated integration of networked robots in a ubiquitous computing environment by the use of Semantic Web Services Technologies.

Keywords: Ubiquitous robotics; Ubiquitous computing; Networked robotics; Service robotics; Service planning; Semantic Web services

1. Introduction

Because of recent innovations in communication network technology, there has been great interest in researching Internet-based networked robotic systems [31]. Most of these systems are focused on a human-supervised tele-operation or monitoring of networked robotic devices (for example, mobile robots, unmanned vehicles, vision sensors, and the like) in the Internet environment. Several attempts have been made to develop such Internet-based networked robotic systems using the World Wide Web and distributed object technologies such as Common Object Request Broker Architecture (CORBA) and Java. Leveraging the advantages of Internet technology, such systems allow users from all over the world to visit museums, tend gardens, navigate

* Corresponding author. Tel.: +82 42 860 6375; fax: +82 42 860 6790.
E-mail address: ygha@etri.re.kr (Y.-G. Ha).

0020-0255/$ - see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.ins.2006.07.003

undersea or float in blimps 24 hours a day. They have great potential for industry, education, entertainment and security by making valuable robotic hardware accessible to a broad audience.

In recent years, motivated by the emergence of ubiquitous computing [32] technology as the next generation computing paradigm, a new class of networked robots called ubiquitous robots has been introduced [7,8]. They are actually networked robots integrated into ubiquitous computing environments (including ubiquitous sensors and devices). They can even be implemented as a form of ubiquitous computing environment themselves as demonstrated in the robotic room [15]. The Ubiquitous Robotic Companion (URC) [14] is our conceptual vision for bringing these ubiquitous robots to users and providing them with the services they need anytime, anywhere in the ubiquitous computing environments. There are requirements to be met for the vision of URC. One of the essential requirements is that the robotic systems must support ubiquity of services. This means that a robot service must be always available even though there are changes in the service environment.

However, the current networked robotics approach is mainly focused on a behavior-oriented tele-operation of remote robotic devices with Web applications or distributed objects programmed for specific environments as shown in Fig. 1(a). Surely, this helps people to overcome temporal and spatial limitations for providing services to some extent. Nevertheless, to provide ubiquitous services, robotic systems need to be automatically interoperable with ubiquitous sensors and devices in the current service environments rather than being statically pre-programmed for the environments. For example, they should be automatically interoperable with consumer electronics, embedded actuators, wireless sensors and other similar devices.

In this paper, the design and implementation of infrastructure for URC called the Ubiquitous Robotic Service Framework (URSF) is presented. It enables automated integration of networked robots in ubiquitous computing environments in a service-oriented way. The URSF utilizes Semantic Web Services [9], which are a state of the art Web technology, and an Artificial Intelligence (AI) planning methodology. These technologies are used to provide automated interoperation between various networked robots and ubiquitous computing devices in the service environment. As shown in Fig. 1(b), Web services [41] for robots, networked sensors and devices are implemented as a unified interface method for accessing them. Then, knowledge about such Web services is described in Web Ontology Language for Services (OWL-S) [29], the semantic description

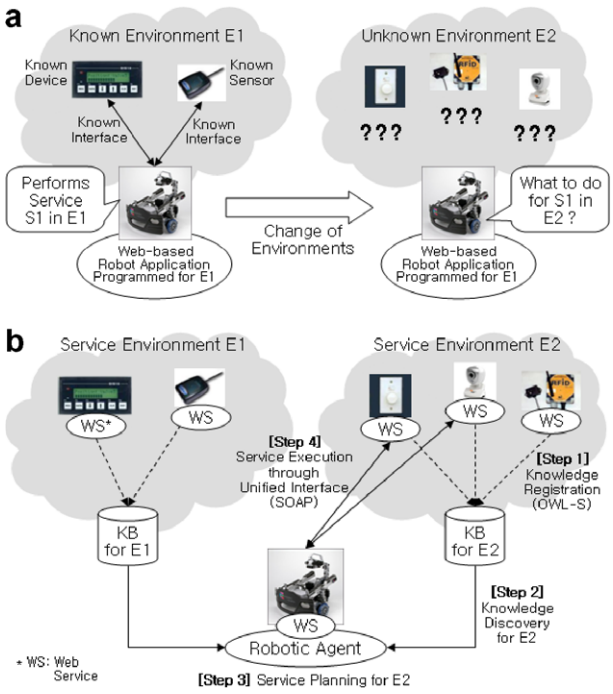

Fig. 1. Traditional networked robot system and the URSF approach. (a) Traditional networked robotic system for specific environments E1. (b) Automatic interoperability with changed service environments based on knowledge about sensors and devices in the environments.

language for Web services. At this point, this knowledge is registered to environmental Knowledge Bases (KB) so that a robotic agent can automatically find the required knowledge and compose a feasible service execution plan for the current environments. Finally, the agent provides the service by automatically interacting with robots, sensors and devices through the use of the Simple Object Access Protocol (SOAP) [40], which is a Web service execution protocol, according to the service plan.

This paper is organized as follows. Section 2 briefly introduces existing networked robotics approaches and the Semantic Web Services technologies as the fundamental background for the paper. Section 3 describes the detailed design of the URSF and its interoperability with standard service frameworks. Section 4 explains the prototype implementation, experiments and performance evaluation of the URSF in our networked home test bed. Finally, Section 5 discusses conclusions, known limitations and future work.

2. Background

2.1. Networked robotics technologies

A "networked robot" [31] is a robotic device connected to a communications network such as the Internet or Local Area Network (LAN). The network could be wired or wireless and based on any of a variety of protocols such as Hypertext Transfer Protocol (HTTP), Transmission Control Protocol (TCP), User Datagram Protocol (UDP) or IEEE 802.11 wireless LAN. Many new applications are now being developed ranging from automation to exploration. There are two subclasses of networked robots: tele-operated, where human supervisors send commands and receive feedback via the network; and autonomous, where robots and sensors exchange data via the network to perform a task. In such systems, the sensor network extends the effective sensing range of the robots, allowing them to communicate with each other over long distances to coordinate their activity.

The existing approach to networked robotics is mainly focused on a behavior-oriented tele-operation with Web applications or distributed objects programmed for known operational environments such as known objects, sensors and actuators. Typical Web-based networked robotic systems use HTTP combined with Common Gateway Interface (CGI) or Java applets to control remote sensors and actuators. Some examples of these are the University of Southern California's tele-excavation system, Mercury [3]; Carnegie Mellon University's indoor mobile robot, Xavier [17]; Ecole Polytechnique Fédérale de Lausanne's maze robot, KhepOnTheWeb [16]; and Roger Williams University's PumaPaint [19]. Another approach to networked robotic systems is based on distributed object technology such as CORBA and Java Remote Method Invocation (RMI). Some examples are the Network Robot Service Platform (NRSP) [20] and the Distributed Architecture for Internet Robot (DAIR) [4].

In recent years, a new kind of networked robotics project has been undertaken in the Japanese National Institute of Advanced Industrial Science and Technology (AIST) incorporating the concept of ubiquitous computing. The goal of the project is to develop a knowledge distributed robot system [6], based on a Radio Frequency Identification (RFID) technology, which can automatically integrate objects in service environments. In the knowledge distributed robot system, every object in service environments has a RFID tag which stores the address of a Web page containing the object information encoded in eXtensible Markup Language (XML). A robot can read addresses for object information Web pages from RFID tags and automatically handle the environmental objects by obtaining required information about the objects from the Web pages. However, this project is only focused on physical integration of the environment, that is, identification and physical handling of objects in service environments. So, the object information in the knowledge distributed robot system mainly includes physical feature data of the objects such as object's name, size, position, weight, and so on. In addition, there is no consideration about context-dependent services.

2.2. Semantic Web services

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 airline reservation services and e-commerce applications. Web-accessible programs and databases implement these

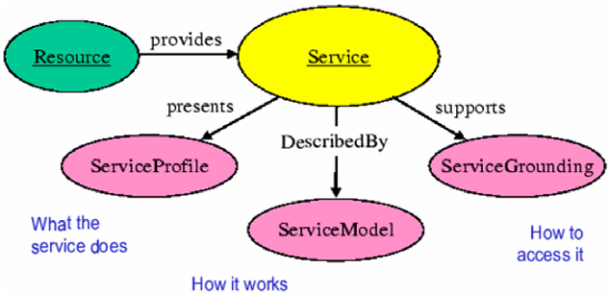

Fig. 2. Top level of OWL-S service ontology.

services through the use of CGI, Java, ActiveX or the Web services [40,41] technology. 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 [1] of services whose semantics, such as properties, capabilities and interfaces, are encoded in an unambiguous, machine-understandable form that are based on Resource Description Framework (RDF) [37] or Web Ontology Language (OWL) [35]. The Semantic Web Services technology is developed to meet this need by describing Web services with OWL ontology, namely OWL-S [29], which provides AI-inspired markups (classes and properties) for specifying a richer-level of service semantics. As shown in Fig. 2, OWL-S markups are grouped into three essential parts for describing the service profile, the service model and the service grounding.

The OWL-S service ontology can be encoded with XML and the following example shows the top level of OWL-S service ontology in XML. Note that in this paper, most of the example data is encoded with XML. And the semantics of each vocabulary used in the examples is based on the formal semantics of RDF [37,38], RDF Schema (RDFS) [38,39], OWL [35,36] and OWL-S (service, profile, process and grounding ontology) [11,29] as it is prefixed by a namespace.

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

2.2.1. Service profile

The service profile tells "what the service does," that is, it gives the type of information needed by a service requester 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 requester agent so that a matchmaker has a convenient dual-purpose representation upon which to base its operations. The OWL-S profile ontology provides useful markups to describe a service profile: Profile, has_process, serviceName, textDescription, serviceCategory, hasInput, hasOutput, hasPrecondition, hasEffect, and so on.

The following example shows a profile of an airline reservation service named "Profile_Airline_ReservationService," which is described with the markups for specifying Input, Output, Precondition and Effects (IOPE) of the service.

```
<profile:Profile rdf:ID="Profile_Airline_ReservationService">
  <profile:serviceName>Airline_ReservationService</profile:serviceName>
  <profile:hasInput rdf:resource="#DepartureAirport_Input"/>
  <profile:hasInput rdf:resource="#ArrivalAirport_Input"/>
  <profile:hasInput rdf:resource="#OutboundDate_Input"/>
  <profile:hasOutput rdf:resource="#ReservationID_Output"/>
  <profile:hasPrecondition rdf:resource="#SeatExists_Precondition"/>
```

```
    <profile:hasEffect rdf:resource="#HaveSeat_Effect"/>
  </profile:Profile>
```

2.2.2. Service model

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 requester 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. The OWL-S process ontology provides the following markups to model services as processes (atomic or composite processes): ProcessModel, hasProcess, AtomicProcess, CompositeProcess, composedOf, components, Sequence, Choice, Split, Split-Join, Repeat-Until, and so on.

The next example shows "Airline_Reservation_ProcessModel," a composite process model for the above airline reservation service, which is described as a sequence of three component processes "GetDesiredFlightDetails," "SelectAvailableFlight" and "BookFlight."

```
  <process:ProcessModel rdf:ID="Airline_Reservation_ProcessModel">
    <process:hasProcess>
      <process:CompositeProcess rdf:ID="Airline_Reservation_Process">
        <process:composedOf>
          <process:Sequence>
            <process:components rdf:parseType="Collection">
              <process:AtomicProcess rdf:about="#GetDesiredFlightDetails"/>
              <process:AtomicProcess rdf:about="#SelectAvailableFlight"/>
              <process:CompositeProcess rdf:about="#BookFlight"/>
            </process:components>
          </process:Sequence>
        </process:composedOf>
      </process:CompositeProcess>
    </process:hasProcess>
  </process:ProcessModel>
```

2.2.3. Service grounding

The service grounding specifies the details of how an agent can access a service. Typically, a service grounding will specify concrete 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. The OWL-S grounding ontology provides the following markups to ground OWL-S atomic processes with concrete Web services implementations whose interfaces are described in the Web Services Description Language (WSDL) [42]: WsdlAtomicProcessGrounding, owlsProcess, wsdlOperation, portType, operation, wsdlInputMessage, wsdlInputs, WsdlInputMessageMap, owlsParameter, wsdlMessagePart, WsdlOperationRef, and so on.

The next example shows part of a service grounding for the OWL-S atomic process "GetDesiredFlightDetails" in the previous example. The example describes that the process itself is mapped into a Web service operation "GetDesiredFlightDetails_operation" and its input "DepartureAirport_Input" is mapped into an input message "departureAirport" of the concrete Web service http://www.bravoair.com/services/BravoAirReservationService.

```
  <grounding:WsdlGrounding rdf:ID="Grounding_GetDesiredFlightDetails">
        ...
    <grounding:wsdlDocument>
        http://www.bravoair.com/services/BravoAirReservationService?wsdl
    </grounding:wsdlDocument>
```

```
<grounding:wsdlOperation>
  <grounding:WsdlOperationRef>
    <grounding:operation>GetDesiredFlightDetails_operation</grounding:operation>
  </grounding:WsdlOperationRef>
</grounding:wsdlOperation>

<grounding:WsdlInputMessageMap>
  <grounding:owlsParameter rdf:resource="#DepartureAirport_Input"/>
  <grounding:wsdlMessagePart>departureAirport</grounding:wsdlMessagePart>
</grounding:WsdlInputMessageMap>
        ...
</grounding:WsdlGrounding>
```

3. Design of the URSF

3.1. The architecture of the URSF

As shown in Fig. 3, the architecture of URSF consists of three major components, a Robotic Agent (RA), Device Web Services (DWS) and an Environmental Knowledge Repository (EKR). The RA, as a service requester agent, plays the main role in the automated integration procedure of the URSF. Each DWS is a concrete implementation of Web services for ubiquitous devices in service environments and the EKR is used for registration and discovery of knowledge about the service environments including OWL-S descriptions for each DWS. The EKR also includes OWL ontology of general concepts necessary to describe the knowledge. The following subsections will explain details of each URSF component based on its activities.

3.1.1. Robotic agent (RA)

The RA consists of a service application, a URSF Application Programming Interface (API), a plan composition module, a knowledge discovery module, a plan execution module, an OWL reasoner and a protocol stack for Web services execution including SOAP, XML and HTTP.

To request a robot for a service, a user can input a command with a user interface for the service application. Then, the service command is encoded with vocabularies in OWL-S profile ontology and the concept ontology stored in the EKR so that the knowledge discovery module and the plan composition module can understand the user's service request. For instance, suppose that a user inputs a command "Come here" with the user interface, which tells a robot to move to the location where the user is currently positioned. It is encoded into the following OWL-S service profile to explicitly represent semantics of the user's request. The profile has "UserLocation_Input" an instance of "UserLocation" concept as an input and "AtLo-

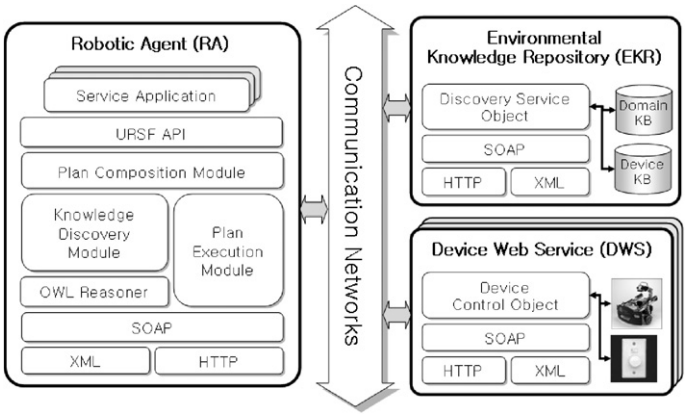

Fig. 3. Detailed architecture of URSF.

cation_Effect'' an instance of ''AtLocation'' concept as an effect. Note that a concept class prefixed by ''concepts'' namespace is defined in the concept ontology.

```
<!--Instance declarations for the concept classes-->
<concepts:UserLocation rdf:ID="UserLocation_Input"/>
<concepts:AtLocation rdf:ID="AtLocation_Effect"/>

<profile:Profile rdf:ID="Profile_ServiceRequest">
  <profile:hasInput rdf:resource="#UserLocation_Input"/>
  <profile:hasEffect rdf:resource="#AtLocation_Effect"/>
</profile:Profile>
```

To search KB's in the EKR for required knowledge in order to compose a service plan, the RA creates discovery queries encoded in RDF Data Query Language (RDQL) [10] by inferring the semantics of the request profile with the OWL reasoner. For instance, the RA can recognize that it needs to know the user's location as an input value for the service by inferring from the above request profile. The user's location is one of the essential service contexts as well. So, the knowledge discovery module of the RA needs to discover a user location provider service using the following RDQL query, which tells the EKR to find a service described by a process model having a process that outputs an instance of ''UserLocation'' concept.

```
SELECT   ?service
WHERE    (?service, service:describedBy, ?processModel)
         (?processModel, process:hasProcess, ?process)
         (?process, process:hasOutput, ?output)
         (?output, rdf:type, concepts:UserLocation)
```

As the result of querying, the knowledge discovery module can receive OWL-S knowledge about a user location provider service that has an instance of ''UserLocation'' concept as its output. By invoking the discovered service based on its service grounding, the RA can gather the required context data. Then by executing the semantic service discovery algorithm (refer to Section 3.2 for details) with the request profile, the knowledge discovery module can obtain required knowledge for planning from the EKR. This includes OWL-S description about ''MoveToService'' which is a feasible robot navigation service. As shown in the next example, the discovered knowledge includes ''MoveToServiceProcessModel,'' an atomic process model for the service, which has ''Location_Input'' an instance of ''Location'' concept (a super-concept of ''UserLocation'' of the request profile) as its input, and an instance of ''AtLocation'' concept as its effect. It also includes ''MoveToServiceGrounding,'' a grounding for the discovered service, which will be used in the plan execution module.

```
<service:Service rdf:ID="MoveToService">
  <service:described By rdf:resource="#MoveToServiceProcessModel"/>
  <service:supports rdf:resource="#MoveToServiceGrounding"/>
</service:Service>

<process:ProcessModel rdf:ID="MoveToServiceProcessModel">
  <process:hasProcess>
    <process:AtomicProcess rdf:ID="MoveToServiceProcess">
      <process:hasInput rdf:resource="#Location_Input"/>
      <process:hasEffect rdf:resource="#AtLocation_Effect"/>
    </process:AtomicProcess>
  </process:hasProcess>
</process:AtomicProcess>

<grounding:WsdlGrounding rdf:ID="MoveToServiceGrounding">
```

... : **Details of the grounding are omitted.**
</grounding:WsdlGrounding>

After the gathering of context data and the discovery of all the required knowledge, the RA starts to generate a service plan using an AI-based planning methodology (refer to Section 3.3 for details). The following example in XML shows a service plan generated by the plan composition module using the context data and the discovered service knowledge. The plan is simply composed of a single robot control service that navigates the robot to the user's current location "LivingRoom." However it is not an executable plan. The plan execution module translates the service plan into an executable format using a service grounding for each service and finally executes the executable plan through the Web service protocol stack (refer to Section 3.4 for details).

```
<plan>
  <service name="MoveToServiceProcess">
    <input name="Location_Input" value="LivingRoom"/>
  </service>
</plan>
```

3.1.2. Device Web service (DWS)

As briefly introduced in the beginning of this section, the DWS is actually a concrete implementation of Web services for ubiquitous devices including mobile robots, sensors, actuators, digital appliances, and so on (i.e. implementation of a robot navigation service and user location sensing service). Each DWS has a device control object and a WSDL description to represent how to bind to the control object with SOAP Remote Procedure Call (RPC) [40]. The control object can be associated with either one device or multiple devices that work cooperatively. For instance, a mobile robot or a group of RFID readers to sense where the user is currently could be used (refer to Section 4 for details). And the DWS also includes a Web service protocol stack to communicate with the RA.

3.1.3. Environmental Knowledge Repository (EKR)

The EKR is a permanent storage of OWL-S knowledge necessary to compose a feasible service plan for specific service environments. The EKR contains two kinds of knowledge bases: a domain KB and a device KB. The domain KB stores OWL-S descriptions of composite process models including control flows of component processes. Each description represents a common service model for a certain domain called the domain service model. A typical example of the domain service model was given as "Airline_Reservation_ProcessModel" in Section 2.2. The domain KB also stores the OWL ontology of general concepts to describe the IOPE of services: for instance, the "AtLocation", "UserLocation" and "LivingRoom" concepts in the previous and forthcoming examples. The device KB stores OWL-S descriptions of atomic services with corresponding groundings. Each description represents a DWS for specific service environments. An example of DWS knowledge is given as "MoveToService" in the above robotic agent subsection. More running examples of the service knowledge bases will be presented in Section 4.

In addition to the knowledge bases, the EKR includes the discovery service object to handle RDQL queries from the knowledge discovery module of the RA. The EKR also includes a Web service protocol stack because it works as a Web service itself. That is, the RA can access the EKR with SOAP (the unified interface protocol in our framework).

Fig. 4 summarizes the automated integration procedure of URSF, which is divided into the context data gathering and the service composition, and the activity of each component during the procedure (for reference, the dashed arrows in the figure mean SOAP RPC messaging). As illustrated in the figure, the automated integration procedure starts with passing the service request profile to the plan composition module by calling the URSF API. Especially, the service composition is broken down into three phases: knowledge discovery phase, service plan composition phase and plan execution phase. The following sections will explain each service composition phase in more detail using specific data examples.

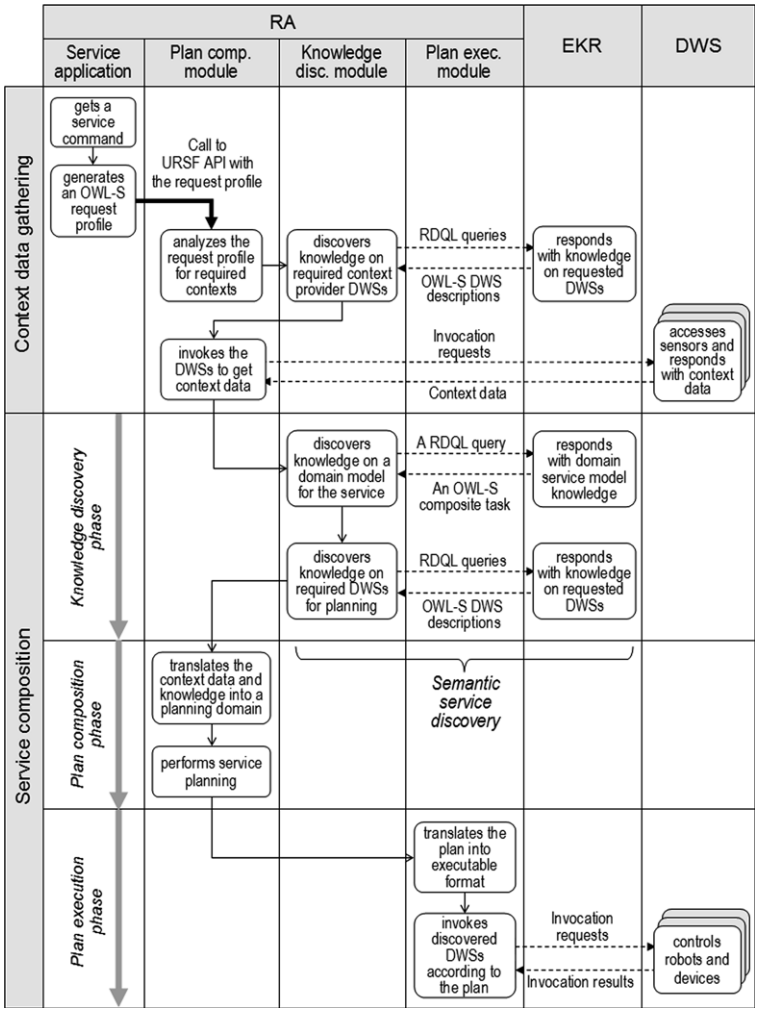

Fig. 4. Automated integration procedure of URSF.

3.2. Knowledge discovery phase

As mentioned above, knowledge about domain service models and device Web services are described with OWL-S so that the RA can automatically discover the required knowledge to compose a service plan for a user request. Knowledge about device Web services, which is described with OWL-S atomic processes [29] and corresponding service groundings, is used to generate the space of feasible actions in specific service environments during the plan composition phase. At this point, knowledge about domain service models, which is described with OWL-S composite processes [29], is used to constrain how the service plans are to be composed independently of specific service environments.

As shown in Fig. 4, the knowledge discovery is actually performed by sending discovery queries to the EKR and receiving responses through SOAP. 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 device Web services is found from the device KB. In the knowledge discovery module, we implement the semantic service discovery algorithm to compose a feasible plan for the requested service. Prior to explaining the details of the semantic service discovery algorithm, we will introduce the existing service discovery approach and discuss its limitation in ubiquitous service environments.

3.2.1. Existing service discovery approach

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 [25], 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 Universal Plug and Play (UPnP) [33] uses a syntactic service attribute matching, based on service descriptions in XML. And most of the existing service discovery technologies such as Service Location Protocol (SLP) [27], Salutation [30] and DEAPspace [13] also use a similar syntactic level matching algorithm.

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 EKR have knowledge of a robot navigation service that has a ''Location'' as its input argument and makes the robot move to the given location. If a robotic agent tries to discover a robot navigation service that has a ''UserLocation'' as its input argument using a syntactic level matching algorithm, it will fail because there is no exact match for the requested input argument. So, we need a more expressive method to describe services and to discover services in a semantic manner for ubiquitous service environments.

3.2.2. Semantic service discovery algorithm

The semantic service discovery algorithm of URSF can overcome the limitations of the existing service discovery approach by matching with a semantically compatible (replaceable) service for every device Web service 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 input argument which is equivalent to or a super-class of ''UserLocation'' by reasoning about the concept ontology.

As formerly explained, knowledge about domain service models and device Web services are described with OWL-S composite and atomic processes, which have OWL individuals [35] as their IOPE property [29] values. For instance, an atomic process ''MoveToServiceProcess'' in the example of Section 3.1 has an OWL individual ''Location_Input'' (an instance of ''Location'' concept) as a value of its ''hasInput'' property. Therefore, we defined some semantic relations between OWL individuals and OWL-S processes based on a predicate logic to describe the algorithm formally: that is, $p(s,o)$, where p is a predicate, s is a subject of p, and 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 a part of the concept ontology and two atomic processes, ''MoveToServiceProcess'' in Section 3.1 and ''MoveToLocation'' in the ''RaiseIndoorBrightnessProcessModel'' example of Section 4.2, are given as follows:

//Predicate logic representation of the concept ontology.
rdfs:subClassOf(UserLocation, Location)
...
//Predicate logic representation of "MoveToServiceProcess."
process:hasInput(MoveToServiceProcess, Location_Input)
process:hasEffect(MoveToServiceProcess, AtLocation_Effect)
...
//Predicate logic representation of "MoveToLocation."
process:hasInput(MoveToLocation, UserLocation_Input)
process:hasEffect(MoveToLocation, AtLocation_Effect)
...

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

For instance, an individual subsumption relation "Location_Input ⊇ UserLocation_Input" holds for the above predicate logic representations because "Location_Input" is an instance of class "Location" and "UserLocation_Input" is an instance of class "UserLocation" defined as a subclass of "Location" on the concept ontology.

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

For instance, an individual subsumption relation on a property "MoveToServiceProcess ⊇ MoveToLocation/process:hasInput" holds for the above predicate logic representations because "MoveToServiceProcess" has "Location_Input" and "MoveToLocation" has "UserLocation_Input" for the value of each "hasInput" property, and "Location_Input ⊇ UserLocation_Input" holds. This means that "MoveToServiceProcess" accepts more general inputs than "MoveToLocation."

In addition, "MoveToLocation ⊇ MoveToServiceProcess/process:hasEffect" also holds and vice versa for the above predicate logic representations because "MoveToLocation" and "MoveToServiceProcess" both have an instance of the same class "AtLocation" for the value of each "hasEffect" property. This means that "MoveToLocation" produces the same effects as "MoveToServiceProcess."

Definition 3 (*Process compatibility relation* ($x \sim y$)). An OWL-S process x is compatible with an OWL-S process y iff $(x \supseteq y/\text{process:hasInput}) \wedge (x \supseteq y/\text{process:hasPrecondition}) \wedge (y \supseteq x/\text{process:hasOutput}) \wedge (y \supseteq x/\text{process:hasEffect})$.

For instance, a process compatibility relation "MoveToServiceProcess ∼ MoveToLocation" holds for the above predicate logic representations because "MoveToServiceProcess ⊇ MoveToLocation/process:hasInput" and "MoveToLocation ⊇ MoveToServiceProcess/process:hasEffect" hold. This means that "MoveToServiceProcess" can be used instead of "MoveToLocation" but not vice versa because "MoveToServiceProcess" accepts more general inputs than and produces the same effects as "MoveToLocation."

Definition 4 (*Process equivalence relation* ($x \equiv y$)). An OWL-S process or profile x is equivalent to an OWL-S process or profile y iff x and y have the same set of IOPE property values.

The semantic service discovery algorithm is given below. As shown in the pseudo codes, the input of the algorithm is a service request profile R, and the outputs of the algorithm are a discovered domain service model M and a set of discovered device Web services D.

```
Procedure Semantic_Service_Discovery (R, M, D)
  Input: a service request profile R;
  Outputs: a domain service model M, a set of device Web services D;
{
  D ← ∅;
  Discover a domain service model M from the domain KB s.t. composite process of M ≡ R;
  If (the discovery fails) exit with Discovery_Error;

  Call DWS_Discovery(M, D);
} // End of procedure Semantic_Service_Discovery

Procedure DWS_Discovery(M, D)
  Input: a domain service model M;
  Output: a set of device Web services D;
{
For each component process P in M {
  If (P is a composite process) call DWS_Discovery(P, D);

    Discover a device service S from the device KB s.t. atomic process of S ≡ P;
    If (the discovery successes) add S to D;
    Else {
```

```
        Discover a device service S from the device KB s.t. atomic process of S ~ P;
        If (the discovery successes) {
          Add S to D;
          Replace P in the service model M with atomic process of S;
        } Else, exit with Discovery_Error;
      }
    }
  } // End of procedure DWS_Discovery
```

To begin with, the semantic service discovery algorithm discovers a domain service model M that is equivalent to the service request profile R in accordance with the process equivalence relation (Definition 4). On success of the discovery, the algorithm proceeds to discover a device Web service that has an equivalent atomic process to each component atomic process of the domain service model M in accordance with the process equivalence relation (Definition 4). When a match is found, the algorithm adds the matched service to the set of device Web services D. On a failure, the algorithm tries to discover a device Web service that has semantically compatible atomic process for the failure in accordance with the process compatibility relation (Definition 3). If the discovery succeeds, the algorithm adds the discovered device Web service to the set of device Web services D and extends the domain service model M by replacing the corresponding component process with the atomic process of the discovered device Web service.

3.3. Plan composition phase

In the plan composition phase, a feasible service plan for the user request is automatically composed using Hierarchical Task Network (HTN) planning [2]. HTN planning is an AI planning methodology used 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 a planning domain. Such task decomposition concepts 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 domain and 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 device Web services D into the planning operators (note that M and D are outputs from the service discovery algorithm in the previous section).

We programmed such translation algorithms into the plan composition module of the RA. Because we used the University of Maryland's domain-independent HTN planning system, SHOP2 (Simple Hierarchical Ordered Planner) [12] for the implementation, generating a planning domain in terms of SHOP2 domain is required. Inheriting a generic HTN domain, each SHOP2 operator describes what needs to be done to accomplish some primitive task and each SHOP2 method tells how to decompose some compound task into partially ordered subtasks. 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 device Web service, 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. For instance, the following SHOP2 operator is generated from "MoveToServiceProcess" in the example of Section 3.1 by replacing p with "MoveToServiceProcess," v with "?Location_Input," and *Add* with "AtLocation_Effect." Note that *Pre* and *Del* are null because "MoveToServiceProcess" has no preconditions.

```
(:operator    (!MoveToServiceProcess ?Location_Input)
              () () ; No precondition and delete list
              (AtLocation_Effect)) ; Add list
```

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 $P_{1\leqslant i\leqslant n}$, 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 ($P_1 \ldots P_n$). For instance, in the second experiment in Section 4.3, "RaiseIndoorBrightnessProcessModel" is discovered as a domain service model, and "MoveToServiceProcess" and "TurnOnLightServiceProcess" are discovered as device Web services. So, the following SHOP2 method is generated during the plan composition phase of the second experiment by replacing c with "RaiseIndoorBrightnessProcess," v with "?Location_Input," and T with "(MoveToServiceProcess ?Location_Input) (TurnOnLightServiceProcess)," an ordered list of discovered device Web services. Note that Pre is null because "MoveToServiceProcess" has no preconditions.

```
(:method    (RaiseIndoorBrightnessProcess ?Location_Input)
            ( ) ; No precondition
            ((MoveToServiceProcess ?Location_Input)
            (TurnOnLightServiceProcess))) ; List of subtasks
```

Consequently, the SHOP2 planner generates a total ordered set of atomic processes for device Web services based on the planning operators and methods. The next example shows an input planning problem and output service plan of the SHOP2 planner for the second experiment in Section 4.3 where a user is in the kitchen and commands the robot to "Make it brighter." Note that the user's current location "Kitchen" is obtained during the context data gathering procedure and used to formulate the planning problem combined with the domain service model.

Planning problem:
(RaiseIndoorBrightnessProcess Kitchen)
Generated plan:
((MoveToServiceProcess Kitchen) (TurnOnLightServiceProcess))

Before the plan execution phase, the output of the SHOP2 planner is encoded with XML for the future interoperability with other service agents or applications. Examples of XML-encoded service plans are given in the end of Section 3.1 and the experiments of Section 4.3. For more references to SHOP2, see [12,18].

3.4. 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. In our implementation, Business Process Execution Language for Web Services (BPEL4WS) [22] 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 device Web services 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 second experiment in Section 4.3. It includes a sequence of two "invoke" statements of concrete Web services operations and messages: the first one is for the "moveTo" operation of the Web service "NavigationService" with an input message "MoveTo_Input" and the second one is for the "turnOnLight" operation of the Web service "LightControlService." Note that such information about concrete Web services comes from the relevant service groundings.

```
<process    xmlns: ws1="http://robot.etri.re.kr:8080/axis/NavigationService"
            xmlns:ws2="http://robot.etri.re.kr:8080/axis/LightControlService">
```

```
        ...
  <sequence>
        ...
    <invoke portType="ws1:NavigationService"
          operation="moveTo"
          inputVariable="ws1:MoveTo_Input" .../>
    <invoke portType="ws2: LightControlService"
          operation="turnOnLight" .../>
        ...
  </sequence>
</process>
```

Fig. 5 illustrates a data flow diagram for generating an executable BPEL4WS process from discovered OWL-S knowledge about a domain service model and device Web services during the service composition procedure. For reference, shaded 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. Note that at this time, we do not consider contingency planning and execution monitoring features of the framework during the service composition procedure. However, it is important to implement robust service systems and this will be one of our research and development issues for the next project.

3.5. URSF and existing service frameworks

This section will introduce how URSF applications can interoperate with networked devices and sensors deployed using other service frameworks. Currently, there exist many standard service frameworks for heterogeneous devices and sensors. Among these standards, Open Service Gateway Initiative (OSGi) and Open Geographic Information System (OpenGIS) are most effective and commonly accepted technologies. The OSGi Service Platform [28] provides a Java-based service-hosting environment for residential gateways as well as a set of common APIs to develop dynamically downloadable service bundles. It also provides several basic services such as configuration management, user management, device management and HTTP service. The OpenGIS SensorWeb [26] specifies interoperability interfaces and metadata encodings that enable integration of heterogeneous Web-connected sensors into information infrastructures based on XML and Web Services technologies. The SensorWeb specifications provide Sensor Model Language (SensorML), Sensor Observation Service (SOS) and Sensor Planning Service (SPS) over the OpenGIS Web Service (OWS) Framework.

Fig. 6 illustrates how URSF service applications can interoperate with devices and sensors deployed using OSGi Service Platform and OpenGIS SensorWeb. Note that OWL-S knowledge descriptions for all the

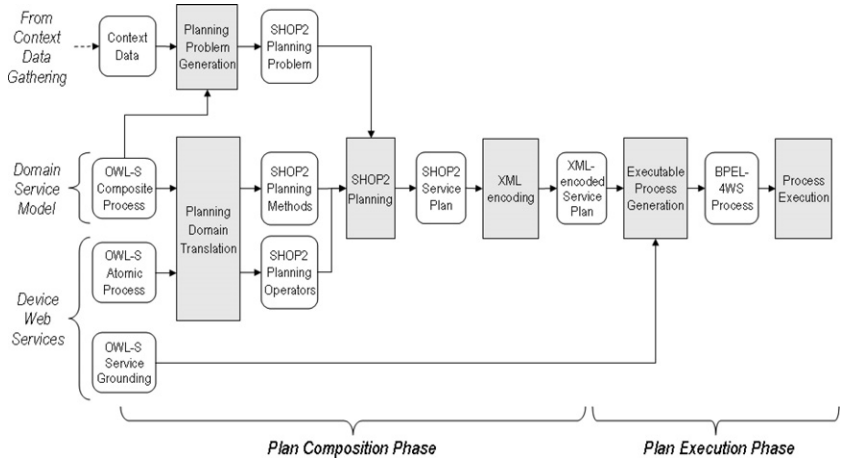

Fig. 5. Data flow diagram for generating a BPEL4WS process.

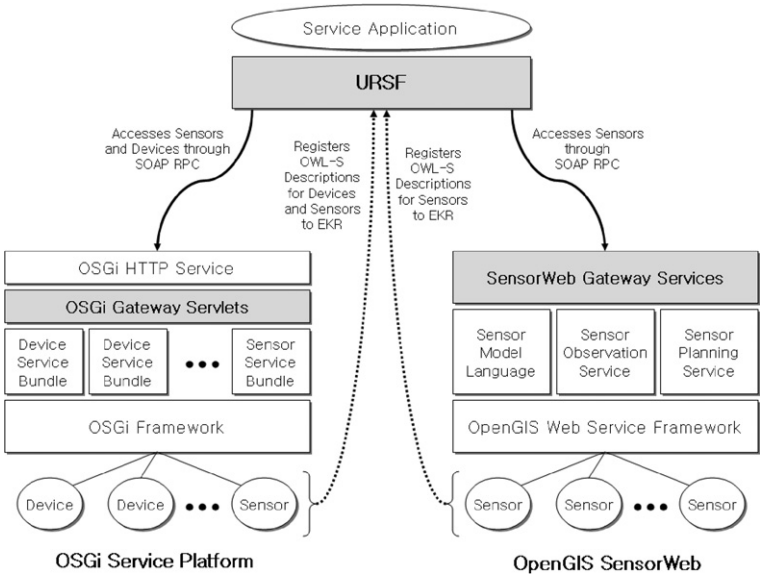

Fig. 6. Interoperation between URSF and standard service frameworks.

deployed devices and sensors need to be registered to the EKR of URSF in advance. Especially, OWL-S knowledge descriptions for OpenGIS sensors can be generated from their SensorML descriptions. As illustrated in the figure, a kind of gateway module can be used for the interoperation. The URSF service application can access OSGi devices and sensors using the OSGi Gateway Servlets implemented on top of each device and sensor service bundle. An OSGi Gateway Servlet receives SOAP RPC request messages for devices and sensors through the HTTP Service. It also parses the request messages into appropriate API calls of its corresponding device or sensor service bundle. The URSF service application can also access OpenGIS sensors using the SensorWeb Gateway Services implemented on top of the Sensor Observation Service and the Sensor Planning Service. A SensorWeb Gateway Service receives SOAP RPC request messages from URSF, translates the request messages into Sensor Observation Service messages for requested sensors and invokes the Sensor Observation Service with the translated request messages.

4. Experiments

4.1. Experimental system

We implemented a URSF prototype system and made experiments in our networked home test bed, which has a bedroom, kitchen and living room as shown in Fig. 7. The experimental environments include an

Fig. 7. Ground plan of the networked home test bed.

Evolution Robotics' Scorpion robot with Vision-based Simultaneous Localization and Mapping (vSLAM) navigation [23]. The robot is equipped with an Infrared (IR) remote controller. The environments also include IR-controlled devices such as a TV set and lighting switches, a Power Line Communication (PLC) controller and PLC-controlled devices such as an air conditioner and a motorized window blind. 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 URSF prototype implementation consists of a DWS host, an EKR server, a laptop PC for a RA mounted on the ERSP Scorpion robot and another laptop PC for a user interface to the RA. They are connected to each other via a wired or wireless LAN as in Fig. 8.

We use the Java2 Software Development Kit (SDK) as the development platform for the URSF prototype system. We also use HP's Jena Semantic Web framework [24] to implement OWL ontology reasoning functionality, IBM's BPEL4WS for Java (BPWS4J) platform to implement BPEL4WS processing functionality and the Java Server Page (JSP) to implement a Web-based user interface of the RA. And the Apache Axis Web services toolkit [34] is used to implement device Web services and SOAP communications between the URSF components. Table 1 lists concrete device Web services which are implemented

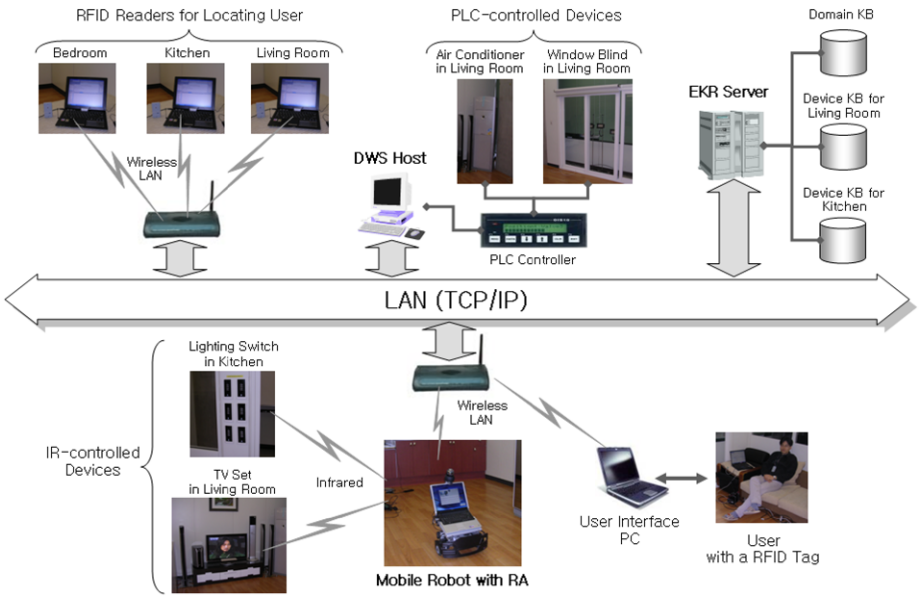

Fig. 8. Configuration of the experimental environments.

Table 1
List of concrete device Web services implementations

Web service operation	Description	Where available	Controlled devices	Installed system
getUserLocation	Returns which RFID reader senses given user ID in its radio boundary	Everyplace	RFID readers	DWS host
raiseWindowBlind	Raises window blind with PLC controller	Living room	PLC controller	DWS host
pullDownWindowBlind	Pulls down window blind with PLC controller	Living room	PLC controller	DWS host
turnOnAirConditioner	Turns on air conditioner with PLC controller	Living room	PLC controller	DWS host
turnOffAirConditioner	Turns off air conditioner with PLC controller	Living room	PLC controller	DWS host
turnOnLight	Turns on lighting with IR remote controller	Kitchen	IR remote controller	PC on the robot
turnOffLight	Turns off lighting with IR remote controller	Kitchen	IR remote controller	PC on the robot
turnOnTV	Turns on TV set with IR remote controller	Living room	IR remote controller	PC on the robot
turnOffTV	Turns off TV set with IR remote controller	Living room	IR remote controller	PC on the robot
moveTo	Moves robot to the specific point of the given place	Everyplace	Scorpion robot	PC on the robot

for the Scorpion mobile robot, RFID readers, PLC controller and IR remote controller mounted on the robot under the test bed.

4.2. Knowledge bases

The experimental domain KB contains OWL-S descriptions of domain service models for a mobile consumer robot and the concept ontology for home service environments. The following example knowledge shows a domain service model to control indoor brightness and a portion of the concept ontology, which are actually used in the experiments.

```
<!-- Instance declarations for the concept classes -->
<concepts:UserLocation rdf:ID="UserLocation_Input"/>
<concepts:AtLocation rdf:ID="AtLocation_Effect"/>
<concepts:BecomeBrighter rdf:ID="BecomeBrighter_Effect"/>

<!-- Common service model to control indoor brightness -->

<process:ProcessModel rdf:ID="RaiseIndoorBrightnessProcessModel">
  <process:hasProcess>
    <process:CompositeProcess rdf:ID="RaiseIndoorBrightnessProcess">
      <process:composedOf>
        <process:Sequence>
          <process:components rdf:parseType="Collection">
            <process:AtomicProcess rdf:ID="MoveToLocation">
              <process:hasInput rdf:resource="#UserLocation_Input"/>
              <process:hasEffect rdf:resource="#AtLocation_Effect"/>
            </process:AtomicProcess>
            <process:AtomicProcess rdf:ID="RaiseBrightness">
              <process:hasEffect rdf:resource="#BecomeBrighter_Effect"/>
            </process:AtomicProcess>
          </process:components>
        </process:Sequence>
      </process:composedOf>
      <process:hasInput rdf:resource="#UserLocation_Input"/>
      <process:hasEffect rdf:resource="#BecomeBrighter_Effect"/>
    </process:CompositeProcess>
  </process:hasProcess>
</process:ProcessModel>

<!-- Concept ontology:semantic hierarchy for general concepts -->
    ...
<owl:Class rdf:ID="BecomeBrighter">
  <rdfs:subClassOf rdf:resource="& process;UnConditionalEffect"/>
</owl:Class>

<owl:Class rdf:ID="LightOn">
  <rdfs:subClassOf rdf:resource="#BecomeBrighter"/>
</owl:Class>

<owl:Class rdf:ID="BlindRaised">
  <rdfs:subClassOf rdf:resource="#BecomeBrighter"/>
</owl:Class>
    ...
```

As shown in the example, "RaiseIndoorBrightnessProcessModel" is composed of a sequence of two atomic processes, "MoveToLocation" and "RaiseBrightness." The first one is to navigate the robot to the user's location given as input and the second one is to raise the brightness of the location. In more detail, the "RaiseBrightness" process has an instance of the "BecomeBrighter" concept as its effect, which is a super-concept of "LightOn" and "BlindRaised" as defined in the concept ontology.

The experimental device KB contains OWL-S knowledge about the concrete device Web services listed in Table 1. As shown in Fig. 8, we construct two device KB's in the EKR server: one for device Web services available in the living room and the other for device Web services available in the kitchen. When the EKR server receives a service discovery query, it decides which device KB is to be used according to the context data sent along with the query. For instance, if a service discovery query with context data "Current_user_location = Kitchen" is received, the EKR uses the device KB for the kitchen to process the query. Note that knowledge about device Web services available everyplace in the test bed is stored in both of the device KB's. The following example shows knowledge about "turnOnLight" service listed in Table 1, which turns on the IR-controlled lighting device with the IR remote controller equipped on the Scorpion robot.

```
<!-- Instance declarations for the concept classes -->
<concepts:LightOn rdf:ID="LightOn_Effect"/>

<!-- OWL-S description for turnOnLight device Web service -->
<service:Service rdf:ID="TurnOnLightService">
  <service:describedBy rdf:resource="#TurnOnLightServiceProcessModel"/>
  <service:supports rdf:resource="#TurnOnLightServiceGrounding"/>
</service:Service>

<process:ProcessModel rdf:ID="TurnOnLightServiceProcessModel">
  <process:hasProcess>
    <process:AtomicProcess rdf:ID="TurnOnLightServiceProcess">
      <process:hasEffect rdf:resource="LightOn_Effect"/>
    </process:AtomicProcess>
  </process:hasProcess>
</process:ProcessModel>

<grounding:WsdlGrounding rdf:ID="TurnOnLightServiceGrounding">
  <grounding:hasAtomicProcessGrounding>
    <grounding:WsdlAtomicProcessGrounding>
      <grounding:wsdlDocument>
        http://robot.etri.re.kr:8080/axis/LightControlService?wsdl
      </grounding:wsdlDocument>
      <grounding:wsdlOperation>
        <grounding:WsdlOperationRef>
          <grounding:portType>LightControlService</grounding:portType>
          <grounding:operation>turnOnLight</grounding:operation>
        </grounding:WsdlOperationRef>
      </grounding:wsdlOperation>
    </grounding:WsdlAtomicProcessGrounding>
  </grounding:hasAtomicProcessGrounding>
</grounding:WsdlGrounding>
```

As described in the above example, the knowledge consists of "TurnOnLightServiceProcessModel" and "TurnOnLightServiceGrounding." The atomic process of the process model has an instance of the "LightOn" concept as its effect, which is a sub-concept of the "BecomeBrighter" concept. The service grounding provides information about accessing the "turnOnLight" service operation implemented in "http://robot.etri.re.kr:8080/axis/LightControlService."

4.3. Experimental results

In the experiments, we will show how a robotic agent built on URSF can automatically integrate ubiquitous sensors and devices in our networked home test bed. In the beginning of the experiment, the user is sitting on a sofa in the living room carrying his RFID tag and the robot is in the kitchen of the test bed. For the first experiment, the user commands the robot to "Make it brighter" with the Web-based user interface of the RA as in Fig. 9. As shown in the figure, the user interface is intuitive and easy enough to use. It has several buttons for some essential commands and an additional text area for the input of ad hoc commands in forms of an OWL-S service profile. Once submitted, 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.

```
<concepts:BecomeBrighter rdf:ID="BecomeBrighter_Effect"/>

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

As explained above, to perform the requested service, the RA must know the current location of the user as one of the essential contexts for services. So, the RA sends the EKR server a discovery query for a service that has an instance of "UserLocation" concept as its output. As the result of querying, the RA receives OWL-S knowledge about the "getUserLocation" service implemented in the DWS host and executes it based on the service grounding. In this experiment, the "getUserLocation" service responds to RA with "LivingRoom" as the current location of the user by scanning the RFID readers installed in each place of the test bed. Each RFID reader used in the experiments simply detects and reports the presence of a RFID tag carried by a user in its detection range. Note that actual detection range of each RFID reader is adjustable and wide enough to cover the whole region of each place of the test bed (up to about 9 m).

After knowing the current location of the user by gathering context data, the RA discovers a domain service model that has an instance of the "BecomeBrighter" concept as its effect. As the result of discovery, the RA receives OWL-S knowledge about the indoor brightness control service model, "RaiseIndoorBrightness,"

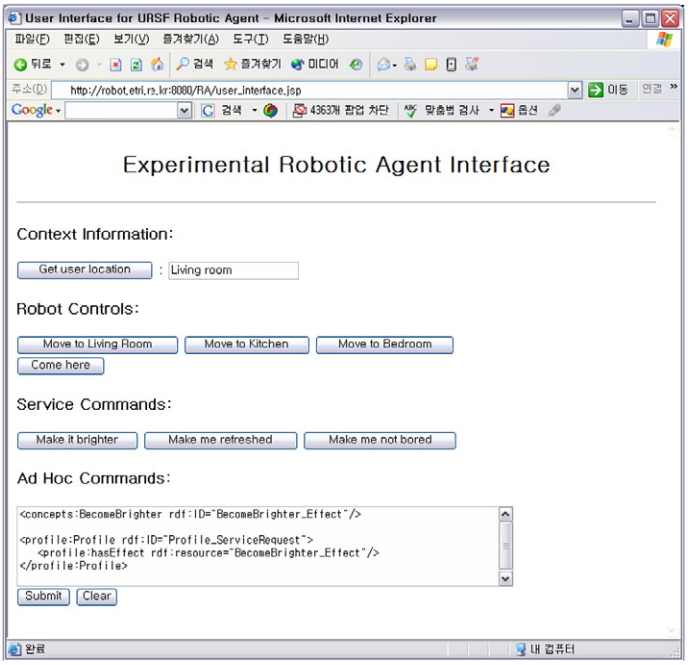

Fig. 9. Screenshot of the Web-based user interface.

given in Section 4.2, and continues to discover device Web services with it. During the device Web service discovery, the RA discovers feasible device Web services by sending the EKR server extended queries using the semantic service discovery algorithm in Section 3.2. For example, the RA tries to find a device Web service that has an instance of the "BecomeBrighter" concept as its effect by sending an exact matching query. When it fails, the RA retries to find a device Web service that has an instance of "LightOn" or "BlindRaised" concept as its effect, which are sub-concepts of the "BecomeBrighter" concept. In addition, the RA sends the EKR server the service context "Current_user_location = LivingRoom" with each query for choosing an appropriate device KB. As a result, "MoveToService," the OWL-S knowledge about "moveTo" service and "RaiseWindowBlindService," the OWL-S knowledge about "raiseWindowBlind" service, are discovered. And the following plan is generated after performing the plan composition phase with the knowledge.

```
<plan type="Sequence">
  <service name="MoveToService">
    <input name="Location_Input" value="LivingRoom"/>
  </service>
  <service name="RaiseWindowBlindService"/>
</plan>
```

Finally, the plan was translated into the BPEL4WS process and successfully executed during the plan execution phase as shown in the upper left and right photos of Fig. 10. The second experiment is the same as the first one, but the user's current location changed to kitchen. That is, from the robot's point of view, the service environments changed. By knowing the change of user's current location through the context data gathering, the RA discovers the appropriate device Web service "TurnOnLightService" that has an instance of the "LightOn" concept, a sub-concept of "BecomeBrighter," as its effect and automatically composes the following service plan for the user in the kitchen.

```
<plan type="Sequence">
  <service name="MoveToService">
    <input name="Location_Input" value="Kitchen"/>
  </service>
  <service name="TurnOnLightService"/>
</plan>
```

Fig. 10. Snapshots of the experiments.

Table 2
Performance of URSF under the experimental environments

Average time for context data gathering	Average time for knowledge discovery	Average time for plan composition	Average time for plan execution	Average latency of URSF
126 ms	312 ms	502 ms	759 ms	1691 ms

As a result of the second plan execution, the Scorpion robot successfully moved to the kitchen and turned on the lighting with the IR remote controller on it, instead of raising the window blind in the living room, as shown in the lower left and right photos of Fig. 10.

4.4. Performance evaluation

This section will describe performance evaluation results for the URSF prototype system. The performance of the URSF prototype can be assessed by calculating average latency from multiple service executions during the above experiments. The average latency of URSF refers to the average time elapsed during the automated integration procedure illustrated in Fig. 4. It provides an effective and straightforward method to evaluate the performance. The average latency of URSF (L_{ave}) is calculated by the following formula where N is the total number of service executions, Tr_i is the time of starting to gather context data for service execution i (more exactly, the time that an OWL-S request profile starts to be generated) and Tp_i is the time of completing plan execution phase for service execution i (more exactly, the time that a BPEL4WS process generation is completed).

$$L_{\text{ave}} = \frac{\sum_{1 \leqslant i \leqslant N} (Tp_i - Tr_i)}{N}$$

Table 2 summarizes the performance evaluation results when the total number of experimental service executions is 20 ($N = 20$). It describes average times elapsed to perform context data gathering and each service composition phase as well as the average latency of URSF. For reference, computer systems and communication network environments used for the performance evaluation are given as follows:

- DWS host: Pentium 4 2.8 GHz/1 GB RAM/Windows XP.
- EKR server: Dual Xeon 2.8 GHz/3 GB RAM/Windows Server.
- RA laptop: Pentium 4 2.0 GHz/512 MB RAM/Windows XP.
- User interface laptop: Pentium 4 2.0 GHz/512 MB RAM/Windows XP.
- Network environments: 100 Mbps LAN and wireless LAN access points.

As shown in the table, the average latency of the URSF prototype system looks acceptable even though there were no considerations for scalability and real-time support of the prototype system. This is because the evaluation is performed under limited experimental environments which have only a small number of devices and sensors as well as predetermined domain service models. That is, small-sized experimental knowledge bases are used for the performance evaluation. In addition, all the components are connected through fast local communication networks. The evaluation results do not give us a basis to assess the performance of a practical URSF system. For this reason, we are planning to use large scale knowledge bases, wide area communication networks and effective metrics to evaluate practical performance more precisely. This work will be done along with the development of the next version of URSF focused on scalability and real-time supports.

5. Conclusions

In this paper, the URSF architecture is proposed and its working prototype system is demonstrated in our networked home test bed. URSF enables automated integration of networked service robots into ubiquitous

computing environments including wireless sensors and actuators to provide ubiquity of robotic services. URSF utilizes the use of semantic Web services technology and an AI-based planning methodology to support automated interoperation between robotic agents, wireless sensors and service devices connected to each other in the ubiquitous networking environments. That is, Web services for robots, sensors and devices are implemented as the unified interface method for accessing them. Then, knowledge about the Web services is described in OWL-S, the semantic Web services description language, and is registered to the environmental knowledge bases so that a robotic agent can automatically discover the required knowledge, compose a feasible service plan and execute the service plan for the service environments.

Currently, there are some limitations in the proposed framework. One is about supporting scalability. URSF agents discover required knowledge from centralized environmental knowledge bases, which are assumed to be well-known and always available to them. However, real ubiquitous computing environments will be mostly based on ad hoc networks that are completely distributed and dynamic in nature. And they will surely consist of a huge number of mobile devices and sensors. This means that the centralized discovery approach is likely to suffer from a serious scalability problem in actual ubiquitous computing environments. Another limitation is its real-time support. URSF does not provide any Quality of Service (QoS) mechanism at a framework level. It is because real-time support for Web services does not matter in LAN environments even though there is no concern of QoS. So, URSF cannot ensure real-time invocation of Web services over Wide Area Network (WAN) environments which often suffer from unpredictable communication delays (i.e. communication between the URSF components dispersed through the public Internet).

Our future research direction will be primarily focused on overcoming the current limitations of the framework. It will include how to effectively distribute the knowledge to robots, ad hoc sensors and devices in the service environments, and how to discover and plan with the distributed knowledge. It will also include how to cooperate and share knowledge between multiple robotic agents under the same or relevant service environments. In addition, we will incorporate a QoS mechanism into the URSF architecture to support real-time services. Another important future research focus is security and privacy issues in URSF. Security and privacy issues arise owing to the use of the Web technologies and Internet protocols, which are open standards such as HTTP, SOAP and XML, as the communication method between the URSF components. We are planning to approach these issues basically from the Role-based Access Control (RBAC) and XML digital signature technologies [43] using the Public Key Infrastructure (PKI). We are also considering using an ad hoc cluster-based security approach [5,21] for dynamic ad hoc service environments.

And finally, we expect that the proposed framework will contribute to the development of intelligent software agents not only for robotic services but for a variety of service domains in the upcoming ubiquitous computing era.

Acknowledgement

This work is financially supported by URC technology development program of Korea Ministry of Information and Communication (MIC).

References

[1] T. Berners-Lee et al., The semantic Web, Scientific American (2001). Available from: <http://www.scientificamerican.com/article.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21&catID=2>.

[2] K. Erol, D. Nau, J. Hendler, UMCP: a sound and complete planning procedure for HTN planning, in: Proceedings of the 2nd International Conference on AI Planning Systems, Chicago, 1994, pp. 249–254.

[3] K. Goldberg et al., The Mercury project: a feasibility study for Internet robotics, IEEE Robotics and Automation Magazine 7 (1) (2000) 35–40.

[4] X. Hou, J. Su, A distributed architecture for Internet robot, in: Proceedings of the IEEE International Conference on Robotics and Automation, New Orleans, 2004, pp. 3357–3362.

[5] S. Jin et al., Cluster-based trust evaluation scheme in an ad hoc network, ETRI Journal 27 (4) (2005) 465–468.

[6] B.K. Kim et al., Web services and ubiquitous control platform for the knowledge distributed robot system, in: Proceedings of the IEEE/RSJ Conference on Intelligent Robots and Systems, Sendai, Japan, 2004. Available from: <http://staff.aist.go.jp/bk.kim/Publication/IC/IROS2004.pdf>.

[7] J.H. Kim, Y.D. Kim, K.H. Lee, The third generation of robotics: ubiquitous robot, in: Proceedings of the 2nd International Conference on Autonomous Robots and Agents, New Zealand, 2004. Available from: <http://www-ist.massey.ac.nz/conferences/icara2004/files/Papers/Paper01_ICARA2004_001_007.pdf>.
[8] K. Kiyoshi, Ubiquitous intelligent robotics, ATR UptoDate, 2003. Available from: <http://results.atr.jp/uptodate/ATR_2003sum/kogure.html>.
[9] S.A. McIlraith, T.C. Son, H. Zeng, The semantic Web services, IEEE Intelligent Systems 16 (2) (2001) 46–53.
[10] L. Miller, A. Seaborne, A. Reggiori, Three implementations of SquishQL, a simple RDF query language, LNCS 2342 (2002) 423–435.
[11] S. Narayanan, S. McIlraith, Simulation, verification and automated composition of Web services, in: Proceedings of the International World Wide Web Conference, Honolulu, 2002, pp.77–88.
[12] D.S. Nau et al., SHOP2: an HTN planning system, Journal of Artificial Intelligence Research 20 (2003) 379–404.
[13] M. Nidd, Service discovery in DEAPspace, IEEE Personal Communications 8 (4) (2001) 39–45.
[14] S.R. Oh, IT-based intelligent service robot, in: Proceedings of the 1st NSF PI Workshop on Robotics and Computer Vision, Las Vegas, 2003. Available from <http://www.vcl.uh.edu/~rcv03/materials/slides/SangRok.ppt>.
[15] T. Sato, T. Harada, T. Mori, Environment-type robot system 'robotic room' featured by behavior media, behavior contents and behavior adaptation, IEEE/ASME Transactions on Mechatronics 9 (3) (2004) 529–534.
[16] P. Saucy, F. Mondada, Open access to a mobile robot on the Internet, IEEE Robotics and Automation Magazine 7 (1) (2000) 41–47.
[17] R. Simmons, Xavier: an autonomous mobile robot on the Web, in: Proceedings of the IEEE/RSJ Conference on Intelligent Robots and Systems, Victoria BC, 1998. Available from: <http://www.ri.cmu.edu/pub_files/pub1/simmons_reid_1999_1/simmons_reid_1999_1.pdf>.
[18] E. Sirin et al., HTN planning for Web Service composition using SHOP2, Web Semantics 1 (4) (2004). Available from: <http://www.mindswap.org/papers/SHOP-JWS.pdf>.
[19] M.R. Stein, Interactive Internet artistry, IEEE Robotics and Automation Magazine 7 (1) (2000) 28–32.
[20] D. Wang, X. Ma, X. Dai, Web-based robotic control system with flexible framework, in: Proceedings of the IEEE International Conference on Robotics and Automation, New Orleans, 2004, pp. 3351–3356.
[21] G. Wang, G. Cho, Compromise-resistant pairwise key establishments for mobile ad hoc networks, ETRI Journal 28 (3) (2006) 375–378.
[22] Business Process Execution Language for Web Services Version 1.1. Available from: <http://www-128.ibm.com/developerworks/library/specification/ws-bpel/>.
[23] ERSP 3.0: Robotic Development Platform. Available from: <http://www.evolution.com/products/ersp/>.
[24] Jena – A Semantic Web Framework for Java. Available from: <http://jena.sourceforge.net/>.
[25] Jini Network Technology. Available from: <http://www.jini.org/>.
[26] OpenGIS SensorWeb. Available from: <http://www.opengeospatial.org/functional/?page=swe>.
[27] OpenSLP Project Homepage. Available from: <http://www.openslp.org/>.
[28] OSGi Technology. Available from: <http://www.osgi.org/osgi_technology/index.asp?section=2>.
[29] OWL-S: Semantic Markup for Web Services. Available from: <http://www.daml.org/services/owl-s/1.0/owl-s.html>.
[30] Salutation Consortium Homepage. Available from: <http://www.salutation.org/>.
[31] Technical Committee on Networked Robots. Available from: <http://www.informatik.uni-freiburg. de/~burgard/tc/>.
[32] Ubiquitous Computing Homepage. Available from: <http://www.ubiq.com/hypertext/weiser/UbiHome.html>.
[33] UPnP Forum Homepage. Available from: <http://www.upnp.org/>.
[34] Web Services–Axis. Available from: <http://ws.apache.org/axis/>.
[35] W3C Recommendation: OWL Web Ontology Language Guide. Available from: <http://www.w3c.org/TR/owl-guide/>.
[36] W3C Recommendation: OWL Web Ontology Language Semantics and Abstract Syntax. Available from: <http://www.w3c.org/TR/owl-absyn/>.
[37] W3C Recommendation: RDF Primer. Available from: <http://www.w3c.org/TR/rdf-primer/>.
[38] W3C Recommendation: RDF Semantics. Available from: <http://www.w3c.org/TR/rdf-mt/>.
[39] W3C Recommendation: RDF Vocabulary Description Language. Available from: <http://www.w3c.org/TR/rdf-schema/>.
[40] W3C Recommendation: SOAP Version 1.2 Primer. Available from: <http://www.w3c.org/TR/soap12-part0/>.
[41] W3C Recommendation: Web Services Architecture. Available from: <http://www.w3c.org/TR/ws-arch/>.
[42] W3C Recommendation: Web Services Description Language (WSDL) Version 2.0 Primer. Available from: <http://www.w3.org/TR/wsdl20-primer/>.
[43] XML Signature Working Group. Available from: <http://www.w3.org/Signature/>.