Enhancing Residential Gateways: a Semantic OSGi Platform

Rebeca P. Díaz Redondo, Ana Fernández Vilas, Manuel Ramos Cabrer, José J. Pazos Arias, Jorge García Duque, Alberto Gil Solla
Dept. of Telematics Engineering, University of Vigo, 36310, Spain
{rebeca, avilas, mramos, jose, jgd, agil}@det.uvigo.es

Abstract

In the last years, OSGi (Open Service Gateway Initiative) Service Platform specification has become the most adopted solution to the technological problem of building residential gateways. As many others Service-Oriented Architectures (SOAs), the framework is supported by a Service Registry which facilitates the interaction between clients and services. However, two main drawbacks can be identified: (1) service discovery is based on syntactic matchmaking, which obliges the potential client to know the concrete interface of the service it demands; and (2) service invocation assumes prior knowledge of the service signatures. Both are important obstacles in a pervasive environment like the one envisioned by OSGi at the smart home. To address this problem, we propose a new Semantic OSGi Platform, in which improvements to the Service Registry enable both the application of semantic reasoning to discovery, and the automation of invocation.

Index Terms—Pervasive Computing, Semantic Services, Residential Gateways, OSGi

I. Introduction

Nowadays, the OSGi (Open Service Gateway Initiative) Service Platform specification [1] is the most adopted solution to the technological problem of building a control system for the networked home. Its success can mainly be ascribed to the following reasons: (a) the communication among all devices in home is easy because it supports different widespread protocols; (b) it defines a cooperative model where applications can dynamically discover and use services provided by others; and (c) it is possible a flexible remote management of these applications and the services they provide.

This solution perfectly fits in with decentralization, diversification and ubiquity in smart home environments in particular, and pervasive environments in general. However, OSGi service discovery and usage assume the client to have too much information about the service to use. Although the client could be unaware of service location, it is obliged to know the name of the service interface (for discovery purposes) as well as its methods’signatures (for invocation purposes).

Surprisingly, this strategy comes into conflict with the remote management of OSGi applications, one of the mainstays of this framework. How can OSGi applications be remotely deployed and work properly on every customer platform if devices at home can be totally different from one customer to another? What is more, as the OSGi specification admits, the service platform at home can be used to run services from different service providers. Therefore, it is hardly realistic to suppose that a specific provider knows, a priori, the interfaces which match the services from other providers. In this context, the simple syntactic matching by name provided by OSGi is not enough. A smarter discovery mechanisms should be provided to make an open residential gateway really possible. This problem is not an OSGi specific problem but one more example of those which motivate the Semantic Web conception [2], a Web where information is well-defined, entailing better cooperation between computers and people.

On the basis of this trend, we propose a semantic approach to service discovery which turns OSGi into a Semantic OSGi platform becoming the cornerstone for an horizontal market at home. In this new platform (1) OSGi services describe its properties and capabilities so that any other software element in the residential gateway can automatically determine its purpose (semantic discovery); and (2) any software element is also automatically able to
determine how to invoke the OSGi services. Both tasks involve the semantic markup of OSGi services by using appropriate ontologies, which is the core of the Semantic Web and, by extension, of the Semantic OSGi platform we propose.

The paper is organized as follows. The next section briefly overviews the characteristics of the OSGi framework, whose drawbacks regarding to services discovery and invocation are described in Sect. III. Our proposal to obtain a Semantic OSGi platform is detailed in sections IV and V. Subsequently, in Sect. VI, we define scenarios where our proposal is especially beneficial. Finally, a brief discussion including conclusions and future work is presented in Sect. VII.

II. An OSGi Overview

The OSGi platform [1] mainly consists of a JVM (Java Virtual Machine), a set of running components called bundles, and an OSGi framework. On the one hand, a bundle is a Java ARchive (JAR) and the minimal deliverable application in OSGi. Bundles have a restrictive lifecycle which is externally managed by the OSGi framework. On the other hand, the minimal unit of functionality is really what OSGi calls a service. Thus, a bundle is designed as a set of cooperating services, which are discovered after being published in the OSGi Service Registry.

Registering OSGi Services: When a bundle registers a service object (by calling the registerService() method), it specifies the name of the Service Interface and it may also include a further description of the service by using a collection of key/value pairs (a Dictionary object). For instance, in Fig. 1 bundle A registers the service object (serviceobj) jointly with a dictionary object (props), under the Printing Service Interface as follows: registerService("Printing", serviceobj, props)

Obtaining OSGi Services: Bundles can obtain an OSGi service by an active search or by the event mechanism provided by the framework (Service Listeners). Because of the latter mechanism, bundles are automatically notified when a service is registered, modified, or is in the process of unregistering. This events can be filtered by the listener. In the example of Fig. 1, bundle B might create a service listener to obtain information only about printing services:
addServiceListener(listener, "(objectClass="+Printing"+)")

Anyway, bundles can also actively search the Service Registry to obtain specific services by using the getServiceReference() method, whose input parameter is the name of the Service Interface. For instance, bundle B can obtain the Service Reference object corresponding to the service registered by bundle A as follows:
ref = getServiceReference("Printing")

Using OSGi Services: Once bundle B has obtained the Service Reference object, it must obtain the service object itself to be able to use the required method, colorPrinting in this case:
printer = getService(ref)
printer.colorPrinting(file)

On another hand, the OSGi framework also offers the possibility of refining the search by using the getServiceReferences() method, whose input parameters are (i) the name of the Service Interface and (ii) filtering information to help the matching process. The filter syntax is based on the LDAP language (Lightweight Directory Access Protocol) [3]. For instance, to obtain the list of all PostScript services belonging to the Service Interface Printing, a bundle could use the order:
ref = getServiceReferences("Printing", "(type=PS)") where the filter type=PS is matched against the dictionary of properties provided on registration.

III. Why a Semantic OSGi Platform?

Undoubtedly, a mechanism for automating service discovery and invocation, like the one supported by the OSGi Service Registry, is essential in ubiquitous environments like the one envisioned by a smart home conception. Even
so, several drawbacks, mainly caused by using syntactic matchmaking, can be ascribed to OSGi mechanisms:

**Discovery:** The current discovery mechanism does not consider the semantics of both the requested service and the requester query; instead, a syntactic matchmaking is used. Thus, the discovering phase only can retrieve Service References which (1) exactly match the Service Interface name in the query or (2) are described by the same syntactic keywords in their properties when a filter is used. This mechanism entails, among others, problems with both synonyms (semantically similar services, but syntactically different) and homonyms (syntactically equivalent services, but semantically different). For instance, the problem when searching for a “Printing” service which was registered under the service name "print".

**Selection:** OSGi discovery may result on a list of matched services, for instance when several printers at home have registered under the “Printing” Service Interface name. Adding filtering information at the discovery phase, “(type=PS)” in our example, can help to the selection process. However, the OSGi filtering is again based on syntactic comparisons, i.e. the properties are simply strings with any meaning; so, the synonym and homonym problems appear again. What is more, without a semantic interpretation of the service’s properties, there is no possibility for a smart selection of the “best” service.

**Invocation:** Actually, OSGi would need a more flexible approach to service invocation. In fact, not only has the requester bundle to know the Service Interface name ("Printing"), but for invoking the service, it also has to know the name of the method ("color Printing") and the number and type of its parameters ("File"). This is clearly and obstacle in a pervasive environment since it prevents “unaware” bundles (i.e. without prior knowledge about the Service Interface) from dynamically invoking the service.

**Key elements of the new Semantic OSGi Platform**

The problem of automatic discovery and invocation of services is not new. In fact, it has been repeatedly analyzed in the Web Services field, where composition and monitoring of services are also specially important factors [4]. As a result of these analysis, it is concluded the solution for services discovery should involve the use of ontologies. Because as “an ontology is a formal, explicit specification of a shared conceptualization” [5], it facilitates knowledge sharing by providing a common understanding for both the service requester and the service provider, in this case. Thus, in the in-home environment, domain-specific ontologies which semantically organize the usual services at home can provide the key.

This semantic conceptualization can also enhance the selection of the most appropriate service, especially if it is applied not only for services, but also for their properties. As it is well-known, an ontology-based search can refer not only to the name of one or more classes (if inheritance) but also to specific values for properties, constraints on values of properties, or any combination of the above. Thus, for a complete semantic solution, these properties should be described according to other domain-specific ontologies which conceptualize the parameter values. Although these ontologies were developed independently, it is possible to explicitly express relationship between their classes. For instance, if print quality was semantically classified, a filter based on this criterion could be unambiguous. In general, with a knowledge base for service properties, it is possible to reason about the available services to smartly select the most appropriate one.

Finally, to solve the service invocation problems, the description of OSGi services should include not only its semantic classification (category and properties) but also enough information for a requester bundle to automatically construct the invoking primitive and to process the response.

Consequently, and with the aim of getting over the obstacles previously described, we define a Semantic OSGi platform. We propose to replace the table-based structure in the OSGi Service Registry by an ontological structure integrating all the ingredients above: semantic classification, semantic description of properties and information of invocation. Within this context, any software element (bundles and/or any other software in the platform) would be able not only to determine the purpose of any OSGi service and to select the most appropriate according to its interests, but also to automatically invoke it in this new framework.

Previous approaches have tried to promote smart spaces by using OSGi, like in [6], whose authors propose using OSGi as a suitable infrastructure to integrate various devices and sensors to provide pervasive computing environments. However, they do not resolve the problem of service search and invocation. In [7] these two problems are directly tackled but not from a semantic perspective, the authors present how services can be imported from and exported to the Jini and UPnP technologies, showing that OSGi perfectly bridges multiple discovery protocols. On another hand, the authors of [8] propose to define a semantic environment for describing contextual information to be used by context-aware applications. However, OSGi is only used as a support layer, without improving the OSGi framework at all. After an exhaustive revision of the state-of-the-art, we have not found any proposal about
integrating semantic reasoning within the OSGi Service Registry to avoid all the aforementioned problems with service discovery, selection and invocation.

Regarding the ontological description of the smart home, some initiatives have come up to lay on research issues of ubiquitous computing and user modeling. For instance, UbisWorld [9] is used to represent parts of the real world (a house, a shop, etc.) and all the individuals and elements belonging to it. Its knowledge is modeled by two ontologies which are under development: GUMO (General User Model Ontology) [10] and UbisOntology, the ontology for ubiquitous computing. SOUPA (Standard Ontology for Ubiquitous and Pervasive Applications) [11] is another relevant approach to model and support pervasive computing applications. It consists of two set of ontologies: SOUPA Core ontologies which define generic vocabularies that are universal for different pervasive computing applications; and SOUPA Extension ontologies which define additional vocabularies for defining specific types of applications.

However, no focus has been on the semantic description of the in-home services. For this, we propose to take advantage of the previous efforts made in the Semantic Web field, where the appropriate semantic description of services is the main point. In this area, the most salient initiatives to describe Semantic Web Services are WSMO (Web Service Modeling Ontology) [12] and OWL-S [13]. Although both approaches cover the same field with the same goals trusting ontologies to get them, OWL-S is considered to be clearly more mature in certain aspects and expressive enough to be applied in the smart home environment [14].

Consequently, we have decided to extend the OWL-S ontology to support the appropriate description of the OSGi services by defining the OWL-OS ontology (the OWL-OSGi Services ontology). However, the right description of OSGi services is not enough. We also need to provide a way for OSGi bundles to register semantically their services and to get the correct references to the needed ones. On the whole, the OWL-OS ontology in the following section; and the new methods for registering and discovering services in Sect. V enable to support the OWL-OSGi framework: a new Semantic OSGi platform.

IV. The OWL-OS Ontology

OWL-S is an OWL-based Web Services ontology which provides a semantic markup language specially thought to specify Web Services in unambiguous and computer-interpretable form. The OWL-S ontology is organized to provide three essential types of knowledge about any Service provided by a Resource (upper part of Fig. 2): (i) the ServiceProfile tells “what the service does”; (ii) the ServiceModel explains “how the service works”; and (iii) the ServiceGrounding details “how to access it”.

We have adapted this ontology to the peculiarities of the OSGi services by defining the OWL-OS ontology (OWL-OSGi Services ontology). Following the recommendations of the OWL-S specification, we take advantage of the OWL subclassing to create a specialized vision of the OWL-S services by defining an OSGiService. As Fig. 2 shows, the first characteristic of this OSGi services entails they are only provided by instances of the devices-in-home ontology. This ontology embraces all the typical devices and appliances at home (heating, dishwasher, doors, blinds, etc.) unifying the conceptualization of this field. Besides, we use subclassing to capture the peculiarities of OSGi services by defining a specific OSGi Service Profile (Sect. IV-A) and a specific OSGi Service Grounding (Sect. IV-B). Regarding the ServiceModel, since the OWL-S one fits perfectly with the OSGi perspective, we do not adopt any restriction in this case, i.e., any OSGi service is described by a ServiceModel.

A. Describing Services: OSGi Profiles

An OWL-S ServiceProfile draws together: (i) information about the entity that provides the service; (ii) its functional description (inputs, outputs, preconditions, etc.); and (iii) the description of additional features that the service may want to specify. The latter information is actually stored in two attributes:

- serviceCategory refers to an entry in some ontology or taxonomy of services
- serviceParameter is an expandable list of properties that may help the requester to find the service.

In the OWL-OS ontology, the OSGiServiceProfile inherits these attributes, although we have particularized them as Fig. 3 shows: defining an OSGiServiceCategory and an OSGiServiceParameter. Regarding the OSGiServiceCategory, we have defined an ontology of the in-home services called operations-at-home, representing the typical services that can be found in the smart home (see Fig. 2). An OSGi Service Profile should provide one or more categorical descriptions taken from the ontology operations-at-home, since a single service often can be categorized according to more than one functionality criteria ("opening a window" can be both an airing service or a lighting service). Despite the operations-at-home ontology is the shared conceptualization we propose for the smart home, services can also be described by external ontologies from third-party providers.
On another hand, the OSGi service provider may specify additional information, to enrich the OSGi service selection process. This information is maintained in a list of properties of the type OSGiServiceParameter. An OSGi Service Profile should provide, at least, one OSGiServiceParameter in the subclass OSGiLocation which stores the service location by using the ontology locations-at-home. This data is essential to define a context-aware framework, since the service location might be a determining factor in the selection process. Moreover, an OSGi Service Profile can specify some other parameters which, depending on the kind of service, can be taken from external ontologies (downloaded on demand from the service provider).

B. Accessing to Services: OSGi Groundings

In OWL-S, both the ServiceProfile and the ServiceModel are thought of as abstract representations, while only the ServiceGrounding deals with the concrete level of specification. Consequently, we have defined an OSGiServiceGrounding (subclass of the ServiceGrounding) which provides a suitable way for mapping the OSGiServiceProfile to the functionality offered, through a Service Interface, by a bundle running in the OSGi framework.

Any OSGiServiceGrounding instance contains a list of OSGiAtomicProcessGrounding instances: really one when it correspond with one public method in the Service Interface of the bundle; or more than one when we use service composition, which is part of our future work. An OSGiAtomicProcessGrounding instance (see Fig. 3) refers to a specific OSGi service invocation, i.e., a specific public method in a Service Interface of a running bundle. This OSGi service invocation implements an OWL-S atomic process (from the process subontology) by defining the following properties: (a) the atomic process to which this grounding applies, owlsAtomicProcess; (b) the interface/class name under which this bundle has registered the OSGi Service, OSGiServiceInterface; (c) the name of the Java public method in the Service Interface, OSGiMethodName; (d) the return value of the method, OSGiReturn; and (e) an object of parameters, OSGiParameters. As we show in the following section, this information is enough to automate the invocation of an OSGi service.

C. A semantic OSGi service in OWL-OS

Using Fig. 4, let us show in this section an OWL-OS semantic representation of our Printing Service Interface in Fig. 1. In the OSGi syntactic approach, bundle A registers a service object serviceobj in the framework which implements the Service Interface Printing with a
set of properties \texttt{props}. In the current (non-semantic) OSGi platforms, the Service Registry stores a table-based representation of that OSGi service (identifier 131 in Fig. 4) which is inspected when bundles look up for services. The discovery process is based on syntactic matching against the following fields: Interface Name and Properties. In view of the limitations in Sect. III, the aim is a new Semantic OSGi Service Registry which, moreover, stores the OWL-OS instances corresponding to that OSGi service. The first remark is about the differences between the OSGi and OWL-OS conceptions of what is a service. On the one hand, an OSGi service is a Java object, registered into the OSGi framework, which implements the public methods in a Service Interface. In our example, the bundle \textit{A} offers an implementation of the Printing Service Interface through the service object serviceobj. So the individual service invocations \texttt{colorPrinting} and \texttt{b&wPrinting} are treated as a whole in the table-based registry— the properties are bounded to the interface and it is not possible to specify different properties from one invocation to another (for instance, a different response time for \texttt{colorPrinting} and \texttt{b&wPrinting}). Surely, the objective of the properties when registering a Service Interface is not to offer parameters of a service, but of a Service Interface, which is a clear limitation. Besides, it is impossible to search for a specific type of invocation inside the Service Interface, that is, to distinguish different functionalities (color or b&w printing) in the same interface.

In a smarter conception of service, the OWL-OS one, the Printing Service Interface offers a set of printing methods (considered services from the OWL-OS perspective) with (1) different category values (\texttt{printColor, printB/W}, \texttt{printSimplex}, etc. in the ontology \texttt{operations-at-home}) and (2) different service parameters values (printing format, quality, response time, etc). Moreover, it will be possible searching for a specific printing method instead of searching for the Service Interface as a whole. So, the Printing Service Interface will be represented in the OWL-OS ontology by means of two different services (see Fig. 4), one for the interface method \texttt{colorPrinting} (OSGiService instance: Service131-1) and one for the interface method \texttt{b&wPrinting} (OSGiService instance: Service131-2). In this case, each service instance is described by the same Service Model, the PrintFileModel, which is an atomic process directly invokable and executed in a single step. At the OWL-OS level, the differences between these two services are their profiles (i.e. their semantic characteristics) and groundings (i.e. the details for invoking the service). Let us take, for instance, the \texttt{colorPrinting} service (Service131-1) and describe its OWL-OS representation:

- **Profile:** An \texttt{OSGiServiceProfile} instance, ServiceProfile131-1, has been created in the ontology containing the semantic characteristics of the functionality \texttt{colorPrinting}. The main elements in the profile which support this objective are
- \texttt{ServiceProfile} instance, ServiceProfile131-1, from the...
ontology operations-at-home) and a list of OsgiServiceParameter instances, one of them from the class OSGILocation. Regarding the service parameters, apart from the location, the service is qualified by the parameter printFormat which takes its values from an external vocabulary.

- **Grounding:** An OSGIServiceGrounding instance, Grounding131-1, has been created in the ontology containing the details to construct the service request, and so resolving the problems of invocation. The main elements in the grounding are the value of the data property MethodName (value colorPrinting); the input parameters (object property OSGIParameters, instance Params131-1) and output parameters (object property OSGIReturn, instance Ret131-1).

In our example, the bundle A registers the Service Interface Printing with a dictionary of properties which include type=PS and location=spare room. We intentionally use the same parameters in the semantic description of the service (see SrvParam131-1-1 and SrvParam131-1-2 in Fig. 4). However, remark that, in the semantic description, these parameters are linked to a specific invocation in the Service Interface. Moreover, the service parameters and their values are taken from an ontological conceptualization of the in-home domain. So, we avoid the problems arisen from the syntactic matchmaking and we open the doors to an intelligent and automatic process of service selection by using semantic reasoning.

V. The OWL-OS/OSGi Framework

As we have previously introduced, the OWL-OS ontology plays an essential role in the new OSGi framework, because it enables a more appropriate description of the OSGi services. However, it is necessary to relate any OSGi service with the constructs in OWL-OS to manage the OSGi services in a semantic way, that is, (i) to indicate the correspondence between a particular method in an OSGi Service Interface and an OWL-OS ServiceModel and (ii) to semantically specify the OSGIServiceProfile. But, who offers this semantic information about the OSGi services? An how is it provided to the OSGi framework?

In our extension, we propose the bundle developer specifies this semantic information in the bundle’s manifest file by defining a set of new manifest headers, which is allowed by the OSGi specification – hereinafter collectively referred to as OWL-OS headers. For a specific bundle, these new headers describe the model and profile correspondence of every single method in the Service Interfaces the bundle provides.
A. Registering Semantic OSGi Services

The Semantic OSGi Service Registry creates new individuals and classes when a new service is registered in the ontology. For this, the registry uses the information in the manifest OWL-OS headers of the bundle which registers the service; so the registering methods in the BundleContext interface remain the same as in the current OSGi specification. As Fig. 5 shows, the bundle uses the same registering method, but as the bundle includes a manifest file with a semantic section, the semantic registry can store much more useful information to manage these services. Note we maintain the dictionary of properties in the registering method just for compatibility reasons with the standard Service Registry.

B. Semantic discovery of OSGi Services

Since obtaining OSGi services precisely the requesting bundle to know too much information in advance, we propose using the knowledge stored in the OWL-OS ontology to make the OSGi services discovery easier. Consequently, this fact is reflected in the querying methods, which have been slightly modified to support more flexible queries. Thus, drawing an analogy with the current OSGi methods (Sect. II), we propose to add the two following methods to the BundleContext interface:

\[
\begin{align*}
\text{semGetServiceReference} & \quad \text{(String OntologyURI, String Category)} \\
\text{semGetServiceReferences} & \quad \text{(String OntologyURI, String Category, String semanticFilter)}
\end{align*}
\]

A flexible service discovery is provided by the first two parameters. Thus, instead of being necessary to provide the name of the Service Interface, the requesting bundle uses the classification provided by the OWL-OS ontology to specify which kind of service is needed. The requesting bundle specifies which ontology is used to classify the required service (the first parameter OntologyURI) and the name of the appropriate service category in that ontology (the second parameter Category). Finally, the semanticFilter parameter in the second method defines filtering information to help the search process. For instance, Fig. 5 shows how a bundle can discover the colorPrinting service by asking for any service belonging to the printColor class defined in the operations-at-home ontology.

Similarly to the current OSGi specification, the first method returns only one Service Reference object. So, if there are multiple objects satisfying the search criterion (OntologyURI#Category), the selection is the one defined by the OSGi specification, i.e. the service object with the highest SERVICE_RANKING. However, the second method would return an array of Service Reference objects satisfying the search criterion (OntologyURI#Category) but also satisfying a selection filter which is specified in the parameter semanticFilter. The service ranking is obtained by using both explicit and implicit reasoning. Applying these inference rules, potentially useful services are selected and marked according to previously defined criteria (frequency of use, energy consumption, etc.).

Regarding the filtering information, the current OSGi specification defines a syntactic filtering based on the LDAP language. However, the flexibility of the query could be enhanced if this filtering were based on semantic reasoning. So, we propose the semanticFilter to be an OWL query about individuals which pertain to the Category specified in the second parameter of the method. Since OWL-QL [15] (OWL Query Language) is the usual language for query-answering dialogues among Semantic Web agents using knowledge represented in OWL, we propose this language to replace LDAP in expressing semantic filters.
C. Invoking the OSGi service

Once the bundle has found the desired service, it just needs to use the information in the service grounding to invoke the service. For this propose, we have extended the interface ServiceReference of the OSGi Framework Specification. A Service Reference is used in the framework to get the service object but also to examine the properties of the service. So, we have added methods to obtain the OWL-OS information linked to the service. This get-set style methods (see getMethodName and getInputParameter in Fig. 5) allows the requester bundle to discover the name of the public method and the parameters in order to construct the invoking primitive (serviceobj.colorPrinting(file) in the example).

VI. Implementation and use case

For the implementation of the proposal we have selected OSCAR (Open Source Container ARehitecture\(^3\), an open software implementation of the OSGi framework. To manage the local OWL-OS ontology and provide the proposed semantic OSGi services, we use the Protégé OWL API\(^4\). This open-source Java library provides classes and methods to load and save OWL files, to query and manipulate OWL data models, and to perform reasoning. We have implemented a semantic version of the OSCAR registry which interprets the new bundle manifests and manage the OWL-OS ontology to accomplish the tasks of registering (populating the ontology) and searching (querying the ontology) services as described in the previous sections. To include query processing in our framework, we use the OWL-QL toolkit, which provides a Java implementation of a OWL-QL Server by integrating JTP (Java Theorem Prover)—both from Artificial Intelligence Laboratory at Stanford University\(^5\). Over this software prototype, we have designed a bundle implementing the following use case:

**Scenario 1:** John has signed a maintenance contract for his OSGI residential gateway. As part of this contract, a “human presence simulator” is received at John’s home. The functionality of this bundle is trying to dissuade potential burglars by simulating that someone is at home by turning on and off some appliances (lights, TV, blinds, etc.).

In this scenario, it is assumed that the provider broadcast the same bundle to all the subscribers’ home. However, the interactions with lighting devices (lights, blinds, etc.) or sounding devices (HI-FI sytem, TVs, etc.) at each house can be slightly different, depending on how many devices are at home and their sophistication. Anyway, the “human presence simulator” bundle needs to interact properly with the controllers of these appliances, taking into account their peculiarities and acting consequently. If the broadcast bundle were programmed according to the current OSGi specification, it would hardly work properly at every single home. However, in the new OWL-OS/OSGi Framework, the bundle is able to obtain the service/s it needs and to know how to use them without having so much information in advance as follows.

Service obtaining is based on selecting potentially adequate services according to the bundle’s requirements. For instance, if ((8:00am AND working_day) OR (10:00am AND non-working_day)) THEN (active sounding services) is one of the behavior rules defined by our bundle. So, the first step is looking for sounding services in the operations-at-home ontology. Secondly, these services are classified according to the criteria defined by our bundle to obtain a service ranking (see Sect. V-B). In this case, (i) the more noise the sounding service makes and (ii) the less energy it consumes, the better qualification it receives. For instance, the vacuum cleaner (80 dB), the Hi-fi system (70 dB), the TV (60 dB) and the washing machine (54 dB) are the top four according to the noise level; whereas the TV (0.065 KWh), the Hi-fi (0.07 KWh), the vacuum cleaner (0.67 KWh) and the washing machine (0.88 KWh) are the top four according to the energy consumption. Since the TV, the Hi-fi and the vacuum cleaner are the services on top of our ranking, they are activated. So, having applied explicit reasoning and datatype properties stored in the service ontology (energy consumption and noise level) a set of appropriate services has been selected. However, the ontology also maintains properties among instances and/or classes which must also be taken into account. For instance, properties like entails and inhibits (simultaneously or sequentially). These kind of properties enables explicit reasoning like, for instance, TV inhibits simultaneously Hi-fi system; or blinds down entails simultaneously lights on. So, as the TV and the Hi-fi cannot be activated simultaneously, the bundle may decide not to use one of them, or starting them in turns, depending on how it has been programmed. Finally, this service ranking can be modified by additional factors like, for instance, an intruder has been detected. In this case, the energy level is not considered, so, and with the aim of making as noise as possible, the vacuum cleaner, the Hi-fi system and the washing machine are simultaneously connected.

One step further, and part of our future work, is improving the Semantic OSGi platform to infer knowledge about the smart home. For this, the ontological structure

\(^1\)http://oscar.objectweb.org
\(^2\)http://protege.stanford.edu/plugins/owl/api/
\(^3\)http://www.ksl.stanford.edu
should also stores information about the use of services, so the framework can acquire extra knowledge regarding the use of sounding and lighting devices in the smart home. From this knowledge, it is feasible for the framework to reproduce the everyday habits of the inhabitants when they are on holidays, providing an “intelligent human presence simulator”.

VII. Conclusions and future work

The primary reference architecture in the OSGi specification, although not considered normative, is based on a model where a operator manages a potentially large network of service platforms. It assumes that the service platforms are fully controlled by the operator and used to run services from many different services providers.

In this scenario, we have shown that the actual service discovery mechanisms in OSGi is insufficient or, at least, only allows automation and interoperability in a restricted way. In the pursuit of a really open and interoperable residential gateway, we propose the semantic description and discovery of the services in the OSGi domain. At this respect, we have defined OWL-OS, a suitable sub-ontology inside OWL-S which allows making a simple semantic search of services based on a categorized structure. Despite we propose operations-at-home as the primary structure to classify the OSGi services, OWL-OS allows an OSGi service to be semantically described according to different ontological structures. This ontological structures would be downloaded on demand from the service provider. Finally, note the Semantic OSGi Framework enhance the OSGi standard, without breaking it; i.e. any non-semantic bundle can work properly within this framework, although it is not able to take advantage of the semantic reasoning for service obtaining.

Although a semantic classification and discovery of OSGi services is a step forward in itself, semantic filters improves the semantic searching mechanisms by supporting more sophisticated queries. The objective is to allow a bundle to express more characteristics of the desired service, i.e. not only the category but also some requirements for the service parameters (quality, response time, availability, reliability, etc). In this paper we have proposed OWL-QL as a language for semantic filtering, but OWL-QL has a more ambitious aim than being used in a local context. In fact, OWL-QL include several distributed features like server automatic selection of the knowledge bases, query-answering dialogs, etc. Other querying languages, like SPARQL, present similar characteristics. However, taking into account that the computational power of residential gateways is not usually high, we consider that creating a simpler language for semantic filtering based on LDAP-style is the best option. In fact, this is an ongoing work, in which we have also identified another important issue: introducing semantic Service Listeners as an analogy to the current Service Listeners. These new elements would be notified when a new service of a specific category is registered in the framework, or even when a new service of this category is registered with a concrete value for a service parameter.

However, specifying the requirements for the desired service is only the starting point. Ideally, from these service requirements, a reasoning engine will select the services which fulfill the requirements or the closets ones. To do so, we take as a point of reference previous works of our research group about applying semantic reasoning to develop TV recommenders [16]. Also in the field of service selection, it would be possible to supplement the Semantic Registry with different specialized software agents which takes into account other factors (ambient intelligence) to automate the service selection. For instance, a personalization agent would contribute to the decision by using an user profile storing historic information about preferred devices and services. Besides, a Semantic OSGi framework is ready to open its doors to context-aware computing, where a specialized software agent would be aware of the particular context (time, location, mood, etc) and reacts by discovering services which meet that context.

Last but not least, supporting the automation of OSGi services composition [17] is a clear benefit of the new Semantic OSGi platform. This automation opens the platform to ambitious applications which relay on the idea that different services at home usually form a pool that is committed to various home activities, such as energy control, security, health care, etc.

References


6http://www.w3.org/TR/rdf-sparql-query


Rebeca P. Díaz Redondo is an associate professor in the Dept. of Telematics Engineering at the Univ. of Vigo, where she received her Ph.D. in Telematics in 2002. She is a member of the Interactive Digital TV Lab, and her major research interests are interactive applications for TV as well as how they interact with the smart home environment.

Ana Fernández Vilas received her Ph.D. in Computer Science from the Univ. of Vigo in 2002. Since 1997, she is an associate professor in the Dept. of Telematics Engineering (Univ. of Vigo). She is engaged in web services technologies and ubiquitous computing environments, being a member of the Interactive Digital TV Lab.

Manuel Ramos Cabrér received his Ph.D. in Telematics from the Univ. of Vigo in 2000, where he is an associate professor in Telematics Engineering since 2001. His research topics are Interactive Digital TV concentrating on recommender systems, integration with smart home environments and interactive applications design and development.

José Juan Pazos Arias received his Ph.D. in Computer Science from the Dept. of Telematics Engineering the Polytechnic University of Madrid in 1995. He is the director of the Networking and Software Engineering Group at the Univ. of Vigo, which is currently involved with projects on middleware and applications for Interactive Digital TV.