Seamless Semantic Integration of Heterogeneous Devices and Services in the Cloud-enabled Smart Home

Ji Eun Kim[1,3], George Boulos[3], John Yackovich[3], Tassilo Barth[4], Christian Beckel[2], Daniel Mosse[3]
[1]Bosch Research and Technology Center, Pittsburgh, PA 15203, USA
[2]Corporate Sector Research and Advanced Engineering, Robert Bosch GmbH, Stuttgart, 70442 Germany
[3] Computer Science Department, University of Pittsburgh, Pittsburgh, PA 15213, USA
[4] Computational Linguistics Department, Saarland University, Saarbruecken, 66123, Germany
jieun.kim@us.bosch.com, gwf5@pitt.edu, jcy8@cs.pitt.edu, tbarth@coli.uni-sb.de, christian.beckel@de.bosch.com, mosse@cs.pitt.edu

Abstract

With the recent trend of ubiquitous access to embedded physical devices over the Internet as well as increasing penetration of wireless protocols such as ZigBee, we have observed increased attention to Cyber Physical Systems in the form of smart homes. These systems consist of sensors, devices and smart appliances that can be monitored and controlled remotely by human users and cloud services. However, the lack of a de facto communication standard for smart homes creates a barrier against the interoperability of devices from different vendors. We address this challenge by proposing a novel software architecture that seamlessly integrates heterogeneous protocol- and vendor-specific devices and services, while making these services securely available over the Internet. Our architecture is developed on top of the well-known OSGi framework and incorporates semantic knowledge about the smart home system by employing ontologies in order to provide semantic services. Based on these concepts, our APIs allow third-party developers to create new applications and drivers and integrate them into the deployed system during runtime. Furthermore, our new access control model accounts for scenarios specific to the smart home, including diverse users, device types, location, status and current time. For proof of our concept, we demonstrate the seamless semantic discovery of home devices during runtime by integrating several protocols including X10, Insteon, ZigBee and UPnP into a real test bed with devices and controllers. Using smart phones and the cloud services together with our proposed home gateway architecture, we further demonstrate the ease of integration of new applications and drivers. We show the scalability of our solution by evaluating the execution time for different numbers of devices and configurations of the semantic device discovery.

I. INTRODUCTION

Cyber Physical Systems (CPS) in the connected home provide automation capabilities that allow home owners to have more complete control over their home, and promote energy efficiency that allows them to save money on energy bills in the process. These smart home solutions integrate many devices having different capabilities such as intrusion detection, video surveillance, fire detection, patient health monitoring and entertainment. Many of these devices use different communication protocols that are mostly incompatible with each other. Examples of such protocols are power line communications of *X10* [1] and *Insteon* [2], wireless communications of *ZigBee* [3] and *Z-Wave* [4], IP-based *UPnP* (Universal Plug-and-Play) [5], SOAP based web Services on devices such as *DPWS* (Device Profiles for Web Services) [6], web of things [7] using Restful Web services and many more proprietary protocols from diverse manufacturers.

Although the market prediction and current technological trends look promising, we observed that no de facto communication standard exists in the smart home, which hinders seamless integration of different services (e.g., energy management, security) as a holistic smart home solution and limits the capabilities of future services available in the connected home. Therefore we propose a smart home software architecture that seamlessly integrates heterogeneous protocols and diverse device types used in home networks. Our architecture allows semantic service discovery of devices.

Current software application development trends with online distributions such as *Apple's AppStore* and the *Android Market* prove that third-party developers play an important role in software ecosystems. Similar to the smart phone industry, we anticipate that uniform access to devices and services of smart homes will allow third-party developers to contribute innovative applications to the smart home ecosystem. Therefore we provide application programming interfaces (APIs) and online distribution channels as part of our cloud infrastructure.

Furthermore, more devices connected to the home result in more diverse groups of users and services that interact with smart home solutions. This requires a new smart home access control concept. For example, the smart home system should not allow a utility company to access home owner's health-related data from medical devices in order to protect his/her privacy. To satisfy this requirement, we propose a new access control model and its implementation supporting our policy model for different users, permissions and multi-attributes including user roles, device type, location, status and time.

Our main contributions to CPS for the connected home are *1) seamless integration of heterogeneous devices and services with support of dynamic and semantic interoperability, 2) integration of cloud services* and *3) a new access control mechanism for the smart home* .

The remainder of the document describes related work in *Section II*, smart home stakeholders analysis and driving use cases in *Sections III* and *IV*, our proposed smart home architecture and design in *Section V*, the semantic integration in *Section VI* and the access control mechanism in *Section VII*. Our prototype demonstrator with the experimental setup is described in *Section VIII*, while *Section IX* presents the

experiment results. We discuss our findings and give an outlook to future work in Section X. We conclude our paper in Section XI.

II. Related Work

Prevailing home network standards use different communication media (e.g., power line, various RF bands), device address schemes of static or dynamic addresses, and different device discovery mechanisms. Low data rate protocols such as *X10* and *Insteon* do not provide device descriptions. *ZigBee*, *KNX* [8] and *EnOcean* [9] provide device descriptions in the form of standardized profiles specific to domains like home automation. The *UPnP* and *DPWS* standards use XML to describe device information. Semantic descriptions are not provided by any of these home networks. Security mechanisms of these protocols vary in behavior such as pairing mechanisms of devices, checksums of message payload, data encryption techniques using symmetric keys and so on. Due to these diverse and often incompatible mechanisms from different network standards, smart home systems in the market remain fragmented and provide only partial solutions addressing single protocols and subsets of devices.

Despite this heterogeneous nature of home network standards, existing smart home research generally assumes a homogeneous underlying architecture. *MavHome* [10], for example, predicts activities in a home and makes the home act as an intelligent agent providing optimal support for its inhabitants. In [11] the authors address the important role of context for smart home applications by providing an adaptive middleware and an API that provides context to applications. Other projects such as [12] and [13] aim at assisting end users to build their own individual smart home applications. While these visions are important for the success of smart homes, dealing with *device and service heterogeneity* is a crucial requirement for them to be realized.

Most similar to our architecture, *DOG (Domotic OSGi Gatway) [15]*, *Hydra [19]* and *Amigo* [20] aim at providing interoperability for home networks by creating a middleware platform based on *OSGi (Open Service Gateway initiative)* [14] and a semantic device model. Although these projects address some aspects of *dynamic interoperability*, our system attempts to unite all of them in a comprehensive and user-friendly way: *plug and play of heterogeneous devices plus services during runtime*, *extensibility* to new devices that are not foreseen during system development, and the use of a barcode reading functionality with smart phone cameras as a unique approach to improve *usability*.

Above all, our approach fully utilizes *cloud services* to increase coverage of the device discovery process and to extend the smart home functionality with new applications, drivers, and computationally intensive services. The authors [35] present an architecture to integrate cloud services and smart home networks. However, unlike us, they do neither implement the architecture in a prototype nor employ it directly in a user interface.

In [21], XACML (eXtensible Access Control Markup Language) [22] has been used to provide Role Based Access Control (RBAC) for the OSGi service environment. However, the authors do not address other attributes such as device type, location, device status and current time in our policy model except roles.

III. Smart Home Stakeholders analysis

In CPS, it is crucial to understand how different users/stakeholders interact with the system. We identify the following five stakeholders in order to elicit requirements of smart home systems.

- *Family*: A family consists of home owners, adults and/or kids. We assume that family members use smart phones or web-based devices to configure, monitor and control their smart home system remotely.
- *Visitor*: A visitor is a temporary user of services of the system. Examples include a dog walker, babysitter, friend, relative, etc.
- *Service provider:* In our context, service providers (e.g., utility companies and professional health care givers) are external users that periodically access the system remotely in the form of services.
- *Device vendor*: A device vendor is a manufacturer of home devices and provides driver software to the system. For example, a heating system vendor provides its proprietary device drivers pluggable to our gateway which connects its devices to the smart home.
- *Application developer*: An application developer uses a Software Development Kit (SDK) to develop specific-domain or cross-domain applications.

IV. Driving Use Case Scenarios

We identify driving use case scenarios, which address the heterogeneous nature of the smart home system. These use cases are developed based on the needs for different stakeholders illustrated in the previous section.

First, home owners often add home devices incrementally over time due to limited budgets and innovations in the market. Therefore, the combined process of *seamless plug and play of devices and discovery of semantic services* is one of the driving use case scenarios. For example, a home owner brings a new device (e.g., Insteon dimmer light) to the home. (S)he uses his/her smart phone camera (or other technologies such as NFC for RFID) to scan a barcode of the device. Then the smart home system provides a corresponding discovery wizard for the device installation. The smart home system connects to the application store in the cloud in order to download relevant software drivers and basic applications recommended by the system. Upon the home owner's approval the software can be deployed on the smart home system. Once this discovery process is completed, the home owner is able to access the discovered device remotely using his/her smart phones or other user interface devices.

Second, application developers should be able to *develop an application without detailed knowledge* about devices and protocol information. Instead they should rely on an *intelligent reasoning* capability. Assume that a developer wants to develop an application to provide a service to turn off all lights

on the first floor of a smart home if there is no one present for more than a certain period of time. Since every home has a different configuration of devices, the developer should be able to get a list of available devices with a certain level of abstraction. In our scenario, the developer would request a list of the available lighting devices using the API and get the corresponding device states to determine the occupancy status without knowing the device address, protocol and other configurations. For example, it should be irrelevant to the developer if a light device uses X10 or a different protocol, or what the device addresses of the lights in the first floor are.

Third, our smart home system connects many different devices, which can be used by diverse users. This requires our system to enable *suitable authentication and authorization for different users and devices*. We provide a mechanism that enables the home gateway to authenticate different roles of users and authorize their different requests. For example, we address the need to prevent a healthcare giver from accessing the home owner's non-healthcare related devices. In addition, our use case examples include access control that utilizes device status, device location and current time as access control attributes.

V. Smart Home Architecture and Design

A. High Level System Architecture

Our smart home system consists of a smart home gateway, which connects all different types of home devices and provides standard interfaces that are accessible through web services by using smart phones or any web-based user interface devices. Home devices are categorized into end-devices and controller devices. End-devices are sensors and/or actuators, such as temperature sensors, video cameras, motion sensors, smart appliances, plug-in modules or any devices that provide some direct smart home functionality. Controller devices do not offer specific services within the home. Instead they are gateway-type devices that allow the home gateway to communicate with end-devices. They typically connect via USB or Serial ports to the home gateway. Examples of these are modems such as the Insteon PowerLinc model 2413s (RS-232), X10 Active Home USB controller, and the Digi XStick Xbee USB adaptor. For protocols like UPnP, which work on top of IP, the controller device aspect is more implicit. The only hardware required to communicate with a UPnP device is a network card, which can be controlled to be the controller device.

The *home gateway* connects to a *cloud solution center* that holds an *application store* and provides *cloud services*. The *application store* manages the dependencies of software and home devices, and downloads software drivers and applications to the user's *home gateway*. Examples of *cloud services* are a video surveillance service requiring large amounts of storage, or an energy management service that needs more processing power than the *home gateway* provides in order to run complex algorithms (e.g., computation fluid dynamics optimization software).

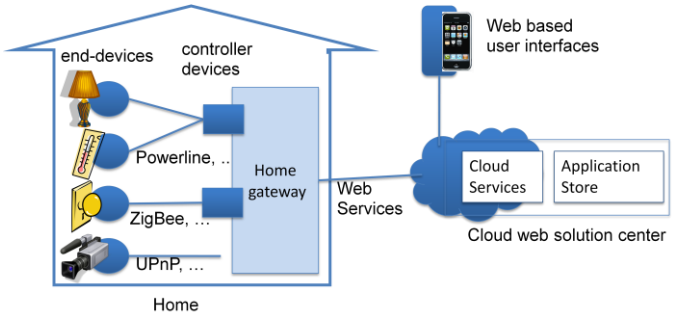

Figure 1: High Level System Architecture for Smart Home

B. Smart Home Gateway Architecture Overview

In a protocol-agnostic smart home environment such as our system, different types of devices and services can be added and removed during runtime, and services within the system need to discover the existence of other services with which they need to interact. To attack this problem, we use OSGi (Open Services Gateway initiative) as a basis for our framework. OSGi is a Java-based framework that enables services to be plugged in and out of the framework at runtime, and provides ways for plug-ins, referred to as OSGi bundles, to expose services to other bundles in the framework. Two major OSGi services intensively used in our software architecture are *device access service* and *event admin service* in order to support seamless plug and play of devices and services. This requires a mechanism to locate a correct driver for a device. The OSGi *device access service* facilitates this process through the device attachment phase. The OSGi *event admin service* is responsible for the communication between components while keeping implementations independent of one another. For example, if a new device is discovered, a corresponding driver bundle registers a new "*device registered*" event and releases it to the framework so that a new device object can be created elsewhere in the system.

As shown in the driving use cases, important architectural features for our smart home system are *interoperability* and *dynamic integration* of many types of drivers and devices developed by different vendors. For example, an X10-based lamp and Insteon-based lamp should be exposed with the same abstract web service interfaces. To support this paradigm we employ a device stack and a dynamic message-handling framework that processes commands in different ways given various contexts. In the following *Sections C* and *D*, we discuss our design of device abstraction and user request handling.

Figure 2 illustrates the software building blocks of our smart home gateway.

- *Core* provides interface definitions, utility methods, and common constants. Any declaration that would be shared between smart home gateway modules (bundles) is defined in the *Core*.
- *HIM* (Hardware Interface Manger) is responsible to detect any controller device that is connected to the system hardware. The OS reports events with information about devices to the HIM.
- *Deployment* connects to the application store in the cloud and forwards to it properties of controller devices discovered by the HIM in order to download proper drivers

for new devices. It manages software download and installation, and updates the home gateway profile in the cloud.

- *Device* implements our smart home device stack specification, which includes devices, drivers, device discovery and command invocation.
- *Configuration* keeps track of all system configurations for persistent objects as well as for ontology implementation.
- *Ontology* provides semantic knowledge about devices. Details are discussed in *Section VI*.
- *Database* implements a persistent repository for device status and information.
- *Scheduler* provides capabilities for scheduling tasks (e.g. device control commands), at a given time. Tasks may be recurring jobs.
- *Authentication* verifies a user's credentials and returns the roles for the user if authenticated.
- *Authorization* is responsible for allowing or rejecting users' requests by consulting the access control specification.
- *Message FW* implements our smart home message specification and message-handling framework specification. Details are discussed in the following section.
- *UI Proxy* provides web service interfaces to be used from within any client user interface devices such as smart phones or web browsers.
- *Web server* is a container for the *Restlet* web server [24], which is a Java-based Restful web server.
- *Controller Discovery* provides discovery of controller devices together with the HIM, Deployment and Device bundles. Details are discussed in Section C.
- *Device Discovery* manages different discovery mechanisms specific to each protocol. Details are discussed in Section C.
- *Notify* keeps track of all notifications to be delivered to users in the home gateway, which include the notifications of new devices discovered, new software installation required and alert messages.
- *Access Control* manages authentication and authorization for different requests.

In order to facilitate application development for third-party developers, we abstract these software building blocks for the home gateway implementation as APIs.

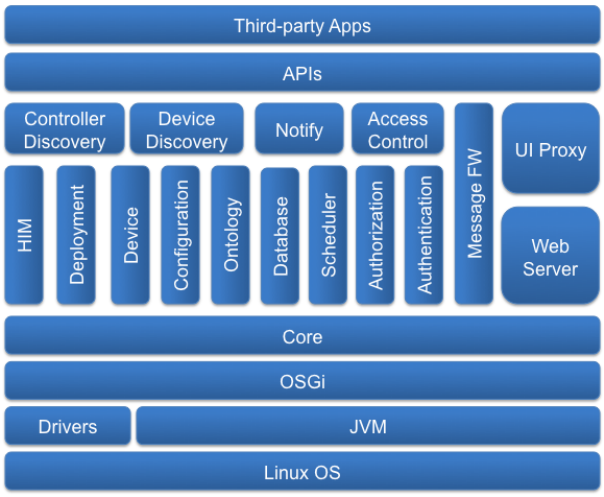

Figure 2: Core software building blocks of Home Gateway

C. Smart Home Device Stack and Discovery

As illustrated in Figure 3 (a), we introduce our smart home device stack consisting of the *ControllerDriver*, *Service Adaptor*, *SHDevice and SHService* implementation modules so that our architecture can allow uniform access of device services to diverse types of devices. Figure 3 (b) shows an example of the Insteon lamp device stack.

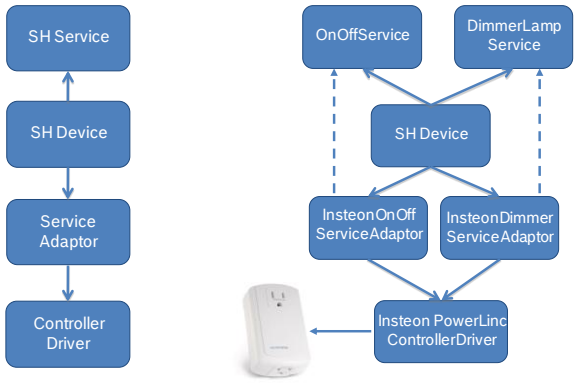

Figure 3: Device Stack (a) and an instance of Insteon device (b)

- *SHService* (*SmartHomeService*) represents a service provided by a device. It is represented in a protocol agnostic way. In (b) the Insteon device's two capabilities of turning on/off and dimming the lamp are represented as two protocol agnostic services of *OnOffService* and *DimmerLampService.*
- *SHDevice* (*SmartHomeDevice*) represents a proxy to a physical end-device. This object contains common home device information such as the device's URI (Universal Resource Identifier), device type (e.g., video camera), services supported by this device, protocol information and device location. We allow *SHDevice* to have properties that enable protocol-specific information representation.
- *Service Adaptor* is specific to a protocol implementation for *SHService*. In (b) *InsteonOnOffAdaptor* and *InsteonDimmerLampAdaptor* are Insteon protocol-specific implementations to turn on/off and dim the lamp device.
- *Controller Driver* represents the features available by the controller device, including the low-level handling of commands. It is responsible for either acting as the base driver or communicating with lower-level hardware drivers such as USB and Serial communication to send and receive information from the device.
- *InsteonPowerLinc ControllerDriver* is an implementation of the Insteon controller driver which sends commands through USB communication in (b).

1) New controller device discovery

We describe the controller discovery through an example (see Figure 4). Upon the plug-in of a controller device (such as the PowerLinc RS232), the *HIM* (Hardware Interface Manager) detects a new controller and sends controller device information to the *DeviceFactory* module. The *DeviceFactory* module, which is responsible for the life cycle of devices, creates a *Controller Device Object* with device information, and registers a "*new device*" event. The *OSGi Device Manager* detects this event and starts the matching process of *Controller Device Object* and the corresponding driver (*Controller*

Driver). If no driver is found, the *OSGi Device Manager* contacts *Driver Locator*, an OSGi Service interface for locating drivers for a device. Our implementation of *Driver Locator* calls *Deployment Manager*, which is responsible for downloading and installing the required software bundles from the *application store* in the cloud. The a*pplication store* manages software dependencies of the connected devices and other properties such as cost of software packages. Once the installation is complete, *Device Driver* registers a *Controller Driver* and *Device Manager* restarts the matching process, and *Controller Device Object* finally matches with *Controller Driver*.

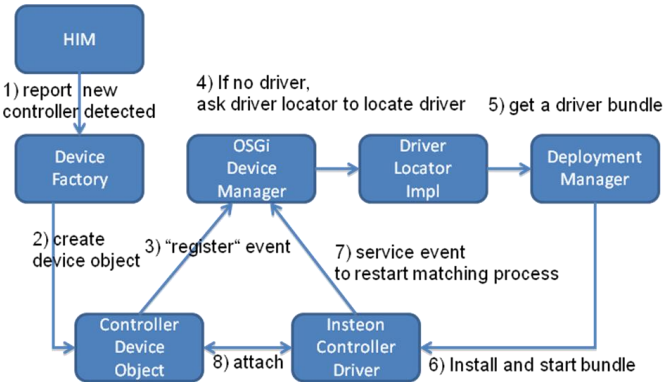

Figure 4: New controller discovery

2) New end-device discovery

Each protocol has its own device discovery mechanism. We categorize discovery mechanisms into three types: *manual*, *semi-automatic* and *automatic device discovery*. *Manual discovery* consists of device discovery that must be done entirely outside of the protocol specification. One example is X10, which provides no means for automated discovery of devices. In these instances the home gateway must derive all information about the device via user input or other techniques, such as barcode scanning. *Semi-automatic discovery* applies to protocols that support some level of device discovery using the protocol, but still require human involvement. Insteon is an example of semi-automatic discovery in that the user is required to either 1) input the hardware address of a device to discover it, or 2) press a hardware button on the device to initiate a discovery mode. The *ZigBee* protocol also falls into this category for home automation profiles. *Automatic discovery* does not require any user interaction unless the system wants to get additional (non-protocol) information specific to the user's environment. An example of automatic discovery is the UPnP protocol which detects new devices automatically and adds them automatically to the network system. In our system, we ask users to confirm the automatically-discovered device and ask them to provide semantic information such as location of the device and a friendly name for the device for automatic discovery.

Manual discovery and semi-automatic discovery rely on user interaction to add new devices to the smart home system. To mitigate the complexity of the discovery process, our architecture utilizes a camera on a smart phone device, which is capable of scanning the barcode of devices and sending the barcode to the *application store* which manages the discovery parameters specific to the device. Depending on the discovery parameters returned by the *application store*, we provide users with a corresponding wizard to guide users to easily add the new device.

At the home gateway, we introduce *Device Discoverer* modules which are specific to each protocol. Depending on the input from the UI, the home gateway calls the corresponding discoverer to add the new device.

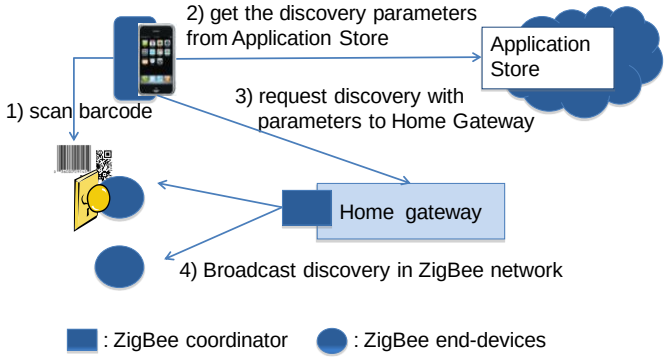

Figure 5: ZigBee End-Device Discovery Process

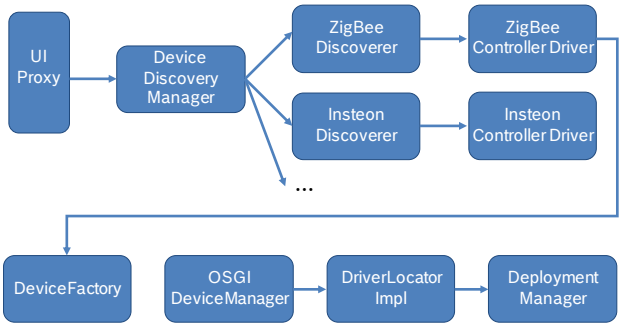

Figure 6: ZigBee End-device Discovery

Figure 5 and Figure 6 illustrate an example of new ZigBee end device discovery. In our scenario, the user does not know which discovery mechanism ZigBee supports. The user uses his smart phone camera to scan the barcode of the ZigBee end-device and gets the proper wizard to discover the new device. The smart phone sends the discovery request with parameters from the *application store* in the cloud to the home gateway. The home gateway finds the corresponding discoverer which sends a broadcast message to the local ZigBee network, which is equivalent to pushing a button on the controller, through *ZigBeeController Driver*. Once a new device is discovered, similar to controller device discovery process, *Device Factory* creates a new device object (*SHDevice*) and *OSGi Device Manager* matches the corresponding *Service Adaptors* for that device. *Service Adaptors*, in addition to matching to *SHDevice object* instances, must be assigned to a *Controller Driver* in order to execute commands.

D. Smart Home Services

Our architecture introduces three different types of services: 1) *Native Service* is a basic service representing functionality or capability of a device as a form of service 2) *Semantic Service* providing semantic query results and 3) *Composite Service* that is similar to rules to determine an action of commands to the system depending on the environment.

We adopt *Restful* web services having parameters encoded with *JSON (JavaScript Object Notation)*, and employed *Restlet web server*, which is a lightweight open source REST framework for the Java platform, for the smart home gateway. Popular user interfaces in our system architecture are smart phone devices or hand-held tablet mobile devices, which often require lightweight communication payloads. The *Restful web service* with *JSON* format fulfills this purpose well. A web resource represents a module within the framework that handles HTTP requests for a given URL. The only way a web service is expected to communicate with the framework is through a *smart home internal message*, which we discuss in the next section. Therefore, the implementation of the web services must create an internal message and enqueue it for execution in the framework to handle the request.

E. Message Framework

We introduce a smart home internal message (*SHMessage*) to communicate with different software modules in the home gateway. *SHMessage* can represent various types of framework messages, such as command invocation messages generated when the user wants to invoke a command to certain devices (e.g., turn on kitchen lights), or state change messages, such as data updates from a motion sensor if it detects motion.

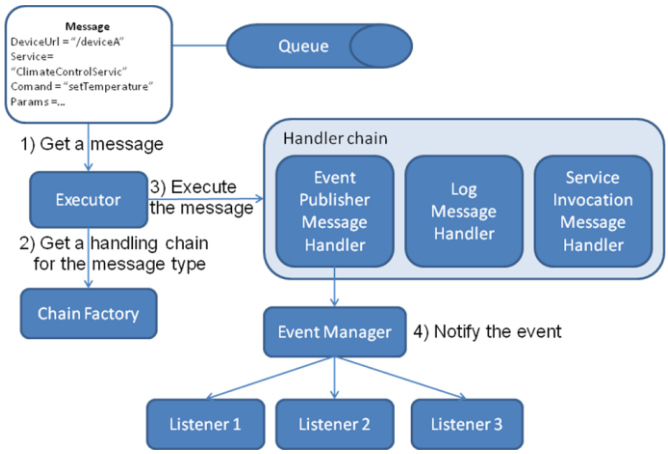

Figure 7: Message Framework

To provide a uniform way of accessing the *SHMessage*, the concept of *Message Helper* is provided. *Message Helper* adopts the *Decorator* design pattern [25] that encapsulates the *SHMessage* object and provides methods to access the fields of that *SHMessage*. *SHMessages* are handled in a centralized yet dynamic fashion using a mixture of *Chain of Responsibility* and *Strategy* design patterns. A *SHMessage* is handled by invoking a list of *Message Handlers*, each of which is a logical entity responsible for executing a specific task such as raising an event throughout the system or logging the message. The list of *Message Handlers* in this chain is constructed on-the-fly by the *Chain Factory* module, which uses the message type to derive the purpose of the *SHMessage* and how it should be handled. In this way we provide an extensible framework where handlers may be added and removed dynamically. In addition, our architecture also provides a *Publish/Subscribe* pattern for a *Message Handler* and the ability to generate the handling chains of new message types. Thus, we allow third party developers to add a listener for a certain event to create their applications by hooking on to specific events in the system. For example, a developer can create an app to automatically switch the speaker input to TV when it detects an event of TV turn- on.

Figure 7 illustrates how our message framework works for a *native service*. A web service creates a *SHMessage* with priority and timestamp to invoke an action on a device (in this example setting the temperature value) and inserts it in the queue. The *Chain Factory* then constructs the chain of *Message Handlers* that will handle the message. In this example, *Event Handler* raises a new event to any modules that are interested in being notified of this particular type of occurrence (the set temperature command). Following this the *LoggingHandler* handles the message, which will record the execution of the message in the log. Finally the *ServiceInvocationHandler* carries out the message's purpose by sending the message's intended command to the appropriate *ControllerDriver*.

VI. Semantic Integration

Our architecture incorporates semantic knowledge to support our second driving use case, in which third-party developers should be able to build applications independent of the concrete environment in which they are deployed. We discuss the main requirements that semantic knowledge for the smart home system has to fulfill and our approaches to provide solutions for the elicited requirements in this section.

A sufficiently complex and adaptable abstraction layer must include explicit semantic knowledge. Consider a typical yet simple example application such as “turn on all lights on the floor the user entered”. The application will use an abstract “lighting” super-type to retrieve all possible lamp devices. Moreover, based on the specified device location and the contextual knowledge (e.g., kitchen is on the first floor), it has to infer that devices located in the kitchen are also located on the first floor. After analyzing such examples, we consider the following properties essential for the semantic knowledge in the smart home domain and its potential use by third-party applications:

- P1: Taxonomy of device types, together with capabilities of devices, to allow for abstraction and on-the-fly semantic device retrieval.
- P2: Contextual knowledge, such as location and time, required for context aware applications.
- P3: Reasoning on the knowledge base to infer implicit knowledge from a minimum number of explicit facts.
- P4: APIs to retrieve and modify semantic knowledge.

Knowledge as defined by P1 and P2 has already been captured in domain ontologies created by related projects such as *DOG*, *Amigo* and *Hydra*. Ontologies translate a domain of interest into a set of concepts, properties and relations governed by strictly formalized semantics. The latter allow for automatic inference operations which can draw implicit conclusions from the explicitly stated knowledge. We use the Web Ontology Language (OWL) to describe our smart home model. The theoretical background of OWL is based on the Resource Description Framework (RDF) [26] and Description Logics (DL). Extensive toolkits, APIs, and efficient automatic reasoning engines exist for RDF/OWL.

Efficiency for automatic reasoning engines is especially important: logical reasoning on the domain ontology imposes trade-offs between the expressivity of the knowledge formalism and the performance needed for situated applications. We address this issue and thus solve P3 in conjunction with P4 by providing a unified interface to reasoning engines of differing expressive power, *OWLIM* [27] and *Pellet* [28].

OWLIM is a very fast and scalable storage mechanism, which is restricted to subsets of *OWL-DL*. For example, only "0" or "1" are allowed as values for cardinality restrictions. Pellet, on the other hand, offers full *OWL-DL* reasoning. It is notable that all of the domain ontologies we considered rely at least partly on OWL-DL constructs. We use *Pellet* to guarantee their consistency and to achieve a full static classification of device instances. *OWLIM* handles the potentially large number of dynamic device retrieval requests, for which weaker semantics are sufficient, as well as the storage of the actual *RDF* triples.

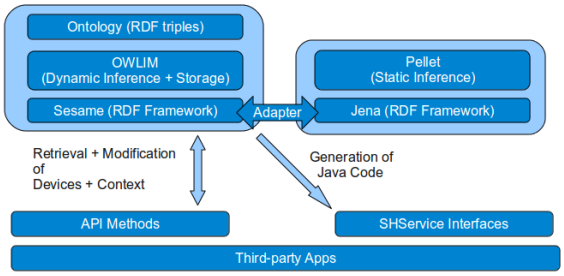

Figure 8: Architecture of Semantic Integration

As illustrated in Figure 8, our API internally employs two *RDF* frameworks in parallel. *Sesame* [29] serves as an interface to *OWLIM* and as the main data management layer. For more expressive inference operations (like consistency checking), the ontology facts are translated into the data structures of *Jena [30]*, another *RDF* framework and the interface to *Pellet*. *Jena* also offers a powerful rule system which might prove useful in the future.

To keep the API consistent with the domain ontology model, a code generator transforms the services defined by the ontology into Java interfaces which *SHService* in our device stack (see *Section V*) has to implement. In general, by keeping the methods to access the knowledge based on an abstract level, there should be no need for third-party developers to be proficient with *RDF/OWL*-related technologies.

As already mentioned, extensive work has been done and is still ongoing on building comprehensive domain ontologies. We thus refrain from creating our own and instead decide to use the ontology published by the *DOG* project. *DogOnt* focuses on the home environment, different network protocols, offers a device taxonomy as well as context concepts, and is sufficiently well-documented to allow for rapid adaption.

Figure 9 shows an excerpt from our adapted domain ontology which supports the example application presented above: Transitive location properties (*isIn* or *contains)* enable contextual reasoning. The concrete lamp device instance is grounded in the multi-level device taxonomy and connected to the services it offers, which allows for abstract semantic device retrieval.

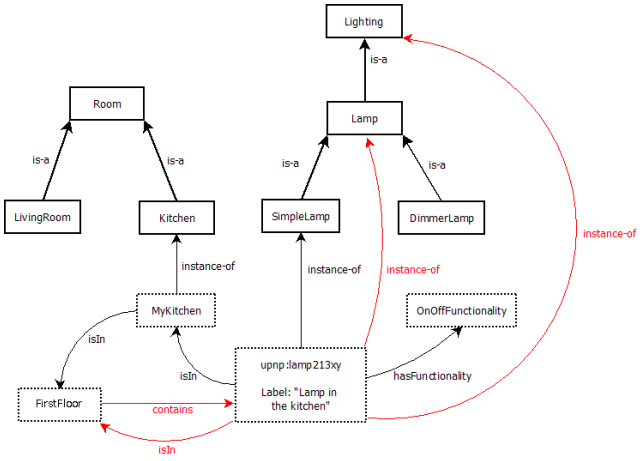

Figure 9: Example Ontology for Smart Home

VII. Access Control

Based on our stakeholder analysis in *Section III*, our solution allows different users to define different policies in the smart home security solution. Our policy aims to control access on the home gateway.

A. Policy Model

Our policy model includes different *roles* and a user can be a member of many roles. Some examples of roles would be administrator, adult or kid. The policy model has four *permission types*: user management, device management, controlling device, and monitoring device. *User management* permission represents the ability to add, remove and modify user roles. *Device management* permission represents the ability to add, remove and modify devices. *Controlling device* permission represents the ability to issue a command to a device such as opening a door lock or turning on a light. *Monitoring device* permission represents the ability to get state information from sensor devices, which includes getting temperature feed, getting heart monitor feed, video streaming and so on.

Our main goal for a smart home policy model is for it to be capable of representing fine grained access policies similar to the ones we enforce in real life. We have identified four *attributes* necessary to express a very wide range of rules when combined. The *device type* attribute limits the access to a given device type (e.g., all light devices). The *location* attribute limits or enables access to devices in a specific location, and also may limit access to a device by putting restrictions on the user's location at the time of access. *Device status* limits the access to a device by restricting the status of the device at that time. The *time* attribute limits the access to a device for a (possibly recurring) period of time. The policy maps a user *role* with *permission* on a device that has a collection of given *attributes*.

The *device type* attribute is important in the smart home domain, because our system interacts with different roles of users (e.g., utility company, healthcare provider, etc) from outside the home to allow limited access. Also, a family usually consists of different age groups which often require different access policies for safety or other purposes. The

location attribute is useful for more fine-grained control. A house resident can limit the access of some places within home for visitors, for example a visitor may be welcomed to control devices in the living room, but not in the bedroom. *Device status* is added to our attributes based on our driving use cases concerning future smart grid integration. One possible scenario in the future smart grid is that the service provider should not turn off the laundry machine while the machine is on. The *time* attribute provides users with a flexible way to control access during different times of the day, like common parental control schemes. For example, parents may only allow their kids to access entertainment devices (e.g., television) during specified time durations.

B. Access Control Architecture and Design

To realize access control within our architecture, we develop a hybrid approach of *OSGi User Admin* service and *eXtensible Access Control Modeling Language (XACML)*. While the *OSGi User Admin* service is moderately expressive, it does not allow us to express the multitude of variables introduced in our policy model. Thus, we use the OSGi *User Admin service* to represent subjects (user's roles) and assist in our authentication process only, and use *XACML* to specify and enforce policy given subjects.

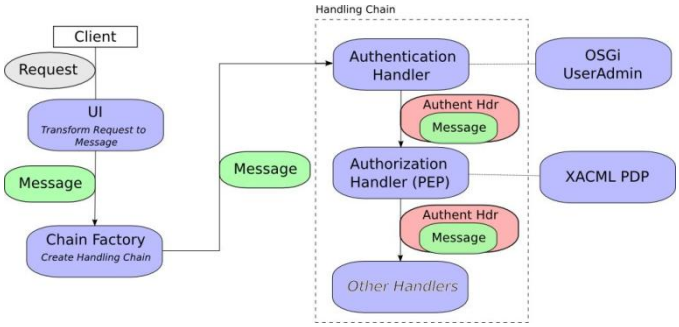

Figure 10: Access Control Design Concept

Figure 10 illustrates how access control concepts are implemented in our architecture.

- *UI Proxy:* The UI proxy modules are responsible for creating internal messages (*SHMessage* discussed in Section V) based upon user requests. For each message, it must wrap the message in an authentication header. The header contains a flexible form of user credentials such as username/password, or an authentication token. Then message is then enqueued within the message framework for execution.
- *AuthenticationHandler* is responsible for checking authentication of users. It consults the *OSGi User Admin service*, which manages a database of users and roles. If the credentials provided in the message are valid, the *AuthenticationHandler* will pass an authorization object into the same message in order to request authorization process to *Authorization Handler*. If not, the message will be discarded as an invalid message.
- *AuthorizationHandler:* The *AuthorizationHandler* fulfills the role of *Policy Enforcement Point (PEP*) as specified by *XACML*. It checks the authorization object of the message (representing the subject), the intended action, and the object of the message, as well as other attributes such as location and time, and submits an *XACML* query based on this information to *Policy Decision Point (PDP*). *PDP* evaluates the *XACML* policy to see if the intended action is indeed an authorized one. If the policy allows the request, the *PDP* will refer the message to the next handler for further processing. If the policy does not permit the action, the message is denied and dropped.

Figure 11 is an example of *XACML* policy used in our demonstrator. The policy defines that kids are only allowed to turn off entertainment devices after 19:00.

```
<Rule RuleId="Rule_6" Effect="Permit">
<Target><Subjects><Subject>
<SubjectMatch MatchId="string-equal">
<AttributeValue DataType="string">Kids</AttributeValue>
<SubjectAttributeDesignator DataType="string" AttributeId="group"/>
</SubjectMatch>
</Subject></Subjects>
<Resources> <Resource>
<ResourceMatch MatchId="string-equal">
<AttributeValue DataType="string">Entertainment</AttributeValue>
<ResourceAttributeDesignator
DataType="string"AttributeId="domain"/></ResourceMatch>
<ResourceMatch MatchId="string-equal">
<AttributeValue DataType="string">On</AttributeValue>
<ResourceAttributeDesignator DataType="string" AttributeId="status"/>
</ResourceMatch></Resource></Resources>
<Actions> <Action>
<ActionMatch MatchId="string-equal">
<AttributeValue DataType="string">control-device</AttributeValue>
 <ActionAttributeDesignator DataType="string"AttributeId="action-
id"/></ActionMatch>
 </Action></Actions> </Target>
<Condition FunctionId="time-greater-than-or-equal">
 <Apply FunctionId="time-one-and-only">
 <EnvironmentAttributeDesignator DataType="time" AttributeId=" current-time"/>
</Apply>
<AttributeValue DataType="time">19:00:00</AttributeValue>
</Condition>
</Rule>
```

Figure 11: Example of XACML Policy (shortened)

VIII. Proof of Concept and Experimental Setup

We demonstrate our smart home architecture and design by showing the discovery of selected new devices and services and use services with semantic information. Our access control concept is also demonstrated with example policies covering selected stakeholders, devices and services. In addition, we evaluate the execution time of remote service calls for different configurations.

In detail, the home gateway prototype is built on top of a Linux-based OS and supports the *ProSyst* [31] and *Equinox* OSGi implementations. It integrates *X10*, *Insteon*, *ZigBee* and *UPnP* protocol based devices. The types of end-devices are various: we have integrated many different types of sensors such as motion sensor and water leak sensor, adapters for the home appliances like on/off adapters and dimmer lamp adapters, RF transmitters to control TV, and video surveillance cameras. For the user interface devices, *iOS* devices such as *iPhone* and *iPad* and *Android* phones are used. The cloud solution center currently supports the *application store* component, which is deployed in our private cloud. Figure 12 shows a picture of our demonstrator with selected

devices.

Our prototype system demonstrates three driving use cases mentioned in Section IV. User connects X10, Insteon and ZigBee controller devices via USB to the home gateway, the home gateway detects the presence of new controller devices and creates new controller device objects as described in Section V. UPnP drivers discover new UPnP video camera when the camera is plugged into the network and sends a notification to the user's smart phone. For other end-devices, our demonstrator uses iOS or Android native apps to start discovery of end-devices by scanning the barcode of the devices. The devices are discovered as discussed in *Section V*. In our demonstrator we also provide an instance of our policy model in order to show access control for different users, permissions and attributes we discussed in *Section VII*.

Users can remotely access discovered devices using smart phones from anywhere based on the access control policy. Our example policy defines all permissions to the users having the *admin* role, while it restricts accesses for the users belonging to the *kid* role. Example policies include "kids are allowed to control all lighting devices", "kids are allowed to control all devices in the guestroom" and "kids are allowed to turn off entertainment devices after 7pm". We use *a permit-override algorithm* which disallows the request as a default behavior and changes the response to authorize the request if the request matches any policy which permits the request.

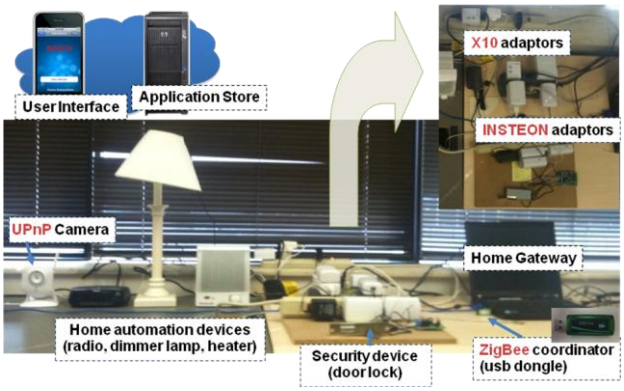

Figure 12 Picture of Demonstrator

To evaluate the real-time behavior of our prototype, we measure the difference between the time the gateway receives a service call from the remote user interface and the time it sends the command out to the controller device. The execution time includes only time spent in the gateway. It excludes the delay caused by network communications, in particular the remote call to the gateway and the device protocol overhead. We compare the execution time for different numbers of devices *n* (1, 5, 20, 50) and additionally three different settings: 1) semantic retrieval of the device given a particular location using *Jena* for ontology access, 2) the same with *Sesame/OWLIM* (from here on shortened to *Sesame*), 3) direct access to a lamp by its identifier without semantic retrieval (*Nosemantic*).

For each experiment, we restart the gateway and send 50 remote service calls (turn on and off a lamp device).

Our hypothesis is that the system should be scalable with constant execution time regardless of the number of devices discovered. This means the average execution time is not influenced by the number of devices, and the overhead of the semantic device retrieval should stay constant for a growing number of devices. We expect the average execution time of *Sesame* to be lower than that of *Jena*.

IX. Experimental Results

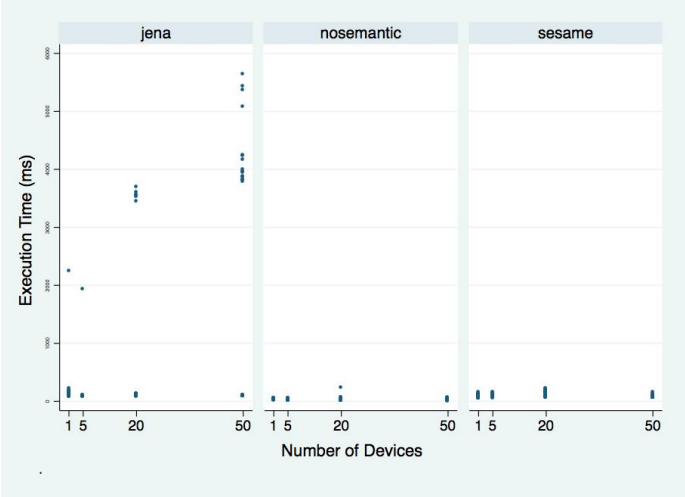

Figure 13: Execution time of service calls to lamp device

Configuration	Mean	Std	Min	Max
Jena	558.33	1286.65	83	5641
Nosemantic	25.45	20.34	4	235
Sesame	89.67	34.44	52	215

Table 1: Statistics for execution time (ms) of service calls to lamp device

Figure 13 gives an overview of the execution times of all service calls for the different experimental settings, while Table 1 presents the corresponding statistics. For *Nosemantic* and *Sesame*, we observe no significant differences in the distribution of the execution time for different device numbers. The average execution time with *Sesame* is 256% higher (overhead of the semantic device retrieval) than without semantic support, but the variation is relatively lower. When the semantic services are provided by *Jena*, execution time is higher on average. Also, the execution time varies over the number of service calls. This is mainly due to the first few service calls: After 1-10 calls a query caching mechanism seems to set in and reduce the delay for further calls to a level comparable to *Sesame*. The proportion of service calls with high execution time depends on the number of devices.

X. Discussion and Future Work

The results of the execution time experiment confirm our hypothesis: The number of devices connected to the gateway has no effect on the average execution time of a remote service call, which means the system scales well with regard to the size of the smart home. This also applies to semantic device retrieval, with a caveat: The selection of the interface to the ontology is crucial, since it might not only introduce a constant overhead (as for *Sesame/OWLIM*), but also slow down the service calls proportionally to the number of devices (*Jena*). Thus, as explained in *Section VI*, we will use *Jena* only for consistency checking and initialization of the ontology, while *Sesame/OWLIM* will handle semantic device retrieval

of calls.

The smart home gateway connects various safety-related devices and the misuse of connected devices may harm the quality of devices and services. For example, a simple service to increase a temperature by one degree every second will end up the failure of devices unless the device itself supports suitable protection mechanisms. Therefore, the protection mechanisms for such use cases should be addressed in our existing architecture. In addition, we envision that different applications that access the same devices with different intents at the same time can be deployed on the gateway. Therefore, we will provide a mechanism for conflict detection and resolution of different intents in the architecture in order to provide safe and reliable smart home systems.

In addition, we believe usable and useful interfaces for home owners to specify access control policies to be an important improvement to be investigated, as well as effective feedback to users who cannot access services currently but may be able to do so under certain conditions.

XI. CONCLUSION

This paper proposes an extensible architecture for highly heterogeneous smart home systems to enable dynamic integration of devices and semantic services leveraging mobile devices and cloud services.

Our prototype is based on the OSGi framework and semantic technologies and illustrates seamless integration of devices connected through X10, Insteon, ZigBee and UPnP protocols. Restful web service APIs on the home gateway enables remote configuration, monitoring and control with mobile devices as well as interaction with cloud services. We believe our developed access control policy and implementation provide robust control to be utilized by home owners for the types of users they allow to access various services within the home. In the future, we see additional applications for the barcode functionality and the authentication of end devices to further improve the mediation ability of the gateway system.

ACKNOWLEDGMENTS

We thank researchers in Bosch Corporate Research and Prof. Adam Lee in the University of Pittsburgh for discussing and reviewing our research.

REFERENCES

[1] X10 PowerHouse, Tech. Rep., 2001, Available http://www.x10.com
[2] Insteon The Details, Smart Home Technology, 2005, Available: http://www.insteon.net/pdf/insteondetails.pdf
[3] ZigBee Home Automation Public Application Profile, ZigBee Alliance, 2010, ZigBee Document 053520r26
[4] Z-Wave Alliance, Available: http://www.z-wavealliance.org/
[5] UPnP Device Architecture v.1.1, UPnP Forum, Available: http://www.upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.1.pdf
[6] Device Profile for Web Services (DPWS) Specification, Available: http://docs.oasis-open.org/ws-dd/ns/dpws/2009/01
[7] D. Guinard, V. Trifa, T. Pham, O. Liechti, "Towards physical mashups in the Web of Things", *Proceedings in 6th International Conference on Networked Sensing Systems (INSS), 2009 , ISBN 78-1-4244-6313-8, pp 1-4*
[8] KNX Specifications, Available: http://www.knx.org
[9] EnOcean Equipment Profiles (EEP) v2.0, EnOcean Alliance, 2009,
[10] Diane J. Cook et al, "MavHome: An Agent-Based Smart Home", *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications, 200., pp 521 - 524*
[11] M.Huebsher, J.McCann, "Adaptive middleware for context-aware applications in smart-homes", *Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing, 2004*
[12] F.Kawsar, T.Nakajima, K. Fujinami, "Deploy spontaneously: supporting end-users in building and enhancing a smart home", *Proceedings of the 10th international conference on Ubiquitous computing, 2008, ISBN: 978-1-60558-136-1, pp 282-291*
[13] T. Zhang, B.Brugge, "Empowering the user to build smart home applications", *Toward a Human Friendly Assistive Environment, IOS Press 2004, pp 171- 183*
[14] OSGi Service Platform Core Specification v 4.2, OSGi Alliance, Available: http://www.osgi.org
[15] D. Bonino, E. Castellina, F. Corno, " The DOG Gateway: Enabling Ontology-based Intelligent Domotic Engironments", *IEEE TRANSACTIONS ON CONSUMER ELECTRONICS. vol. 54/4 ISSN: 0098-3063, pp. 1656-1664.*
[16] "The Eclipse Equinox project", Available: http://www.eclipse.org
[17] D. Bonino, F. Corno , "DogOnt - Ontology Modeling for Intelligent Domotic Environments", *The Semantic Web - ISWC 2008, Lecture Notes in Computer Science, 2008, Volume 5318/2008, 790-803*
[18] "OWL 2 Web Ontology Language Document Overview". W3C, 2009. Available: http://www.w3.org/TR/owl2-overview/
[19] M. Eisenhauer, P. Rosengren, P. Antolin, "Development Platform for Integrating Wireless Devices and Sensors into Ambient Intelligence System", *IEEE SECON Workshops (June 2009)*
[20] M. Janse, P. Vink, M. Georgantas, "Amigo Architecture: Service Oriented Architecture for Intelligent Future In-Home Networks", *Constructing Ambient Intelligence, Communications in Computer and Information Science, 2008, Vol. 11, Part 7, 371-378*
[21] G. Ahn; H. Hu; J. Jin, "Towards Role-based Authorization for OSGi Service Environments", *Proceedings of IEEE International Workshop on Future Trends of Distributed Computing Systems, 2008, pp23-29*
[22] OASIS eXtensible Access Control Markup Language (XACML), Available: http:// www.oasis-open.org/committees/xacml/
[23] OSGi Service Platform Compendium Specification v 4.2, OSGi Alliance, Available: http://www.osgi.org
[24] Restlet: The leading RESTful web framework for Java, Available: http://www.restlet.org/
[25] E. Gamma, R. Helm , R. Johnson, J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", *Addison-Wesley Professional; 1994 , ISBN-10: 0201633612*
[26] "RDF: Resource Description Framework", W3C, Available: http://www.w3.org/RDF
[27] OWLIM: Semantic Web Standard, W3C, Available: http://www.w3.org/2001/sw/wiki/Owlim
[28] E. Sirin, B. Parsia, B.Grau, A. Kalyanpur, Y. Katz , "Pellet: A practical OWL-DL reasoner", *Web Semantics: Science, Services and Agents on the World Wide Web, Vol. 5, Issue 2, June 2007, Pages 51-53, Elsevier*
[29] Sesame: RDF Schema Querying and Storage, Available: www.openrdf.org/
[30] B. McBride " Jena: a semantic Web toolkit", *IEEE Internet Computing, 2002, vol.6, pp 55-59*
[31] "ProSyst OSGi services", http://www.prosyst.com
[32] Sun's XACML Implementation, Available: http://sunxacml.sourceforge.net/
[33] "Challenges in Access Right Assignment for Secure Home Networks", H.Kim et al, *2010, Proceedings of the 5th USENIX Workshop on Hot Topics in Security*
[34] "Research and design architecture of cloud architecture for smart home", Wei Zhiqiang, Qin Shuwei, Jia Dongning, Yang Yongquan; *2010, 2010 IEEE International Conference on Software Engineering and Service Sciences (ICSESS)*