5

Ontology Driven Smart Space Application Development

M. Mohsin Saleemi, Natalia Díaz Rodríguez, Espen Suenson, Johan Lilius and Iván Porres

Department of Information Technologies, Turku Centre for Computer Science (TUCS), Åbo Akademi University, Turku, Finland; e-mail: johan.lilius@abo.fi

Abstract

This chapter presents an approach to create an abstraction layer and appropriate tools for rapid application development for *Smart Spaces*. The proposed framework is described together with the overall process for application development. An approach of how to integrate OWL-S grounding into the agents is described for allowing service discovery and composition. We also show results of a case study implementation to illustrate the functionality of the proposed framework. This case study shows that interoperability can be realized by agents that describe information about themselves using some common ontology.

Keywords: smart space, ontology, interoperability, application development, OWL-S.

5.1 Introduction

While the Semantic Web envisions more well-structured data enabling new possibilities for the Internet, the semantic concept is also being adopted into other areas. One of these areas is the *Smart Space*, which, although very similar, comes with a different set of restrictions, challenges and possibilities.

S.F. Pileggi and C. Fernandez-Llatas (Eds.), Semantic Interoperability: Issues, Solutions, and Challenges, 101–125.

The reason is that *Smart Spaces* heavily depend on heterogeneous devices, systems and services and need to be made seamlessly interoperable to be used effectively in a device, vendor and domain independent manner. The diversity of devices and standards raises the problem of enabling interoperability of different services within different devices.

A *Smart Space* is an abstraction of space that encapsulates both the information in a physical space as well as the access to this information, such that it allows devices to join and leave the space. In this way, a *Smart Space* becomes a dynamic environment whose identity changes over time when the set of entities interact with it to share information between them. Moreover, as *Smart Spaces* provide information about a physical environment which is shared with inherently dynamic applications, ubiquitous ambient services should adapt user preferences in each particular context. These ubiquitous applications employ a range of different devices in addition to mobile phones to provide a set of innovative services that are both social and personalized. This requires more advanced methods for data handling and understating and approaches for data exchange and communication across heterogeneous sources. Furthermore, there are several other issues to be solved such as interoperability issues, common application development platforms and the development tools for rapid application development.

The main research problems we are dealing with are how to develop fast and rapid applications for *Smart Space* using the traditional Object Oriented (OO) programming approach and how to achieve interoperability in these applications. The objective is to develop a generic and comprehensive interoperability solution that enables the devices and applications from different domains to communicate with each other and construct a scalable smart network of diverse devices.

In this chapter, we present our solutions for the given research problems. We proposed and developed user level tools, which make use of Web Ontology Language (OWL). OWL not only allows structuring the *Smart Space* content in terms of high-level programing language concept of classes but also specifies relations between the classes and their properties. Hence, entities interacting with *Smart Space* can consume and produce content according to high level OWL ontology terms. As dealing with ontologies could be very difficult for the programmers and end-users, we developed a user-level tool that generates ontology API by mapping OWL ontology concepts into Object Oriented programming language concepts. This enables application developers to create innovative *Smart Space* applications using traditional Object Oriented programming concepts without worrying about

the complexity of OWL. Moreover, service discovery and composition under "unchoreographed" conditions [8] can be tackled with existing semantic web services and our proposed tools [21], which provide a solution to make easier for the programmer the combination of services differently implemented.

This chapter is organized as follows. Section 5.2 gives an overview of the related work on ontology based infrastructures. Section 5.3 gives an overview of the *Smart-M3* concept, a particular implementation of *Smart Space*. It also describes the ontology point of view for this approach and describes the tools developed for support of Knowledge Processors (KP) creation. Section 5.4 shows a case study using our framework to give proof of concepts. Section 5.5 gives a structured view of *Smart Space* and the interaction between KPs. Section 5.6 describes a service ontology and its function. It also proposes an OWL-S Python binding for service composition. Finally, Section 5.7 makes some conclusions and depicts directions for future work.

5.2 Related Work

Context-aware computing research shows a large number of context-aware systems and approaches for application development. Ontology context modeling differentiates according to simplicity, flexibility, extensibility, genericity and expressiveness [9]. Since 2004, many ontology-based systems have been developed. CoBrA and SOCAM are some examples, which use their own OWL-based approach for context processing while others like Context Managing Toolkit describe context in RDF. CoBrA [11], as agent-based infrastructure for context modeling, context reasoning and knowledge sharing, provides techniques for the user's privacy control, while Soupa and CoBrA-Ont provide some of their ontologies. SOCAM [15] introduces another architecture for building context-aware services focused on information sensing and context providers using a central server. All these systems use SQL to access the central database. In contrast, we propose RDF as a more efficient way and to restrict the queries to a smaller set of statements.

In [20], operating system concepts include context-awareness. Quaternary predicates are used for information representation, in which the fourth one is context-type. DAML + OIL is employed, as well as an MVC model. These and other projects as [28] focus basically on creating ontologies for context-representation. However, we intend to build a framework for creating context-aware development of services or applications based on the semantic architecture (which, in contrast, has a blackboard architecture).

Context Toolkit [13] is another case that enables application development through reusable components. However, its attribute-value tuples, not being meaningful enough, make application programming restricted. Another example is HIPPIE [18], which utilizes existing users' information with an awareness system to distribute context information to the users' devices. For compensating its lack of handling interaction, it was combined with NESSIE [19] which added event based awareness but still lacked semantic information description. When rules must be specified in the *Smart Space* in OWL, an OWL-Script language [25], prototyped in our group, can be considered.

When it comes to deploying services, there are several alternatives. The main ones consist of OWL-S and WSMO (Web Service Modeling Ontology). The grounding in OWL-S provides the details of how to access the service mapping from an abstract to a concrete specification of the service. WSMO approaches for grounding use mapping to XML with SAWSDL (Semantic Annotations for WSDL and XML Schema) while OWL-S utilizes WSDL (and SWRL for rules) and possibly XLST transformations. However, both groundings reduce in the end to the use of WSDL to where both OWL-S and WSMO services must be mapped for a concrete specification [7].

There are different approaches and architectures that address the issue of service composition. These approaches can be classified using several service composition features such as automatic composition [17], semi-automated composition [23], end-user interaction [24], service specification language [14], etc. In [9], the authors give a comparison of different service composition approaches. A middleware solution for end-user application composition is provided in [12]. Other approaches of flexible service composition in mobile environments are described in [10] and [26]. While existing research efforts deal with theses issues separately, there has been very limited work in ubiquitous service compositions in smart environment. In [27], the authors proposed a system consisting of a middleware and user-level tools that enable the end-users to combine, configure and control the services using their smart home devices.

Aiming at facilitating the creation of smart services we can observe that diverse technology-specific frameworks exist, but none of them results in a rapid and functional application programming tool. Comparing with previous systems, we describe an approach that tackles the challenge of context-aware ubiquitous computing using automated ontology code generation (Python and C) giving complete control over ontologies. The communication with the *Smart Space* is therefore encapsulated for the developer.

5.3 Smart-M3 Architecture

A concrete implementation of *Smart Space* is Nokia's *Smart-M3* [5], a Multi-domain, Multi-device and Multi-vendor (M3) platform consisting of a space based communication mechanism for independent agents which communicate implicitly by inserting and querying information in the space. *Smart-M3* is an open source, cross-domain architecture where the central repository of information, Semantic Information Broker (SIB) is responsible for information storage, sharing and management through the *Smart Space* Access Protocol (SSAP). SSAP provides the KPs access to the *Smart-M3* space by means of the operations: *Join*/*Leave* the *Smart-M3* space, *Insert*/*Remove* information from the SIB, *Update*, *Query* and *Subscribe* to changes.

Entities called Knowledge Processors (KPs) implement functionality and interact with the *Smart Space* by inserting/retrieving/querying common information. An application is constructed by aggregating normally several KPs where each performs a single task and communication does not happen device to device but through the *Smart Space* central repository (SIB). One device can host any number of different KPs. The application is constructed by the composition of several KPs where each KP performs a specific task. The application design in this approach differs from the traditional single device control-oriented application.

The information level interoperability provided by *Smart-M3* allows objects and devices in the physical space to define a common information representation model with Resource Description Framework (RDF). Information in the SIB is stored as RDF graphs or Triples (*Subject, Predicate, Object*).

The *Smart-M3* space is composed of one or more SIBs but even if the information may be distributed over several SIBs, the information result is the union of information stored in all SIBs associated with that space. Since SIBs are routable, devices see the same information and it does not matter to which particular SIB in a M3 space a device is connected. Figure 5.1 shows the *Smart-M3* Architecture.

This chapter presents a framework for simplifying the development of KPs agents. The *Smart Space* interface is abstracted by hiding the underlying complexity involved in ontology-driven approaches. This is achieved through a Python and C API generated from an OWL-DL ontology.

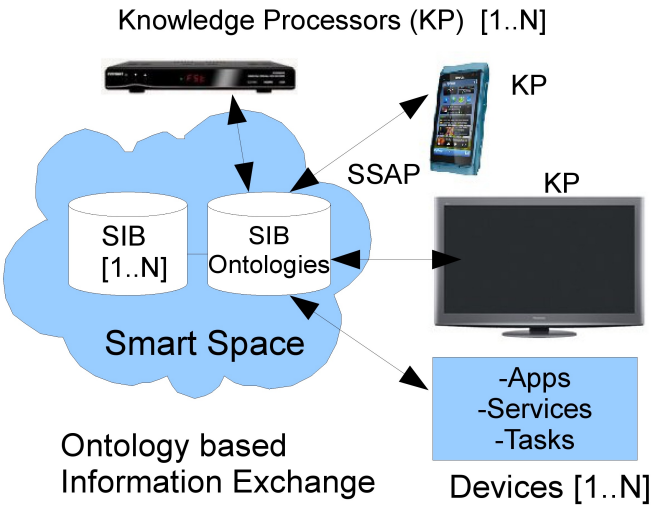

Figure 5.1 Smart-M3 architecture.

5.3.1 Ontologies in *Smart-M3*

The information interoperability needed in *Smart Spaces* can be realized by agents that describe their information using a common ontology. In the last version of the Ontology Web Language, OWL 2, ontologies can also be viewed as RDF graphs, i.e. the structural form is mapped to the RDF graph form and vice versa [2].

OWL ontologies can define the context information representing data directly obtained from context providers plus inferred information from this data using inference rules. Thus, we chose ontology based context modeling for *Smart Space* for several reasons. Firstly, the platform *Smart-M3* provides an interoperability architecture based on ontology models with support for RDF graphs storage, code generation and ontology reasoning; secondly, we chose ontology based context modeling for the fact that ontologies are the most promising and expressive models satisfying information interoperability requirements. Moreover, ontology based models provide flexibility, extendibility and genericity, key factors in context-aware ubiquitous spaces [9].

Since the main design goal for our research is the rapid and easy application development for *Smart Space* environments, the dynamic nature of OWL is something from which we can definitely benefit in the modeling.

Ontologies enable the expression of information and relations in an application. An ontology allows KPs to access and process the information related to their functionality from the *Smart-M3* space, consequently driving the KPs through the space [22].

5.3.2 Tool Support for Knowledge Processors creation

Our approach consists of two development tools modules:

1. The first part is a Python code generator that creates a static API from an OWL ontology [4] as illustrated in Figure 5.2. These mapping generate native Python classes, methods and variable declarations which can then be used by the application developer to access the data in the *Smart-M3* space as structured and specified in the OWL ontology. The generator loads an OWL ontology into a Java ontology model which provides interfaces for accessing the RDF graph. A reasoner is connected to the model to complete the inferred part of the ontology. The generator then lists all named classes in the ontology and the handler creates a counterpart OWL class in Python which is added to the code model. The class handler will list all properties and call the *ObjectProperty* and *DatatypeProperty* handlers which, in turn, translate every restriction that the property may have, e.g. `Cardinality` and `Range` restrictions.
2. The second component is the middleware layer which abstracts the communication with the *Smart-M3* as illustrated in Figure 5.3. Its main functionality is the handling of information in the central SIB with the generated API. This consists of inserting, removing and updating RDF Triples and committing changes to the *Smart Space*. It also provides functionality for synchronous and asynchronous querying. Our approach enables application developers to use the generated API to develop new applications without worrying about the *Smart-M3* interface as the generated API takes care of the connection to the *Smart Space* each time an object is created. From the *Smart-M3* point of view, the proposed framework simplifies the development of KPs by making the *Smart Space* interface more abstract and hiding the underlying complexity involved in ontology-driven approaches [16].

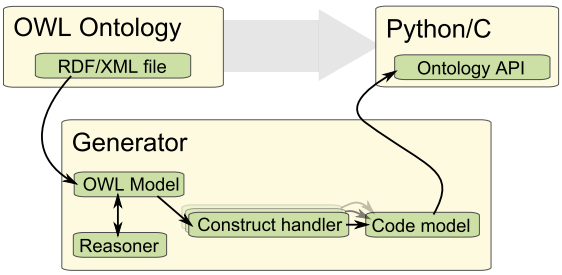

Figure 5.2 Framework overview.

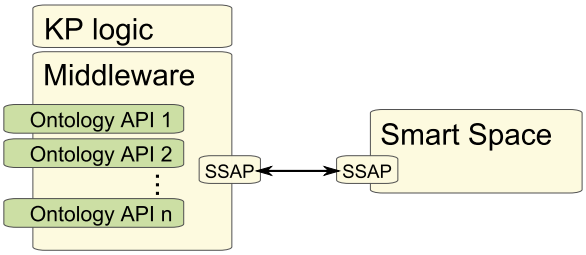

Figure 5.3 Runtime middleware for *Smart-M3*.

5.4 Case Study: From Ontology Editing with Protégé to an Application Example in Home Automation

Our demonstration scenario [16] uses a home state switch, reflecting the global state (i.e. "Home", "Away" and "Vacation"), and a heating system. Moreover, there are two additional parts for enabling interoperability: a controller and a configuration tool. In addition to these interoperating components, there is also a temperature display, and a temperature slider which can be configured to correspond to or set the different temperatures available from the heater appliance. The demonstration application consists of several KPs representing the functionality of the devices and a user interface. All devices at home connect to the SIB and insert information about themselves. A conceptual model is shown in Figure 5.4. The demonstration implementation contains a temperature service concept in addition to the house state concept shown in Figure 5.4. The temperature data service is contained in the Heater, Temperature Slider and the display. An example configuration is to set the display to show the active temperature setting in the heater.

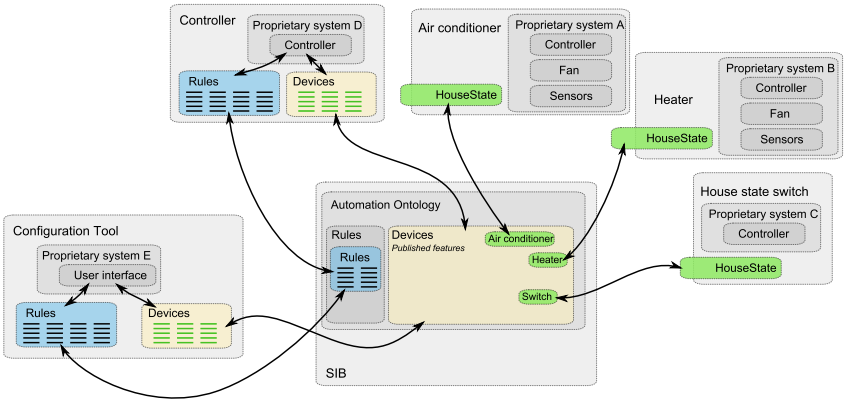

Figure 5.4 Case study overview showing an interoperability solution.

In this use-case we created an ontology containing rules for automation, concepts for expressing the house state, and the temperature. These ontology components and their attributes were edited with *Protégé* [6]. The Python Code Generator was used to generate the agent ontology API with populated instance properties.

The case study illustrates an application development approach for *Smart-M3* using our proposed framework. The KPs are developed from the generated ontology API and are able to communicate through the *Smart-M3* space providing interoperability between different devices in the example application.

All devices in the home connect to the SIB, through their respective SIB interfaces, and insert information about themselves. This information consists of a user friendly name, a list of services it provides and the data which describes its state. No automatic configuration about how they interact exist at this time. When the configuration tool is run, the user is presented with devices registered in the SIB, and can then configure rules. Rules are interpreted by a controller KP. The controller subscribes to changes in the data of the devices. In order to catch changes in state of the switch, the controller listens to new instances of the `Event` class. This instance contains information about what has occurred. When the controller receives a new instance of an `Event`, it parses through the list of rules and if there is a matching rule, it will execute the rule. In this simple implementation, a matching rule will create a new instance of the class `Invoke` and adds properties to it according

to the configured rules. The new `Invoke` instance is subscribed to by the KP representing the service invoked, and can then be used to alter the internal state accordingly.

All devices of interest connect to the SIB through their respective SIB interfaces, insert information about themselves, the service they provide and the data which describes its state. When the configuration tool is run, the user is presented with devices registered in the SIB, and can then configure rules. Rules are interpreted by a controller KP which subscribes to changes in the data. In order to catch changes in the switch state, the controller listens to new instances of the `Event` class.

The API generator and DIEM Mediator[1] source code is available from *Smart-M3* at SourceForge [5]. The demonstration was tested with Python 2.6.x, PyQt v.4.5.4 for Python 2.6 and Nokia SIB revision 98.

The following agents in the building automation demonstration try to connect to a *Smart Space* named 'x' on 127.0.0.1 at port 10010 by default:

- `Controller.py`
- `ConfigurationTool.py`
- `HomeStateSwitch.py`
- `Heater.py`
- `TemperatureSet.py`
- `TemperatureDisplay.py`

Running `python SIB.py x` starts a SIB running locally at port 10010 with the *Smart Space* name `x`. The simplest use-case to run is the *HomeStateSwitch* and the *Heater*, with pre-configured addresses 17 and 7 respectively. These can be connected by the configuration tool using the following commands:

```
Command] list
Command] connect
Source address: 17
Source feature: 0
Destination address: 7
Destination feature: 0 (State might have another number)
Rule name: TestRule
```

If commands are executed correctly the heater will output its changing state following the home state switch. The KPs can be started in any order, but the

[1] A caching middleware for accessing the SIB

configuration tool KP does not find any devices until they are started. The suggested order is to run the controller and the configuration tool, and then any of the service providing devices or KPs. Note that subscriptions to the SIB result in a TCP timeout if no subscribed data is sent by the SIB within a quite short period of time depending on network configuration. Python cannot platform independently set TCP keep-alive messages and it is therefore recommended to run the demo locally. There are some subscriptions which are not required after running the configuration tool, thus the example might work even after this timeout error.

More information about the development tool can be found in [16].

5.5 Structured View of Smart Space

In order to explain the structure of the *Smart Space* and the role of the KPs ontologies we will consider a simple application for not missing our favorite TV program. Let us suppose that the user's favorite program is starting in few minutes according to the user profile information or fan page in Facebook and the TV guide available on the broadcaster's web page. Then the GPS in his mobile phone or his personal calendar could find out that he is not at home and start the PVR (Personal Video Recorder) to record the program. In order to address this kind of cross-domain scenario where technical and conceptual problems arise, the concept of *Smart Space* appears to encapsulate and abstract information from different services with the aim of allowing heterogeneous service composition.

A PVR could be considered as a form of API with different functions. One or several KPs can be perceived as a service, for example several KPs handling calendar activities in an application could shape a calendar service. Thus, each service acts as service provider exposing its functionality to other KPs and services through the *Smart Space*. At the same time each service acts as requester too.

In this way, we could have the PVR's KP and the mobile phone's KP connected to the SIB. Figure 5.5 shows the registered devices' KPs with their information described in their respective ontologies.

Each of these subservices within a device inform their inputs and outputs among other parameters in each of their profiles. In order to deploy the scenario of recording the favorite program, the composition of required services must be deployed in the SIB, which knows about the devices connected to the *Smart Space*. Here we find the problem that the SIB offers a persistent data repository but is a plain database giving just access to the data; no control

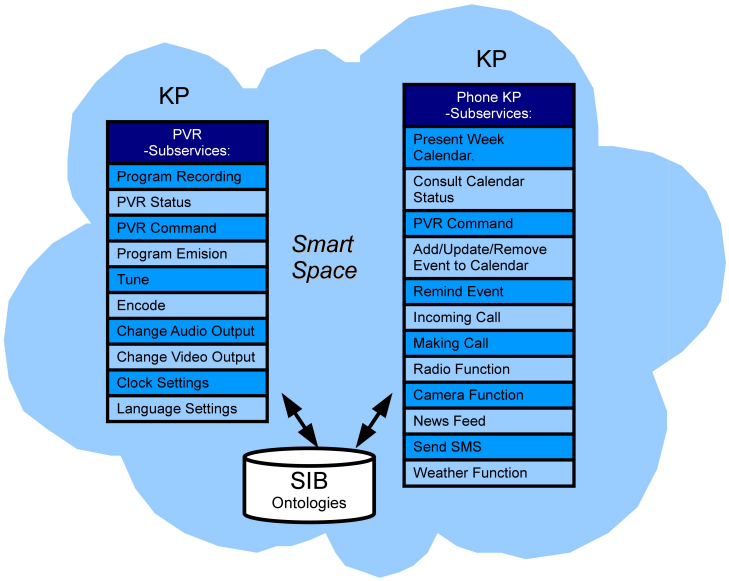

Figure 5.5 KP's services structure.

structure or computation is provided. However, if we add to the SIB a description of each subservice, all devices would be represented with a unique standard allowing language and device independent service composition. For this purpose we suggest service description with OWL-S [3] representation. The reasons are that OWL-S enables declarative advertisement of service properties and capabilities that can be used for automatic service discovery and because it describes the services in terms of capabilities based on OWL (also supported by *Smart-M3*). In addition to provide specification of prerequisites of individual services, OWL-S language describes services composition including data flow interactions [22].

By storing OWL-S instances (associated to each KP) in the SIB, the SIB would gain processing control. In this case an OWL-S interpreter running in parallel with the SIB would control the matching of compatible services when requested. Consequently, the original *Smart-M3* framework would be enhanced.

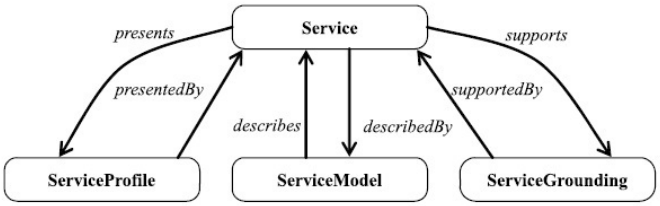

Figure 5.6 Main classes in OWL-S service ontology.

5.6 Service Ontology Description

The new generation of markup languages such as OWL was developed to support the description of specific web sites and reasoning applications. The idea behind them is to access web resources by content rather than just by keywords. When we talk about services, we will refer to local or web applications that allow actions or changes to happen in a semantic environment that enables users to locate, query, select, invoke, compose, reuse and monitor web-based services automatically [3].

OWL-S (formerly DAML-S) is formally defined as an ontology of services that allows users and software agents to interact. The OWL-S ontology is structured in three parts (Figure 5.6):

- *Service Profile* for advertising and discovering services: "What the service does". Each instance of the class *Service* "presents" an instance of the *ServiceProfile* class.
- *Process Model* for describing a service's operation: "How the service works". The class *ServiceModel* captures the property which "describes" a *Service* instance.
- *Grounding* for specifying how to interoperate with a service via messages: "How to access the service" [1]. The *ServiceGrounding* class serves as a "support" property of a *Service* instance.

A service is described by at most one service model, and a grounding must be associated with exactly one service. However, it can be useful for some services to present multiple profiles and/or groundings. An OWL-S Profile describes a service as a function of what organization provides the service, what function the service computes, and what features specify characteristics of the service. The Process Model specifies how to interact with the service. Profile and Process Model hold two different representations of the same service coinciding in input, output, precondition, and effects (IOPEs). The

properties described link the Service Profile class with the Service class and Process Model class. Parameters can conveniently be identified with variables in SWRL (Semantic Web Rule Language), the language for expressing OWL Rules. Finally, the Grounding class specifies an unambiguous way of exchanging I/O data: communication protocol, message formats and other service-specific details. An important goal for Semantic Web markup languages, then, is to establish a framework within which these descriptions are made and shared.

OWL-S supports two categories of services; one is *atomic* services where a single web-accessible computer program, sensor, or device is invoked by a request, performs its task and perhaps produces a response to the requester. There is no ongoing interaction between the user and the service. In contrast, *complex* or *composite* services are formed of multiple primitive services and may require an extended interaction/ conversation between the requester and the set of services to be utilized. This distinction suggests us to add to every KP agent an OWL-S specific capability to make easier the interaction of device applications with web services.

While Service Profile and Service Model are abstract representations, the Service Grounding deals with the concrete level of specification. In order to inputs and outputs of a process to be realized concretely as messages, Web Services Description Language (WSDL) carries out the OWL-S grounding mechanism with a consistent binding. In OWL-S a binding is an abstract object with two properties: *toParam* (name of the parameter) and *valueSpecifier* (value description). It results in an effort to provide a specification as concise as possible in variety of situations. WSDL contains document and process information in XML format to bind to a concrete network protocol and message format allowing the service developer to benefit from WSDL's functionality and similar languages (as SOAP, HTTP GET/POST, and MIME) for message exchange. As the OWL-S concept of grounding is generally consistent with WSDL's concept of binding, WSDL is used as the ground of an OWL-S atomic process. For OWL-S/WSDL grounding both languages are needed and complementary. Both languages overlap in the area where WSDL's abstract types, which are used to characterize inputs and outputs of services (specified in XML Schema) correspond to OWL-S' DL-based abstract types OWL classes. However, WSDL/XSD is unable to express the semantics of an OWL class and OWL-S cannot express the binding information that WSDL captures in messages [1]. Therefore, an OWL-S/WSDL grounding uses OWL classes as abstract types of message parts declared in WSDL and then relies on WSDL binding constructs to specify the formatting of the messages.

Table 5.1 Basic elements to define a service.

Basic Elements to define a Service	
OWL-S Element	**Definition**
Type	Data type definition to describe the exchanged messages.
Message	Abstract definition of the data being transmitted.
PortType	Set of abstract operations. Each operation refers to an input message and output messages.
Binding	Concrete protocol and data format specifications for the operations and messages defined by a particular *PortType*.
Port	Address for a binding defining a single communication endpoint.
Service	Used to aggregate a set of related ports.

Grounding OWL-S with WSDL and SOAP (assuming HTTP as transport mechanism) involves the construction of a WSDL service description with all the usual parts (types, message, operation, port type, binding, and service constructs). The essence of an OWL-S/WSDL grounding can be summed up by creating an instance of the OWL-S Grounding class which includes all required information regarding relationships between relevant OWL-S constructs and WSDL constructs [1].

The basic elements for defining a service are described in Table 5.1.

The most basic and concrete class in OWL-S, *WsdlAtomicProcessGrounding* establishes the grounding mechanism details within a WSDL specification. In order to formalize the details of the grounding, Table 5.2 shows the main properties in the *WsdlAtomicProcessGrounding* class.

5.6.1 Proposed OWL-S Python binding

As OWL-S language is an ontology itself, in order to express preconditions and effects is combined with SWRL (Semantic Web Rule Language), the language for expressing OWL Rules based on OWL DL and Lite and RuleML (Rule Markup Language). Inputs/Outputs are subclasses of SWRL variables. Discovery and composition operate on description logic reasoning.

Table 5.2 Properties representing the OWL-S grounding class *WsdlAtomicProcessGrounding*.

OWL-S Grounding Class	
OWL-S Grounding Property	**Definition**
wsdlVersion	URI indicating WSDL version.
wsdlDocument	URI of WSDL document of the referring grounding.
wsdlOperation	URI of WSDL operation corresponding to the atomic process.
wsdlService and *wsdlPort* (optional)	URI of WSDL service (or port) that offers the given operation.
wsdlInputMessage	Object containing URI of the WSDL message definition carrying inputs of the given atomic process.
wsdlInput	Object containing a list of mapping pairs, (instance of *WsdlInputMessageMap*) with *wsdlMessagePart* property-URI of input object (*owlsParameter*) or *xsltTransformation* property (string or URI) which generates the message part from an instance of the atomic process.
wsdlOutputMessage	Analogue to *wsdlInputMessage*.
wsdlOutput	Analogue to *wsdlInputs*.

Below we show a simple Python example to describe a situation in which the user is watching TV and his PVR (Personal Video Recorder) is On. In order not to miss part of the program and to be able to speak without noise, we would like the PVR to pause automatically when the user receives a phone call. This application could be modeled as Listing 5.1 shows.

```
1 if phone.isRinging(user.getPhoneNo)) & PVR.isOn():
2   PVR.setPause()
```

Listing 5.1 Python Rule Example

A rule expressed in an OO language for modeling a KP behavior can involve different services among its components. We could consider each call to a Python function as a different atomic service which, combined, form a composite service. Once a rule is created or developed, as we have done in

Python, it needs to go through different phases in its execution cycle for it to be completely deployed. The phases start with the installation of the rule in the *Smart Space* until its uninstalation or removal. Next the rule sequential phases are enumerated:

- Rule INSTALLED: All instances of the different KPs involved in the rule must be known and available in the SIB, in other words, a rule is installed when its involved KPs are connected to the *Smart Space* and registered in the SIB.
- Rule RESOLVED: A rule is resolved when for each call to a Python function there exists a specified and available service grounding. Then the rule can be considered as registered in the *Smart Space*. Access control to the KP services is also checked and is here where OWL-S acts as intermediate specification. Without service formalization and grounding in OWL-S the binding of the rule realization would become language dependent. Consequently, OWL-S helps achieving multi device interoperability.
- Rule STARTED or SUBSCRIBED: When the rule's corresponding services have requirements or conditions constraining its triggering, the rule will not become active until they are satisfied. The rule is installed and resolved but is blocked. This means that an asynchronous subscription to the SIB is done in order to get a notification callback when the constraints are met. When that occurs, the rule is released and becomes active or published.
- Rule ACTIVE or PUBLISHED: A rule is active or published when the service has been started, triggered and the execution grounding has been realized. The corresponding effects or updates are published in response to subscriptions.
- Rule UNINSTALLED: The rule is removed from the SIB in the *Smart Space* for it to no longer take effect.

In order to generalize the service binding to Python we have to be conscious that not only *atomic* and *composite* services exist. In applications including simple Python rules as the previous conditional construct (Listing 5.1) we can find *active* and *passive* services. Pasive services translate to queries checking availability in the SIB and subscriptions. Active services can, however, translate Python calls into different effects, either actions or changes in the *Smart Space* or simple data updates. For this reason we have had to specify formally the different phases which the rule passes through during its binding. Here appears also our motivation for extending the existing

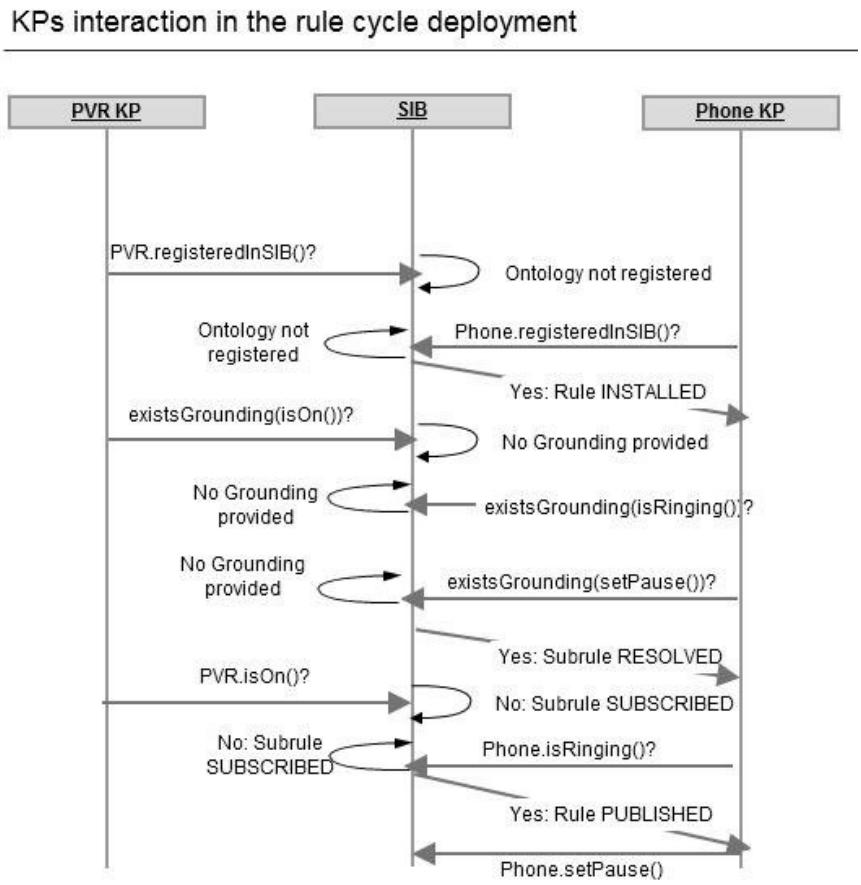

Figure 5.7 Sequence diagram representing the KP's interaction in the rule cycle deployment.

OWL information in the SIB with an OWL-S specification which allows the developer to classify the different types of services and act consequently. For a rule structure to execute in Python, it must be invoked. However, if we structure the OO structures to be represented in form of OWL-S services and introduce them into the SIB, the rules will be always running in the background until their removal is desired.

The Python example in Listing 5.1 could be modeled as the KPs interaction shown in Figure 5.7 representing the rule cycle.

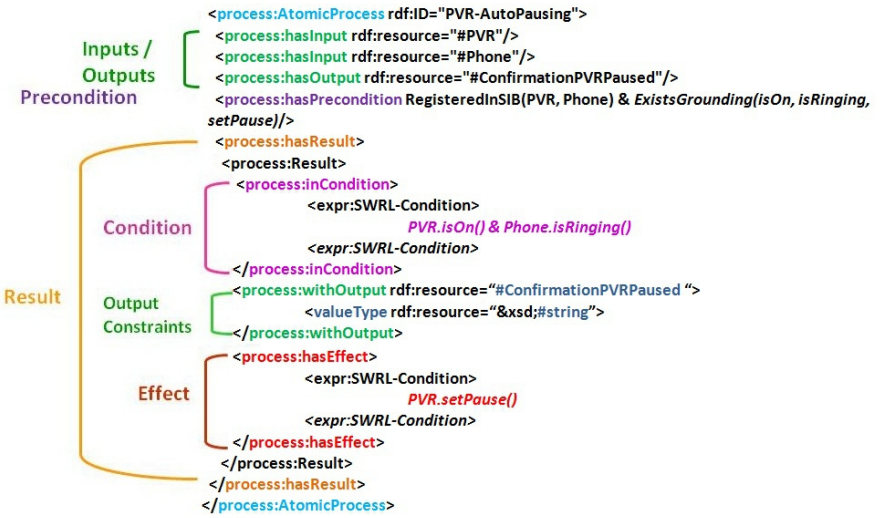

Figure 5.8 Atomic process example.

The previous example showed how one can develop a rule and how can one deploy it in a semantic environment. In order to realize the concrete binding of the service into Python, the OWL-S grounding is needed. We could then model a process as presented and structured in Figure 5.8 and concretize it with an OWL-S grounding as Listing 5.2 specifies.

```
3
4 <!-- The PVR Auto Pausing atomic process -->
5
6  <process:AtomicProcess rdf:ID="PVR-AutoPausing">
7   <process:hasInput>
8   <process:Input ref:ID="PhoneNumber">
9       <process:parameterType rdf:about="&xsd;#string"
            >
10  </process:Input>
11  </process:hasInput>
12  <process:hasInput>
13    <process:Input ref:ID="isPhoneRinging">
14      <process:parameterType rdf:resource="#Phone/
            isRinging">
15    </process:Input>
16  </process:hasInput>
```

```
17   <process:hasInput>
18     <process:Input ref:ID="isPVROn">
19       <process:parameterType rdf:resource="#PVR/isOn"
             >
20     </process:Input>
21   </process:hasInput>
22   <process:hasOutput>
23     <process:Output ref:ID="ConfirmationPVRPaused">
24       <process:parameterType rdf:resource="&xsd;#
             string">
25     </process:Output>
26   </process:hasOutput>
27 </process:AtomicProcess>
28
29
30   <!-- OWL-S Grounding  -->
31 <grounding:WsdlGrounding rdf:ID="SmartPVR">
32   <grounding:hasAtomicProcessGrounding rdf:resource="
       #PVR-AutoPausing"/>
33 </grounding:WsdlGrounding>
34 <grounding:WsdlAtomicProcessGrounding rdf:ID="PVR-
     AutoPausing">
35   <grounding:owlsProcess rdf:resource="#PVR-
       AutoPausing">
36   <grounding:wsdlOperation>
37     <grounding:WsdlOperationRef>
38       <grounding:portType>
39         <xsd:uriReference rdf:value="http://SmartPVR.
               com/PVR-AutoPausing.wsdl#PhonePortType"/>
40       </grounding:portType>
41       <grounding:operation>
42         <xsd:uriReference rdf:value="http://SmartPVR.
               com/PVR-AutoPausing.wsdl#SmartAutoPausing"
               />
43       </grounding:operation>
44     </grounding:WsdlOperationRef>
45   </grounding:wsdlOperation>
46
47   <grounding:wsdlInputMessage rdf:resource="http://
       SmartPVR.com/PVR-AutoPausing.wsdl#
       PausingPVRInput"/>
48   <grounding:wsdlInput>
```

```
49    <grounding:wsdlInputMessageMap>
50      <grounding:owlsParameter rdf:resource="#isPVROn
          ">
51      <grounding:wsdlMessagePart>
52        <xsd:uriReference rdf:value="http://SmartPVR.
            com/PVR-AutoPausing.wsdl#pvr.isOn">
53      </grounding:wsdlMessagePart>
54    </grounding:wsdlInputMessageMap>
55  </grounding:wsdlInput>
56  <grounding:wsdlInput>
57    <grounding:wsdlInputMessageMap>
58      <grounding:owlsParameter rdf:resource="#PhoneNr
          ">
59      <grounding:wsdlMessagePart>
60        <xsd:uriReference rdf:value="http://SmartPVR.
            com/PVR-AutoPausing.wsdl#PhoneNr">
61      </grounding:wsdlMessagePart>
62    </grounding:wsdlInputMessageMap>
63  </grounding:wsdlInput>
64  <grounding:wsdlInput>
65    <grounding:wsdlInputMessageMap>
66      <grounding:owlsParameter rdf:resource="#
          isPhoneRinging">
67      <grounding:wsdlMessagePart>
68        <xsd:uriReference rdf:value="http://SmartPVR.
            com/PVR-AutoPausing.wsdl#phone.isRinging">
69      </grounding:wsdlMessagePart>
70    </grounding:wsdlInputMessageMap>
71  </grounding:wsdlInput>
72
73  <grounding:wsdlOutputMessage rdf:resource="http://
      SmartPVR.com/PVR-AutoPausing.wsdl#PVROutput"/>
74  <grounding:wsdlOutput>
75    <grounding:wsdlOutputMessageMap>
76      <grounding:owlsParameter rdf:resource="#
          ConfirmationPVRPaused">
77      <grounding:wsdlMessagePart>
78        <xsd:uriReference rdf:value="http://SmartPVR.
            com/PVR-AutoPausing.wsdl#pvr.SetPause()">
79      </grounding:wsdlMessagePart>
80    </grounding:wsdlOutputMessageMap>
```

```
81  </grounding:wsdlOutput>
```

Listing 5.2 OWL-S Atomic Process and Grounding Example

5.7 Conclusions and Future Work

A solution for application development was presented in this chapter integrating ontologies and the *Smart-M3* platform. First, developing a tool for mapping OWL to OO languages (available in Python and C) providing complete control over ontologies was described. Second, a middleware was constructed encapsulating the communication with the *Smart Space*. This module allows Python to write applications by using the generated Ontologies APIs. Later a case study implicitly showed the communication among agents through *Smart-M3* Space achieving device interoperability.

For these ontologies to allow composition of services, we finally proposed a deployment through a fixed grounding in the SIB by using an OWL-S description of each subservice. Thus, by extending the previous framework with OWL-S information making it available in the SIB we add support for service interaction and composition.

We conclude by showing the suitability of *Smart Space* for ubiquitous applications where physical environments adapt to the user and the surrounding information is reusable and dynamic. With the integration of these different components with *Smart-M3*, rapid development of context-aware applications for *Smart Space* is made available so that agents can share information independently of which device they are embedded in.

In the future, we aim at creating a context processing library with a Python module for embedding rules expressions [21]. Other challenges to be tackled are e.g. SIB consistency related issues or efficient subscriptions implementation. Then, use cases for other environments can be applied.

Acknowledgment

The presented work was funded through the ICT-SHOCK DIEM project by TEKES (Finnish Funding Agency for Technology and Innovation).

List of Abbreviations

- *Smart-M3*: Multi-domain, Multi-device and Multi-vendor (M3) *Smart Space* platform.

- SIB: Semantic Information Broker.
- KP: Knowledge Processor.
- SSAP: Smart Space Access Protocol.
- OWL: Ontology Web Language.
- OWL-S: Ontology Web Language Services.
- OWL-DL: Ontology Web Language - Description Logic.
- OO Programming: Object Oriented Programming.
- WSDL: Web Service Definition Language.
- WSMO: Web Service Modeling Ontology.
- UPnP: Universal Plug and Play.
- RDF: Resource Description Language.
- SOAP: Simple Object Access Protocol.
- HTTP: Hypertext Transfer Protocol.
- DAML: DARPA Agent Markup Language.
- DARPA: Defense Advanced Research Projects Agency.
- XML: Extensible Markup Language.
- API: Application Programming Interface.
- MIME: Multipurpose Internet Mail Extensions.
- SWRL: Semantic Web Rule Language.
- RuleML: Rule Markup Language.
- WQL: Wilbur Query Language

References

[1] Describing Web Services using OWL-S and WSDL: http://www.daml.org/services/owl-s/1.1/owl-s-wsdl.html.

[2] OWL 2. [online]:http://www.w3.org/tr/owl2-overview/.

[3] OWL-S. [online]:http://www.w3.org/submission/owl-s/.

[4] Smart-M3 Ontology to Python API Generator: http://sourceforge.net/projects/smart-m3/files/smart-m3-ontology_to_python-api_generator_v0.9.1beta.tar.gz/.

[5] Smart-M3 software at sourceforge.net, release 0.9.4beta, May 2010. [Online]. Available: http://sourceforge.net/projects/smart-m3/.

[6] The Protege Ontology editor and knowledge acquisition system: http://protege.stanford.edu/.

[7] WSMO Grounding: http://wsmo.org/tr/d24/d24.2/v0.1/.

[8] Grigoris Antoniou and Frank van Harmelen. A semantic web primer. *International Journal of Ad Hoc and Ubiquitous Computing*, 2:212–223, 2008.

[9] Matthias Baldauf, Schahram Dustdar, and Florian Rosenberg. A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, 2, 2007.

[10] Dipanjan Chakraborty, Anupam Joshi, Tim Finin, and Yelena Yesha. Service composition for mobile environments. *Mob. Netw. Appl.*, 10:435–451, August 2005.

[11] Harry Chen, Tim Finin, and Anupam Joshi. An ontology for context-aware pervasive computing environments. In *Proceedings of the Workshop on Ontologies in Agent Systems*, 2003.

[12] Oleg Davidyuk, Nikolaos Georgantas, Valérie Issarny, and Jukka Riekki. MEDUSA: Middleware for End-User Composition of Ubiquitous Applications. In *Handbook of Research on Ambient Intelligence and Smart Environments: Trends and Perspectives*. IGI Global, 2010.

[13] Anind K. Dey and Gregory D. Abowd. The Context Toolkit: Aiding the development of context-aware applications. In *Workshop on Software Engineering for Wearable and Pervasive Computing*, 2000.

[14] Jing Dong, Yongtao Sun, Sheng Yang, and Kang Zhang. Dynamic web service composition based on OWL-S. *Science in China Series F: Information Sciences*, 49:843–863, 2006.

[15] Tao Gu, Hung Keng Pung, and Da Qing Zhang. A middleware for building context-aware mobile services. In *Proceedings of IEEE Vehicular Technology Conference (VTC)*, 2004.

[16] Andre Kaustell, M. Mohsin Saleemi, Thomas Rosqvist, Juuso Jokiniemi, Johan Lilius, and Ivan Porres. Framework for Smart Space Application Development. In *Proceedings of the International Workshop on Semantic Interoperability (IWSI 2011)*, 2011.

[17] Shalil Majithia, David W.Walker, and W.A.Gray. Automated web service composition using semantic web technologies. In *Proceedings of the International Conference on Autonomic Computing (ICAC04)*, 2004.

[18] R. Oppermann and M. Specht. A context-sensitive nomadic exhibition guide. In *Proceedings of Second Symposium on Handheld and Ubiquitous Computing*, pages 127–142, Springer, 2000.

[19] W. Prinz. NESSIE: An awareness environment for cooperative settings. In *Proceedings of the Sixth European Conference on Computer-Supported Cooperative Work*, pages 391–410, 1999.

[20] M. Roman, C. Hess, R. Cerqueira, and A. Ranganathan. A middleware infrastructure for active spaces. In *IEEE Pervasive Computing*, 2002.

[21] M. Mohsin Saleemi, Natalia Diaz, Johan Lilius, and Ivan Porres. A framework for context-aware applications for smart spaces. In *Proceedings of ruSMART 2011: The 4th Conference on Smart Spaces*, 2011.

[22] M. Mohsin Saleemi and Johan Lilius. End-user's service composition in ubiquitous computing using Smartspace approach. In *Proceedings of The Sixth International Conference on Internet and Web Applications and Services (ICIW 2011)*, 2011.

[23] Evren Sirin, James Hendler, and Bijan Parsia. Semi-automatic composition of web services using semantic descriptions. In *Proceedings of workshop on Web Services: Modeling, Architecture and Infrastructure (ICEIS2003)*, pages 17–24, 2002.

[24] Z. Song, Y. Labrou, and R. Masuoka. Dynamic service discovery and management in task computing. In *Proceedings of Mobile and Ubiquitous Systems: Networking and Services (Mobiquitous2004)*, 2004.

[25] Espen Suenson, Johan Lilius, and Ivan Porres. OWL web ontology language as a scripting language for Smart Space applications. In *Proceedings of the International Symposium on Rules, RuleML*, 2011.

[26] Mathieu Vallee, Fano Ramparany, and Laurent Vercouter. Flexible composition of smart device services. In *Proceedings of the 2005 International Conference on Pervasive Systems and Computing (PSC-05)*, Las Vegas, pages 27–30, 2005.

[27] Paul Wisner and Dimitris N. Kalofons. A framework for end-user programming of smart homes using mobile devices. In *Proceedings of the Consumer Communications and Networking Conference, CCNC*, 2007.

[28] X.H. Wang, D.Q. Zhang, T. Gu, and H.K. Pung. Ontology based context modeling and reasoning using OWL. In *Workshop Proceedings of the 2nd IEEE Conference on Pervasive Computing and Communications*, 2004.