Framework for Smart Space Application Development

André Kaustell, M. Mohsin Saleemi, Thomas Rosqvist, Juuso Jokiniemi, Johan Lilius and Ivan Porres

Department of Information Technologies
Turku Centre for Computer Science (TUCS), Åbo Akademi University
Turku, Finland
{firstname.lastname}@abo.fi

Abstract. The *smart space* concept envisions interoperability for creating multi-domain inter-device service mashups, and innovative applications. This paper presents an appraoch to create an abstraction layer and appropriate tools for rapid application development for the low-end devices that run agents to access a smart space. We present our proposed framework, describe the overall process for application development and explore the features of a real-world implementation. We describe a case study implementation to illustrate the functionality of the proposed framework. This case study shows that interoperability can be realized by the agents that describe information about themselves using some common ontology.

1 Introduction

Recent advances in information and communication technologies have made available the device, service and information rich environment for the users. It helps simplify and manage complex lives e.g the ubiquitous smartphone that manages your calender, contacts and task-list and helps you keep your life organized, or the PVR whose time-shift functionality allows us to watch TV programming when we want, not at the time prescribed by the broadcaster. However each of these devices is basically an island, with no proper connectivity between the applications. In order to take full advantages of information rich environment, devices need to interact with each other. Although some devices are able to interoperate, this is usually possible between devices by the same manufacturers e.g the connectivity in a home cinema system. These proprietary solutions have limitations in terms of scalability and interoperability. By exposing the internal data and functionality of the devices and ensuring interoperability of data, a whole new universe of applications will be possible. For example, your phone could notice that your favorite TV program will start in 5 minutes, based on your profile information or on a fan page on facebook and on the TV guide available on the broadcaster's web page. Then it could use GPS to realize that you are not at home, and deduce that it needs to start the PVR at home.

To enable this kind of cross-domain scenarios, there are many technical and conceptual problems to be solved. One way to address these issues is through the notion of smart space. Smart space is an abstraction of space that encapsulates both the information in a physical space as well as access to this information, such that it allows devices to join and leave the space. In this way, smart space becomes a dynamic environment whose membership changes over time when the set of entities interact with it to share information between them. For example, communication between the mobile phone and the PVR in the above scenario does not happen point-to-point but happens through the smart space whose members are the mobile phone and the PVR.

Our research is focused on rapid and easy developement of smart space applications. We have developed user level tools that enable application developers to create innovative smart space applications using traditional object-oriented programming concepts. The main contributions of this paper include the framework for the application developement approach and the results from a practical case study implementation using our framework.

The rest of the paper is organized as follows. In Section 2 we provide related work in this area. Section 3 gives an overview of the Smart-M3 concept. Section 4 presents our approach towards the development of the framework. The same section also describes the proposed framework, its implementation and validation in detail. Section 5 presents the implementation detail and results of a practical application scenario using our framework to give the proof of concepts. In Section 6, we present our conclusions and depict directions for future work.

2 Related Work

Research in the area of middleware and application development for ubiquitous computing has provided many research concepts and prototype development of these concepts. The work presented in [7] is an example of an ubiquitous computing approach consisting of networked devices and services and enables the end-users to combine, configure, and control the services using their smart devices. The described work provides prototype implementation and an example of composing tasks perceived by the end-users by mapping the concepts to the available devices through semantic matching.

Smart space research still lacks the availability of commonly adopted middleware platforms and tools for rapid development of interoperable smart space systems and applications. The semantic web introduced many standards such as XML, RDF, Jena [1], OWL and OWL-S, which provide interoperability and enable the development of web services and applications. There are quite a few approaches that try to integrate programming languages and the semantic web standards. In [3], an approach to integrate OWL into object oriented (OO) language is presented using Python metaclasses. Using this approach, the problems can be defined by description logic (DL) and manipulated by dynamic scripting languages. Another approach to map OWL ontology into Java is presented in [5]. The work provides a solution for OWL-Full which is the most expressive sub-language of OWL. The authors identify the semantic differences between DL and the OO systems.

None of these approaches provide a complete solution for the development of smart space based applications. Our approach gives a complete solution for rapid applications development. It incorporates OWL ontology for application description, a developed tool for mapping OWL to OO languages (Python and C) to give complete control over ontologies for application developement, and a middleware for encapsulating communication with the smart space.

3 Smart-M3 Concept

The main concept in smart space based systems is a publish-subscribe method and it can be implemented using different approaches and technologies. Our approach for rapid development of interoperable smart space applications is based on Nokia's open source Smart-M3 architecture [6]. The Smart-M3 architecture provides a particular implementation of smart space where a central repository of information is the Semantic Information Broker (SIB). The smart-M3 space is composed of one or more SIBs where information may be distributed over several SIBs for the later case. The information in the M3 space is the union of information stored in the SIBs associated with that space. The set of SIBs in a M3 space are completely routable and the devices see the same information, hence it does not matter to which particular SIB in a M3 space a device is connected. The information is accessed and processed by the entities called Knowledge Processors (KPs). KPs interact with the M3 space by inserting, retrieving or querying the information in any of the participating SIBs using access methods defined by the Smart Space Access Protocol (SSAP). Smart-M3 provides information level interoperability to the objects and devices in the physical space by defining common information representation models such as Resource Description Framework (RDF). Fig. 1 shows the overall Smart-M3 architecture. It comprises the following components and concepts that work together to provide a functional framework.

The *SIB* is the core component in Smart-M3 as it is responsible for information storage, sharing and management. The information is stored in the SIB as RDF graphs. The SIB acts as information broker between the devices for storing, receiving or querying information in the RDF triplestore contained in the SIB. For information sharing, devices do not communicate directly with each other, instead information is passed via the SIB through the SSAP protocol.

The *Knowledge Processor (KP)* is the component that performs only a single task. A KP must first join the M3 space in order to access the information. It can then manipulate the information in the SIB by accessing it through the operations such as insert, remove, update, query and subscribe defined by the SSAP communication protocol. A single device can run any number of KPs.

The *Application* is constructed by the composition of several KPs where each KP performs a single task. The application design in this approach differs from the traditional single device control-oriented application.

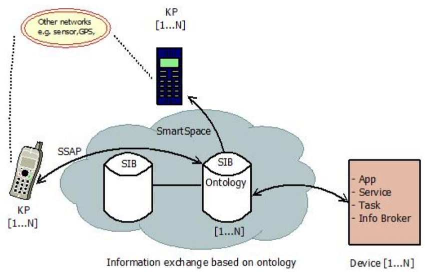

Fig. 1. Smart-M3 Architecture

4 Framework Proposal

The primary design goal of our research is to develop user level tools for rapid and easy application development for smartspace environments, while being able to provide benefits from the dynamic nature of OWL. We have developed a framework consisting of a generator and a middleware, for rapid application development for smart-M3. The generator generates an API from a given ontology at design-time using a set of defined mappings. At run-time, this API, in combination with a middleware layer, enables application developers to create applications for smart spaces using traditional well-defined object oriented programming approaches. The middleware conceals the communication details and provides a smart space independent interface to the generated library. 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.

4.1 Architecture and Design Decisions

In addition to the Smart-M3 concept, some constraints have been added. Where the Smart-M3 builds around RDF, we assume that OWL-DL will be needed for solving the data interoperability issue during runtime as RDF is not restrictive enough to be decidable. OWL-DL is the most expressive subset of OWL, and is derived from Description Logics which makes it computationally complete and decidable [4]. Therefore, our design choices attempt to retain OWL-DL expressiveness. The emphasis of this work is to make the features provided by the Smart-M3 concept available to the application and KP developer and to explore possible models for developing applications to the smart space environment. In addition, we assume that during runtime, the OWL-DL consistency and reasoning will be done by the back-end.

When designing an API for accessing OWL ontologies many problems arise, and there are a number of design choices to be made. OWL individuals do not strictly belong to a class which means that an individual whose properties change, can dynamically change class memberships. One solution is to build a model which can contain all the OWL assertions. The drawback with this

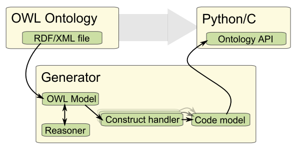

Fig. 2. Smart space development framework overview

approach is that accessing this model is not straightforward, and requires the programmer to understand the OWL concept in great detail. An alternative approach is to provide access to the ontologies by mapping OWL into OOP. This enables direct access to the data using objects or data structures, and can produce significantly more readable and maintainable code. This is the approach of SETH [2], which uses the meta-class capabilities of Python to model the ontologies and instances. This approach is more suitable for a programmer with no, or little, prior knowledge of OWL.

Our approach consists of two parts; 1) A generator which creates a static API from an OWL-DL ontology, which is illustrated in Fig. 2. 2) A middleware layer which abstracts the communication with the persistence layer, as illustrated in Fig. 3. The generated API contains parts of the structure definition from the ontology such as the OWL classes and properties. The generated classes have *get* and *set* methods for properties according to the ontology.

4.2 Generator

The generator creates the API from OWL construct mappings. The generated code contains OWL classes, properties and restrictions. These can then be used by the KP developer to access the data, as structured and specified in the OWL ontologies. The code is also accessed by the middleware layer during run-time to commit changes to the smart space.

API Generation Mappings The first mappings create data structures for containing the named OWL classes, and their subclass structure. Then the data structures for OWL data and object properties are created. There are also several mappings for transferring comments from the ontologies to the API. Class documentation as well as property documentation is injected into the generated API. A well documented ontology also translates into a well documented ontology API. The comments are implemented for getters and setters of both data and object properties, and for all named classes. This is especially important for aiding fast development, and also assists in understanding how the ontology was designed to be used. In addition to the class and property documentation, comments on the ontology are also copied to the generated API.

Reasoning Considerations The benefits of using the OWL-DL, subset of OWL, is that it is restricted in a such way that it can always be checked for consistency, thus preserving logical correctness. This however, requires a reasoning engine which realizes logical conclusions from the ontology and asserted

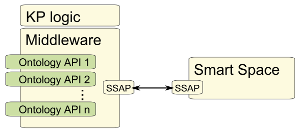

Fig. 3. Runtime architecture overview

facts. During runtime, the reasoner will usually work on the asserted individuals and their properties, rather than the ontology specification like the compile time reasoning. After the reasoner has stepped through the model successfully it is guaranteed that the ontology is valid. That is, the classes, properties and restrictions are constructed properly, all properties have their correct counterparts and the properties data type restrictions are feasible. Reasoning executed in this fashion creates one drawback on the code model of the OWL ontology. The problem is that reasoning is only performed before the actual framework is used. In theory, it is possible for the coder to create an application that would need runtime reasoning to ensure that a complicated ontology stays consistent. Not having runtime reasoning is, however, acceptable at this stage as the goal is not to convert a full model of OWL into Python, but to create a platform for application development.

4.3 Runtime Layer

This section explains the runtime middleware layer. Its main concept is to give the ontology API a smart space independent interface that encapsulates the communication and provides modularization. In addition, it provides the KP developer with the functionality that is not based on the ontology from which the ontology API is generated from.

Runtime Middleware Layer The main functionalities that the middleware provides to the generated ontology API are triple handling, synchronous and asynchronous querying. The middleware layer has been implemented for the generated C and Python ontology API's. They both share the same common concept but have some minor differences in functionality.

The Python middleware communicates directly with the SIB while the C middleware uses the *Whiteboard library*. The *Whiteboard library* is part of the Smart-M3 solution and it provides access to the smart space infrastructure. If the communication interface with the smart space changes, only the middleware layer needs to be changed, not the generated ontology API.

The C middleware contains a local cache where all the triples that have not been inserted are stored. This gives the KP developer the possibility to choose when to insert triples into the smart space. The usage of the local cache can be enabled or disabled at runtime. When disabled, triples are sent directly to the SIB and storage to the local cache is skipped. The local cache in the Python middleware contains pointers to every locally declared or retrieved instance;

when the KP developer wants to send the data to the smart space the middleware generates triples from the local cache and sends them to the SIB.

Validation The code generator generates an architecture for local ontology validation. There are two contexts to the validation architecture; the validation support created by the code generator, and the actual validation process. We have used two methods for validation of the ontology. In the first method, the ontology is validated when the KP attempts to submit local changes in the ontology to the SIB. All classes in the OWL ontology extend the top most class `Thing`, and implement the public abstract `validate(self)` interface in `Thing`. The code generator generates private validation methods for each and every class or property restriction defined in the ontology. These private validation methods are called from within the public validate method when local changes are submitted to the SIB. If a restriction has been violated, the transaction will halt and one or more exceptions are raised. In the second method the ontology is validated at runtime when modifications are locally done to the ontology. Runtime validation of the ontology has been implemented for the following restrictions: *maxcardinality*, *range*, *functionalproperty* and *inversfunctionalproperty*. If the restriction has been violated the change is discarded and the KP developer is notified. The runtime validation can be enabled or disabled at runtime.

4.4 Implementations

Code Generator 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 calls the class handler for each OWL class. The handler creates an OWL class counterpart in form of a Python class and adds it to the code model. The class handler will then list all properties and call their respective handlers, namely the ObjectProperty handler and the DatatypeProperty handler. These Property handlers will in turn call appropriate handlers for every restriction that the property may have, e.g. `Cardinality` and `Range` restrictions. Some constructs need to have full ontology generated before it can itself be generated. This includes `owl:inverseFunctionalProperty`, which is handled last in the code generation process.

```
class ControllerKP(KnowledgeProcessor):
    def initialize(self):
        self.registerOntology(CtrlOntology())
        SensorValue.onNew(self.writeSetValue)
    def writeSetValue(self, sensorValue):
        e = SetValue()
        e.addValue(sensorValue.getValue())
        self.commit()
```

Listing 1.1. Minimal implementation of an event-driven writing KP in Python

The generator is completely dependent on the reasoner, for most ontologies. The reasoner adds classes, properties and restrictions to the inferred model based on the asserted statements in the ontology, and thus relieves the code generator from manually connecting a property to the correct class or classes.

Knowledge Processor Development A minimal KP implementation written within this approach both in Python and C is illustrated in Listing 1.1 and Listing 1.2 respectively.

```
void myHandler(subscription_container_t *container){
  individual_t *e = g_slist_nth_data(container->individuals, 0);
  assert(is_classtype_of(e, CLASS_SensorValue) == 1);
  individual_t *setValue = new_individual(CLASS_SetValue);
  property_set(setValue,PROPERTY_Value,get_property_value(e,PROPERTY_Value,0)
  ); flush_to_SIB();}
int main(){
  register_ontology();
  subscription_container_t *container = new_subscription_container();
  container->handler = myHandler;
  subscribe(CLASS_SensorValue, container); }
```

Listing 1.2. Minimal implementation of an event-driven writing KP in C

5 Application Example

Our demonstration scenario 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. A conceptual model is shown in Fig. 4. All these components contain a proprietary solution, and provide only a limited set of services through their KPs. Additional services could be added to the model, and published to the SIB in order for the configuration tool to make new rules of interaction. The demonstration implementation contains a temperature service concept in addition to the house state concept shown in Fig. 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, but if there was an independent temperature sensor it could be configured to display its value as well.

The configuration tool can add and remove dependencies, rules and connections by querying the SIB. In our scenario the heater and air-conditioning agents can be configured to request changes in the State switch value, or remove their dependency. The state is indicated by an integer value. When configuration has been set up, the configuration tool KP can leave the Smart Space, as all the necessary data is contained within the ontology in the SIB.

```
def createConnection(self,srcDevice,dstDevice,srcFeature,dstFeature,name):
    c = Connection()
    c.addSourceDevice(srcDevice)
    c.addDestinationDevice(dstDevice)
    c.addSourceFeature(srcFeature)
    c.addDestinationFeature(dstFeature)
    c.addName(XSDString(name))
    self.commit()
```

Listing 1.3. Excerpt of Configuration Tool KP

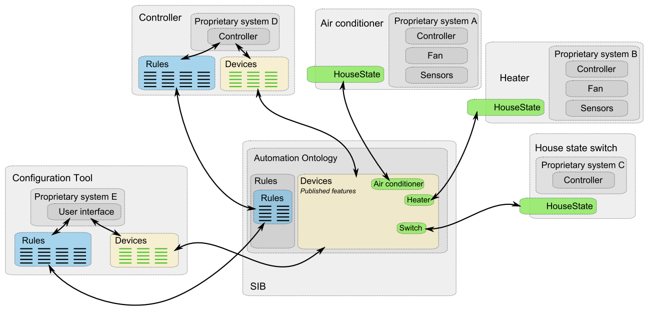

Fig. 4. Case study overview showing an interoperability solution

Example scenario: 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.

```
def handleNewEvent(self, obj):
  print "onNewEvent: "+str(obj)+" uuid: "+obj.getSourceFeatureList()[0].
       getTargetInstance().uuid
    for connection in self.connections:
      if connection.getSourceFeatureList()[0].getTargetInstance().uuid == obj
           .getSourceFeatureList()[0].getTargetInstance().uuid:
         fl = connection.getSourceFeatureList()
         for feature in fl:
             print "    feature: "+str(feature.getTargetInstance().
                  getNameList()[0].getTargetInstance().getData())
               invoke = Invoke()
               invoke.addDestinationFeature(connection.
                    getDestinationFeatureList()[0].getTargetInstance())
               invoke.addValue(XSDInt(obj.getValueList()[0].getTargetInstance
                    ().getData()))
```

Listing 1.4. Excerpt of Controller KP

Ontology: In this use-case we created an ontology containing rules for automation, concepts for expressing the house state, and the temperature. The house state is mapped to an integer value. An overview of the ontology is shown in Figure 5. In this way the configuration tool could suggest potentially inter-

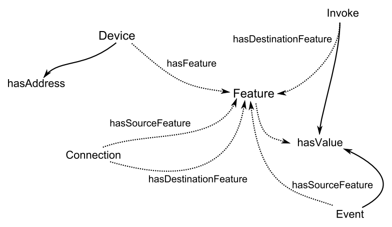

Fig. 5. Overview of the ontology used in the implementation of the case study.

esting counterparts for creating connections or rules. The ontology is expressed in OWL-DL and contains classes, data properties and object properties.

Code Generation: The Python Code Generator was used to generate the agent ontology API. The generated API populates all instance properties depending on which class it is loaded from. All properties are queried separately.

The case study illustrates application developement approach for Smart-m3 space using our proposed framework. The KPs are developed from the generated ontology API and are able to communicate through the Smart-m3 space providing the interoperability between different devices in the example application.

6 Conclusions

In this paper, we presented our approach of rapid developement of interoperable applications for smart spaces. We described our proposed framework, the appliation development tool and the overall process for application development using our tool. The case study implementation shows that agents communicating implicitly through M3 spaces can achieve the interoperability and our appraoch works well for the development of interoperable applications for smart spaces.

References

1. Jena framework: http://jena.sourceforge.net/.
2. SETH: http://seth-scripting.sourceforge.net/.
3. M. Babik and L. Hluchy. Deep integration of python with web ontology language. In *Proc. of 2nd Workshop on Scripting for the Semantic Web*, 2006.
4. J. de Bruijn, A. Polleres, R. Lara, and D. Fensel. Owl dl vs. owl flight: Conceptual modeling and reasoning for the semantic web. In *proc. of International World Wide Web Conference, 2005.*
5. A. Kalyanpur, D. Pastor, S. Battle, and J. Padget. Automatic mapping of owl ontologies into java. In *16th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2004.
6. I. Oliver and J. Honkola. Personal semantic web through a space based computing environment. In *Proc. of Middleware for Semantic Web at ICSC'08.*
7. P. Wisner and D. N. Kalofonos. A framework for end-user programming of smart homes using mobile devices. 2007.