Combining Ontology Alignment with Model Driven Engineering Techniques for Home Devices Interoperability

Charbel El Kaed[1,2], Yves Denneulin[2],
François-Gaël Ottogalli[1], and Luis Felipe Melo Mora[1,2]

[1] France Telecom R&D
[2] Grenoble University
charbel.elkaed@orange-ftgroup.com,
yves.denneulin@imag.fr,
francois-gael.ottogalli@orange-ftgroup.com,
luisfelipe.melomora@orange-ftgroup.com

Abstract. Ubiquitous Systems are expected in the near future to have much more impact on our daily tasks thanks to advances in embedded systems, *"Plug-n-Play"* protocols and software architectures. Such protocols target home devices and enables automatic discovery and interaction among them. Consequently, smart applications are shaping the home into a smart one by orchestrating devices in an elegant manner.

Currently, several protocols coexist in smart homes but interactions between devices cannot be put into action unless devices are supporting the same protocol. Furthermore, smart applications must know in advance names of services and devices to interact with them. However, such names are semantically equivalent but syntactically different needing translation mechanisms.

In order to reduce human efforts for achieving interoperability, we introduce an approach combining ontology alignment techniques with those of Model Driven Engineering domain to reach a dynamic service adaptation.

Keywords: SOA, Plug-n-play protocols, ontology alignment, MDE.

1 Introduction

Ubiquitous Systems imagined by Mark Weiser in [21] where computer systems are anywhere and invisible are not that far. Many projects from the industry WRally[1] and academia GatorTech[2] are pushing this vision further. Such systems rely on the service-oriented architecture which provides interactions between loosely coupled units called services.

Discovery, dynamicity and eventing are the main features of service oriented systems that suit best so far ubiquitous systems characteristics. In such systems,

[1] http://www.microsoft.com/whdc/connect/rally/default.mspx
[2] http://www.gatorhometech.com/

S.L. Min et al. (Eds.): SEUS 2010, LNCS 6399, pp. 71–82, 2010.

devices interact with each others and inter-operate transparently in order to accomplish specific tasks.

Smart applications are currently being deployed on Set-Top-Boxes and PC acting as control points by orchestrating home devices such as lights, TV, printers. For example a *Photo-Share* smart application automatically detects an IP digital camera device and, on user command, photos are rendered on the TV and those selected are printed out on the living room printer. The Photo-Share application actually controls such devices and triggers commands upon user request, all the configuration and interaction is completely transparent to the user who only chooses to buy Photo-Share from an application server and deploys it on his home gateway.

Plug-n-Play protocols follow the service-oriented architecture approach where home devices offer services and associated actions, for example a UPnP light device offers a SwitchPower service with two associated actions: SetTarget(Boolean) to turn on or off a light and the GetStatus() action to retrieve the actual state of the light. Currently, UPnP, DPWS, IGRS and Apple Bonjour protocols coexist in smart-homes environments but interactions between devices can not be put into action unless devices are supporting the same protocol. Furthermore, smart applications need to know in advance, names of services and actions offered by devices in order to interact with them. Equivalent device types have almost the same basic services and functions, a printer is always expected to print independently from the underlying protocol it is using. Unfortunately, equivalent devices supporting different protocols share the same semantics between services and actions but not the same syntax for identifying such services and actions. This heterogeneity encloses smart applications into specific and preselected device and service orchestrations.

Smart applications need to be set free from protocol and service syntax heterogeneity. The user must not be restrained to one type of protocol and devices, he should be able to integrate easily and transparently equivalent devices to his home environment. Existing work proposes service interoperability frameworks and techniques which starts by identifying similarities between services and functions. The identification and matching process is performed most of the time manually followed by different techniques to abstract service semantics, the process eventually ends up by applying service adaptability through template based code generation.

In order to reduce human efforts for achieving interoperability we introduce in this article an approach that combines ontology alignment techniques with those of Model Driven Engineering domain to reach a dynamic service adaptation and interaction.

The remainder of the paper is organized as follow: section 2 provides a brief overview of Plug-N-Play protocols. Section 3 overview SOA and OSGi whereas section 4 deals with Ontology Alignment and MDE techniques. Section 5, 6 describes our approach and its implementation. Section 7 discusses relevant related works. Eventually, Section 8 draws conclusions and outlines future works.

2 Plug-N-Play Protocols

UPnP [20], IGRS [10], Apple Bonjour [2] and DPWS [17], the newly standardized protocol supported mainly by Microsoft and included in Windows vista and 7, cohabit in home networks and share a lot of common points. They are all service-oriented with the same generic IP based layers: addressing, discovery, description, control and eventing. They also target the same application domains, multimedia devices are shared between UPnP, IGRS and Bonjour while the printing and home automation domains (printers, lights) are dominated by UPnP and DPWS.

Each protocol defines standard profiles specifying required and optional implementation that manufacturers need to support. Of course, vendors can extend the standards using specific notations and templates.

Even though those protocols have a lot in common, devices cannot cooperate due to two main differences:

- Device Description: expose general device information (id, manufacturer, model etc), supported service interfaces along with associated action signatures and parameters. However, equivalent device types support the same basic functions which are semantically similar but syntactically different. For instance, on a DPWS light [19], Switch(Token ON/OFF) is semantically equivalent to the SetTarget(Boolean) on a UPnP light [20]. This heterogeneity prevents smart applications to use any available device, regardless of their protocol, to accomplish a certain task such as printing or dimming lights.
- The IP based layers: plug-n-play protocols define their own underlying protocols each adapted to its environment. UPnP and IGRS uses GENA[3] for eventing, SSDP[4] for discovery and SOAP for action invocation while DPWS is based on a set of standardized web services protocols (WS-*). DPWS uses WS-Discovery and WS-MetaDataExchange to discover a device, WS-Eventing for notifications and SOAP for action invocations. WS-Security, WS-Policy are used to provide secure channels during sessions. Apple Bonjour uses multicast-DNS. Obviously, this heterogeneity between such protocols increases the complexity of device interoperability starting from the IP based layers.

  Thus, most interoperability frameworks propose a centralized approach using specific protocol proxies rather than a P2P interactions between home devices.

3 Service Oriented Architecture

The service-oriented architecture is a collection of entities called services. A provider is a service offering one or more actions invocable by entities called service clients. Every service is defined by an interface and a signature for each provided action. The service registry is an entity used by service providers to

[3] General Event Notification Architecture.
[4] Simple Service Discovery Protocol.

publish their services and by the clients to request available services in an active (available providers) or passive discovery mode (listens to service registration events). The service interface and other specific properties are used to register and request services in the service registry. The service client must know in advance the interface name of the requested service. Since the request is based on the interface name, a client requesting an interface with a semantically equivalent but syntactically different name will not receive the reference of a similar service.

OSGi, as an example, is a dynamic module system based on the service-oriented architecture. An OSGi implementation has three main elements : Bundles, Services, and the Framework. A bundle is a basic unit containing Java classes and other resources packaged in (.jar) files which represent the deployment units. A Bundle can implement a service client, a provider or an API providing packages to other bundles. The framework defines mechanisms to manage dynamic installation, start, stop, update, removal and resolution of dependencies between bundles. Once a bundle dependency is resolved, it can be started and the service can interact with other services.

In [4], **Base Drivers** are defined as a set of bundles enabling to bridge devices with specific network protocols. A base driver listens to devices in the home network then create and register on the OSGi framework a proxy object reifying the founded device. Services offered by real devices on the home network can now be invoked by local applications on the OSGi framework. The device reification is dynamic, it reflects the actual state of the device on the platform. The local invocation on OSGi is forwarded to the real device.

Currently, there is a UPnP base driver[5] implementation published by Apache, a DPWS and Apple Bonjour Base Drivers previously developed in our team [3], [4] and a DPWS base driver proposed by Schneider [19]. Base drivers solve the IP based layers heterogeneity of Plug-n-Play protocols (see section 2), but the device and service description ones remain.

4 Ontology Alignment and Model Driven Engineering

According to [12], *"An ontology is an explicit representation of a shared understanding of the important concepts in some domain of interest"*. In our work, the domain of interest is the home network and concepts of the ontology are devices, services, actions and parameters. Every real device is modeled by an ontology reflecting its specific information with the predefined concepts.

Alignment [7] is the process of finding a set of correspondences between two or more ontologies, for example finding equivalent services, actions and parameters between a UPnP and a DPWS Light Device. The correspondence between entities is expressed using a normalized similarity value within an $\mathbb{R}^+[0,1]$ interval.

Model Driven Engineering is a software development methodology based on different levels of abstraction aiming to increase automation in program development. The basic idea is to abstract a domain with a high level model then to transform it into a lower level model until the model can be made executable

[5] `http://felix.apache.org/site/apache-felix-upnp.html`

using rules and transformation languages like in template-based code generation tools. The Model Driven Architecture [18] promoted by the Object Management Group (OMG) is considered as a specific instantiation of the MDE approach. It defines standard models like UML and MOF. MDA defines four architectural layers: M0 as the instance layer, M1 for the model, M2 for the meta-model and the M3 for meta-meta-model layer.

5 Combining MDE and Ontology Alignment Techniques

Our approach is based on two techniques for device adaptability. The first one uses ontology alignment to identify correspondences semi-automatically between equivalent devices. The output of the alignment is then used as a transformation language in order to automatically generate a proxy device which will receive service invocations and adapt them to the equivalent device. All the adaptation process is transparent to the orchestrating application and the existing devices. The proxy will actually bridge syntactic heterogeneity between semantically equivalent device types and services.

Our approach follows four major steps to accomplish device interoperability:

5.1 Ontology Representation

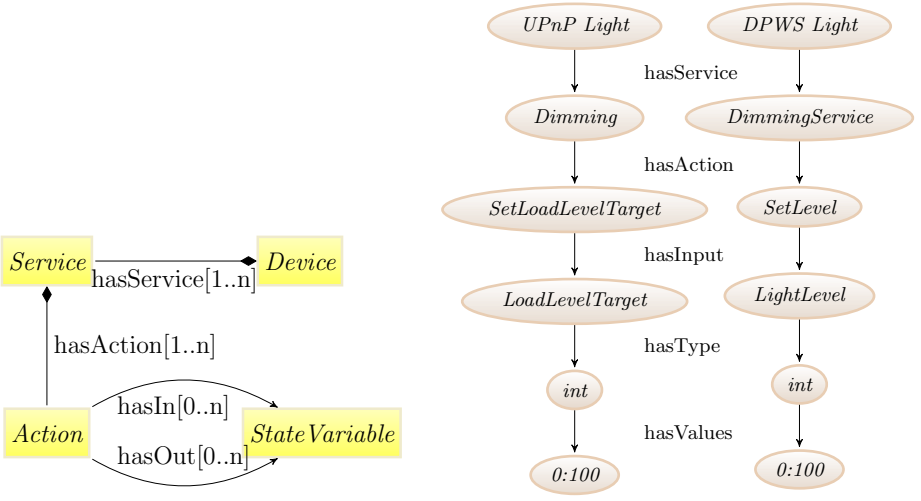
The first step aims to hide device description heterogeneity by modeling each device with common concepts using an ontology. We chose to use common concepts based on the UPnP description model. Every device is modeled with an ontology which is conformed to the meta ontology described in fig.(1 a). We use the concepts as follows: every device has one or more services, every service has one or more actions and each action has one or more input/output state variables. Now we can build device ontologies conformed to this meta ontology.

In fig.(1 b) we present UPnP and a DPWS Device lights ontologies. (For simplicity we omitted from figure (1 b) class types, device, service and properties such as service Id, name, etc).

We automated the construction of device ontologies by using specific OWL Writer bundles fig.(2 a) that listens to the appearance of Plug-N-Play devices, if the device ontology was not yet build for a device type and model (check the device type, model, services etc) then the build process can proceed and once terminated the Aligner is notified.

5.2 Ontology Alignment

The MDA defines 4 levels of architecture, the meta device ontology is on the M2 layer while the instantiated ontologies reflecting home devices are on the M1 layer. Transformation models in the MDE aim to build bridges between models linking entities between two existing models. Those bridges are actually transformation rules written manually or generated using graphical tools using specific languages like ATL [11]. Fig. 3 shows the overview of the approach. We

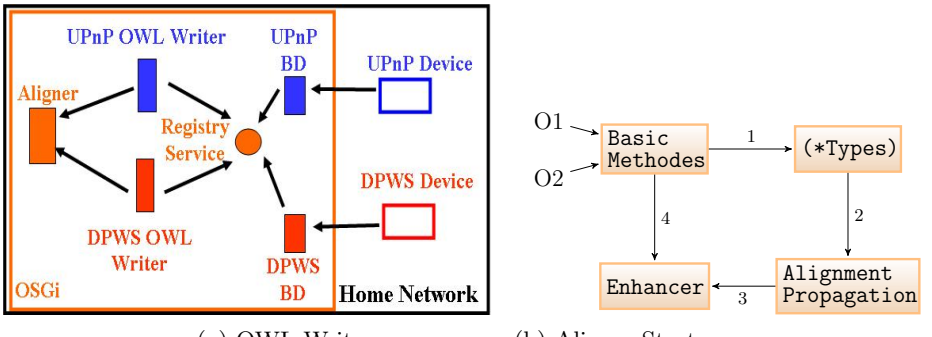


(a) M2: Meta Model Ontology (b) M1: UPnP and DPWS Light ontology

Fig. 1.

use ontology alignment techniques to match entities in a semi-automatic way fig.(2 b) by applying heuristics to match entities of the ontologies.

Step 1 of the aligner takes two ontologies $O1$ and $O2$ then apply basic matching techniques described in ([7], chapter 4) such as Hamming, Leveinshtein, smoa and other techniques based on an external dictionary *WordNet*. We propose and implement an enhanced smoa based technique using wordNet to detect antonyms (Set $\neq$ Get, Up $\neq$ Down etc) and provide more accuracy during action matching like "SetLoadLevel" $\neq$ "GetLoadLevel". Basic techniques provides those two actions as a potential match while smoa++ clearly reduces the similarity value between these two strings. The first step matches all concepts, it uses all the



(a) OWL Writers (b) Aligner Strategy

Fig. 2.

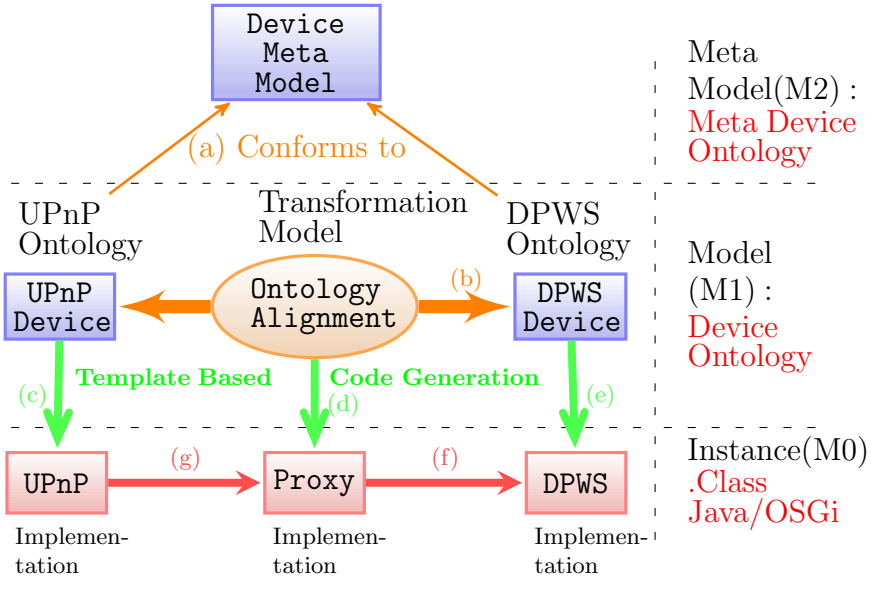


Fig. 3. Overview of the approach

available information in the ontology. We found in some cases that there is a higher similarity between a service name and an action name, we will use this information to enhance similarities in the step 4 of the aligner.

Since each basic technique has its own advantages and disadvantages, we combine all these techniques and use a weight α_k for each method k, $\alpha_k \in \mathbb{R}^+[0,1]$. We give a higher weight for methods using an external dictionary.

Step 2 of the aligner filters the alignment by applying typed classes, it keeps same concept type correspondences (device-device, service-service, etc), we use $\beta_{i,j} = 1$ if concept i and j are the same and zero if not. The output of the first and second steps is a matrix of similarities between concepts.

$Sim1_{i,j,k}$ is the similarity result after step 1 between entity i from $O1$ and entity j from $O2$ using the technique k. $Sim2_{i,j,k}$ is the output after step 2.

$$Sim2_{i,j} = \left[\sum_{k=1}^{n} (\alpha_k * Sim1_{i,j,k})\right] * \beta_{i,j} \in \mathbb{R}^+[0,1] \quad (1)$$

Depending on the devices complexity, we can choose to trim alignments and keep all similarities higher than a *threshold* and additionally use a Delta Threshold δ to keep matched concepts having a similarity differing at most by a tolerance value δ.

Step 3 of the aligner propagates similarities along the ontology structure, for example when two services have strong similarity, we enhance the similarity of their actions. Our method is a *Down Propagation* method, based on Coma++ [6] which uses an *Up Propagation* method.

Step 4 aims to enhance similarity values, if a $Service_A$ have a high similarity with $Action_B$ of $Service_B$, then the algorithm enhances the similarity between both services.

Step 5 is actually the human intervention to validate or edit correspondences between entities. The output, expressed in OWL, represents the transformation rules for the next step to automatically generate the device proxy with the right matching between devices.

In our approach, UPnP is chosen as a common model, UPnP services are substituted with non-UPnP devices. Other services are matched with UPnP services and if a match is found then we can adapt home applications on the fly to invoke non-UPnP existing services. Our choice is motivated by the wide acceptance of UPnP among device manufacturers and telecommunication operators, and the large number of devices standardized by DLNA (www.dlna.org). Those characteristics makes UPnP the most mature protocol so far among existing plug-n-play protocols and therefore the best pivot candidate for our approach.

5.3 Template-Based Code Generation

The input of this step is the ontology alignment between two devices. Since we have the correspondences between elements of the ontology then the proxy can be generated using already written templates. This process is already used to generate Java code from device description UPnP XML description and WSDL files to Java code (interfaces and classes), then it is up to developers to implement the necessary functional code (fig. 3, c and e arrows). In our approach, all the code is actually automatically generated since the correspondences between devices are provided by the ontology alignment output, fig.(1,b). The templates need only to be filled with the aligned device, service and action names along with parameters. When the application invokes the UPnP dimming action *SetLoadLevelTarget* with the parameter *LoadLevelTarget*, the proxy will invoke on the DPWS device the *SetLevel* action of the *DimmingService* service and *LightLevel* parameter as an input.

We identified another case where an action is an union of two others, such is the case of the standard UPnP and DPWS printers [20,13]. The UPnP action *CreateJobV2* (simple entries) is equivalent to two DPWS Actions *CreatePrintJob* (complex structured entries) and *AddDocument*. The mapping between DPWS. CreatePrintJob and UPnP.CreateURIJob reveals that the parameter *SourceURI* has an equivalent entry parameter *DocumentURL* for the DPWS action *AddDocument*. Consequently, CreateURIJob = UnionOf (CreatePrintJob, AddDocument). Other properties exist too, such as Sequenced-UnionOf where actions have input and output dependencies. This kind of properties can be detected easily using a reasoner to infer on correspondences between services and actions. Other actions on the printer devices are substitutable such as (UPnP.GetJob Attributes & DPWS.GetJobElements), (UPnP.GetPrinterAttributesV2 & DPWS. GetPrinterElements) and (UPnP.CancelJob & DPWS.CancelJob). These equivalent actions make the standard UPnP and DPWS printer devices substitutable.

Table 1. Mapping between a standard DPWS and a UPnP printer action

DPWS (CreatePrintJob)	UPnP (CreateURIJob)
PrintTicket/JobDescription/JobName	JobName
PrintTicket/JobDescription/JobOriginatingUserName	JobOriginatingUserName
PrintTicket/JobProcessing/Copies	Copies
X	SourceURI
PrintTicket/DocumentProcessing/NumberUp/Sides	Sides
PrintTicket/DocumentProcessing/NumberUp/Orientation	OrientationRequested
PrintTicket/DocumentProcessing/MediaSizeName	MediaSize
PrintTicket/DocumentProcessing/MediaType	MediaType
PrintTicket/DocumentProcessing/NumberUp/PrintQuality	PrintQuality

5.4 Service Adaptation

Now that all elements are ready, the proxy can be generated on demand and compiled on the fly in order to provide interoperability between devices, services and actions. For the *à la carte* application, the user can choose the DPWS (or other) device type and model, then based on the existing specific code templates and the ontology alignment on the service provider platform, the proxy can be generated and packaged for deployment. As for the on the fly adaptability, the service request will be intercepted. The correspondent proxy will be generated according to ontology alignment already existing on the gateway or downloaded from the service provider servers. Thus the Photo-Share application will work transparently for the user.

6 Implementation

In this work in progress, we used UPnP felix Apache[6] and DPWS [19] base drivers. We developed OWL Writers on an Felix/OSGi framework using the OWL API and implemented our alignment strategy using the Alignment API [8]. The output of the alignment is expressed with *OWLAxiomsRendererVisitor* which generates an ontology merging both ontologies to express relations between entities. To subsume properties like Union-Of we will use a reasoner on the output to infer such properties. The proxy is currently generated using manually pre-filled templates using Janino[7] on the fly compiler. In future work, we will fill the templates with the information from the ontology alignment.

7 Related Work

Different approaches have been developed, in the literature, to solve the issues related to the interoperation problem, the approaches can be put in three major

[6] `http://felix.apache.org/site/apache-felix-upnp.html`
[7] `www.janino.net`

categories : EASY [1] and MySIM [9] worked on frameworks allowing services to be substituted by other similar services and actions. They both model the domain in a **common ontology** holding all the concepts and properties relating them. Every service interface, action and parameter is annotated with a predefined semantic concept from the common ontology. A service is then substitutable by another if actions and their input/output parameters have similar or related semantic concepts from the common ontology. MySIM uses Java introspection in order to retrieve the annotations then compare semantic concepts and adapt the matched services and actions. The limitation of both approaches is that annotations are filled with predefined concepts, consequently when a new service type appears, the common ontology should be updated first by adding new concepts and connecting them to other existing entities in the ontology. The update process is not that obvious since a new type can have common semantics with more than one existing concept resulting an incoherent ontology. Consequently, a reorganization operation is more often required to reestablish a coherent classification between concepts in the common ontology.

The second category models the domain with an **abstract representation** like an ontology in [5] or with a meta model in [16]. In DOG, similar device types and actions are modeled with abstract ontology concepts, (light device, dimming action, switch on/off) then these concepts are mapped to specific technology actions and syntax using specific rules to fill pre-written Java/OSGi code templates. The interoperability among devices is based on abstract notifications messages and predefined association between commands (the switch OffNotification is associated to the OffCommand on a device light). Then MVEL rules are generated automatically based on the manually written associations, the rules actually invokes technology specific functions on the targeted device. EnTiMid [16] uses a meta model approach instead of the ontology then generates specific UPnP or DPWS Java/OSGi code using transformation rules and templates. Both approaches deals with relatively simple devices like lights with similar actions and parameters. However the abstraction of complex devices like printers is not trivial specially when an action on one device is equivalent to one or more actions on another equivalent device (Table 1). Besides, in both approaches, the mapping between the abstract model and the specific model is done manually by either writing transformation rules or writing predefined associations and automatically generating transformation rules.

The third category uses a **common language** to describe devices with same semantics, Moon et al. works on the Universal Middleware Bridge [15] which proposes a Unique Device Template (UDT) for describing devices. They maintain a table containing correspondences between the UDT and the Local Device Template (LDT). UMB adopts a centralized architecture similar to our approach, where each device/network is wrapped by a proper UMB Adapter which converts the LDT into UDT. Miori et al. [14] defines the DomoNet framework for domotic interoperability based on Web Services and XML. They propose DomoML a standard language to describe devices. TechManagers, one per subnetwork translates device capabilities as standard Web Services, the mapping is

done manually. Each real device is mirrored as a virtual device in other subnetworks, thus each device state change requires considerable synchronization effort among subnetworks. The HomeSOA [4] approach uses specific base drivers to reify devices locally as services then another layer of *Refined drivers* abstract service interfaces per device types as a unified smart device, a UPnP and DPWS light dimming services are abstracted with DimmingSwitch interface then it is up to the developer to test the device type and invoke the underlying specific interface.

8 Conclusion and Future Works

In this article we propose an approach based on MDE and ontology alignment techniques to bridge device and service syntax heterogeneity in order to enable service interoperation. First, we automatically generate for each device an ontology conformed to a meta ontology, then we apply ontology alignment techniques to semi-automatically retrieve correspondences between equivalent device types and services. The ontology alignment is validated by a human. The resulting output corresponds to transformation rules used in oder to generate on the fly specific proxies to enable interoperability. We choose UPnP as a central and pivot protocol. We match other protocol services with those from UPnP. This choice is motivated by the fact that UPnP is the most mature protocol among plug-n-play protocols so far. There is a large number of standardized devices (www.dlna.org) and it is widely accepted and supported by many manufacturers and vendors. The proxy is generated when there is equivalent device type to a UPnP requested device. The specific proxy publishes UPnP service interfaces and actions and once invoked it actually transfer the invoked action to the correspondent device using its own semantics and syntax. In future works, a GUI will be implemented to make the validation and correspondence editing easier. Complex mapping of services and actions (Sequential and simple Union-Of relations) will be investigated in order to infer and deduce such correspondences with a minimal human intervention.

Acknowledgment

The authors would like to thank Sylvain Marie from Schneider Electric for his help with the DPWS Base Driver integration.

References

1. Ben Mokhtar, S., et al.: Easy: Efficient semantic service discovery in pervasive computing environments with qos and context support. Journal of Systems and Software (2008)
2. Bonjour, `http://www.apple.com/support/bonjour/`
3. Bottaro, A.: Rfp 86 - dpws discovery base driver (2007), `http://pagesperso-orange.fr/andre../rfp-86-DPWSDiscoveryBaseDriver.pdf`

4. Bottaro, A., Gérodolle, A.: Home soa -: facing protocol heterogeneity in pervasive applications. In: ICPS '08: Proceedings of the 5th International Conference on Pervasive Services (2008)
5. DOG: Dog: Domestic osgi gateway, http://elite.polito.it/dog-tools-72
6. Engmann, D., Maßmann, S.: Instance matching with coma++. In: BTW Workshops (2007)
7. Euzenat, J., Shvaiko, P.: Ontology Matching. Springer, Heidelberg (2007)
8. Euzenat, J.: Alignment api, http://alignapi.gforge.inria.fr
9. Ibrahim, N., Le Mouël, F., Frénot, S.: Mysim: a spontaneous service integration middleware for pervasive environments. In: ICPS '09 (2009)
10. IGRS: http://www.igrs.org/
11. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: Atl: A model transformation tool. Science of Computer Programming 72, 31–39 (2008)
12. Kalfoglou, Y.: Exploring ontologies. In: Chang, S.K. (ed.) Handbook of Software Engineering and Knowledge Engineering. Fundamentals, vol. 1 (2001)
13. Microsoft: Standard dpws printer and scanner specifications (January 2007), http://www.microsoft.com/whdc/connect/rally/wsdspecs.mspx
14. Miori, V., Tarrini, L., Manca, M., Tolomei, G.: An open standard solution for domotic interoperability. IEEE Transactions on Consumer Electronics (2006)
15. Moon, K.d., et al.: Design of a universal middleware bridge for device interoperability in heterogeneous home network middleware. IEEE Transactions on Consumer Electronics (2005)
16. Nain, G., et al.: Using mde to build a schizophrenic middleware for home/building automation. In: ServiceWave '08: Proceedings of the 1st European Conference on Towards a Service-Based Internet (2008)
17. OASIS: Devices profile for web services version 1.1 (2009), http://docs.oasis-open.org/ws-dd/dpws/1.1/os/wsdd-dpws-1.1-spec-os.html
18. Object-Management-Group, O.: Mda guide version 1.0.1 (2003)
19. SOA4D: Service oriented architecture for devices, https://forge.soa4d.org/
20. UPnP: http://www.upnp.org/
21. Weiser, M.: The computer for the 21st century. SIGMOBILE Mob. Comput. Commun. Rev. 3(3), 3–11 (1999)