Dynamic Service Adaptation for Plug and Play Device Interoperability

Charbel EL KAED
France Telecom R&D
Grenoble University
charbel.elkaed@orange-ftgroup.com

Yves DENNEULIN
Grenoble University
yves.denneulin@imag.fr

François-Gaël OTTOGALLI
France Telecom R&D
francoisgael.ottogalli@orange-ftgroup.com

***Abstract*—Advances in embedded systems, plug-n-play protocols and software architectures bring the ubiquitous system vision to the near future. Home devices supporting such protocols can be automatically discovered, configured and invoked for a given task. Smart applications are shaping the home into a smart one by orchestrating devices in an elegant manner. 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 services names hosted by devices to interact with. However, such names are often semantically equivalent but syntactically different among devices, needing translation mechanisms. In this work we present how ontology alignment techniques assisted with pattern detection rules are used to find such correspondences between equivalent devices. Once the mapping is validated we apply a code generation technique to reach a dynamic service adaptation. We validated the approach on an HP Printer.**

***Index Terms*—OA, Home Devices, plug-n-playOA, Home Devices, plug-n-playS**

I. Introduction

Ubiquitous systems imagined by Mark Weiser in [1] are emerging thanks to the development of embedded systems and plug-n-play protocols like the Universal Plug aNd Play (UPnP)[2], the Intelligent Grouping and Resource Sharing (IGRS)[3] and the Device Profile for Web Services (DPWS) [4]. Such protocols follow the service oriented architecture (SOA) paradigm and allow automatic device and service discovery in a home network. Once devices are connected to the local network, applications deployed for example on Set Top Boxes, discover the plug-n-play devices and act as control points. The aim of such applications is to orchestrate the interactions between devices (lights, TV, printer) and their corresponding hosted services. For example a Photo-Share 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 printer. The configuration is completely transparent to the user who deploys the application on his home gateway.

Devices supporting a plug-n-play protocol announce their hosted services each in its own description syntax. A UPnP light for example hosts a `SwitchPower` service with a `Switch(true/false)` action to control the light while a DPWS light[5] uses the semantically equivalent action `SetTarget(On/OFF)`. The syntactic heterogeneity along with the protocols layers diversity, prevent applications to use any available equivalent device on the network to accomplish a specific task. Designing applications to support multiple protocols is time consuming since developers must implement the interaction with each device profile and its own data description. Additionally, the deployed application must use multiple protocols stacks to interact with the device.

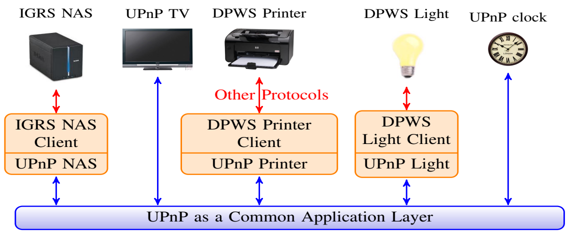


Fig. 1. UPnP as a Common Application Layer

Application vendors and telecom operators need to manage and orchestrate devices through a common application layer [6], independently from the protocol layers and the device description. To accomplish interoperability between plug-n-play devices, we propose to use the UPnP profile description and stack as a common pivot, due to its wide acceptance among device manufacturers and vendors. Moreover, a large set of tools and applications targeting UPnP devices already exist.

Our approach consist in generating proxy modules published as UPnP standard devices and to control non-UPnP devices. In Fig. 1, the proxies allow applications to interact with non-UPnP devices as standard UPnP devices. A UPnP-DPWS Proxy Light for example is exposed as a UPnP Standard Light and controls a DPWS Light through a DPWS client. When the UPnP `Switch (boolean true/false)` action is invoked on the proxy, it will translate the call and invoke the equivalent action `SetTarget (String ON/OFF)` on the DPWS Light. Using UPnP as a common model allows developers to focus only on implementing applications that use the UPnP interaction model.

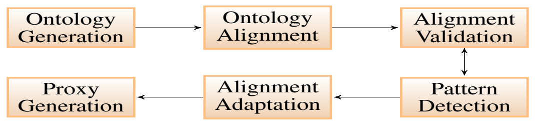


Fig. 2. Steps of the approach

To automatically generate proxies, our approach consists of six main steps as shown in Fig. 2: the first is to automatically generate ontologies from a device description. "An ontology is an explicit representation of a shared understanding of the important concepts in some domain of interest" [7]. In our work, the domain is the home network and concepts of the ontology are devices, services, actions and parameters. Each device is modeled

with an ontology reflecting its specific information.
The second step consists in applying ontology alignment techniques[8] to semi-automatically find correspondences between equivalent devices. The alignment [8] is the process of finding a set of correspondences between two ontologies, for example finding equivalent services, actions between UPnP and a DPWS Light. The alignment is based on heuristic techniques that needs an expert validation (step 3). Thus, the step four, assists the expert during the validation. We propose a pattern detection rules to automatically classify the actions that are fully compatible and those that the expert would try to adapt. The adaptation is performed in step 5, by referring to the specifications: default values, data and code adaptation.

The final step automatically generates proxies using the validated ontology alignments which represent the transformation rules to go from a UPnP standard device description to a non-UPnP device description. This step is based on the Model Driven Engineering (MDE) software development methodology. The MDE is 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 using rules and transformation languages. In our approach, we abstract the domain using ontology representation, the transformation rules are the ontology alignments used to generate executable proxies.

The remainder of the paper is organized as follows. We first overview the related work. Next, we present our ontology generation and the alignment approach. Then, we describe the pattern recognition and the code generation. Then we detail the implementation and its evaluation. Finally, we conclude with a discussion and outline future works.

II. Related Work

UPnP[2], IGRS[3] and DPWS[4], cohabit in home networks and share a lot of common features. Their protocol layers support: discovery, description, control and eventing. They also target similar device types: multimedia devices are shared between UPnP and IGRS while the printing domain (printing, scanning) is dominated by UPnP and DPWS. Each protocol defines standard profiles with required and optional implementation that manufacturers need to support.
Even if those protocols have a lot in common, devices cannot cooperate due to two main differences: the service description and the protocol layers. They all use SOAP for interaction, UPnP and IGRS use SSDP for discovery and GENA for eventing, while DPWS uses a set of standard web services protocols. To solve the networking heterogeneity, the Z2Z project [9] proposed a solution to generate protocol proxies. Other SOA frameworks like *OSGi* proposed a centralized approach using specific protocol proxies to hide the diversity. Such protocol proxies are called *Base Drivers*[10], they are defined as a set of bundles capable of representing devices with specific network protocols as local services on the SOA framework. The device reification is dynamic, it reflects the actual state of the device on the framework. Base drivers also expose local plug-n-play devices as real devices on the network. Since the distributed architecture is an SOA one, base drivers solve the networking layers heterogeneity but the device description remain.

Different approaches have been developed to solve the interoperation problem, it can be put in three major categories:

- Common Ontology: Paolucci [11], EASY [12] and MySIM [13] worked on the service substitution. They model the domain in a common ontology holding all the services and properties relating them. The services are manually classified hierarchically and semantically, for example *Wifi* is a sub concept of *Wireless* which is a sub concept of *Network*. Thus, the service description should be exposed using the same concepts from the common ontology. A service is then substitutable only with a compatible service from the ontology. They apply modifications on the service registry used to search or publish a service. In MySIM a dynamic adaptation is performed by either a redirection or a replication of the OSGi bundles *bytecode*.
Imposing to competitors such as manufacturers and standardization committees a common ontology is too optimistic[14], there had never been a unified description in the proposed standard profiles (UPnP, DPWS and IGRS) for the same device type. Annotating the description manually can be difficult and error prone. Adding a new device to the common ontology is done manually by adding new concepts and connecting them to other existing entities. Therefore, the update can produce an incoherent ontology since a new type can have common semantics with more than one existing concept[14]. In our approach, instead of manually building a common ontology with well defined concepts and semantics, independent ontologies are automatically generated from the device description. Then, correspondences between ontologies are calculated semi-automatically using alignment techniques.

- Abstract Representation: The second category models the domain with an abstract ontology : *DogOnt* in DOG[15] and an extension of the *SOUPA*[16] ontology in [17]. In both works, similar device types and actions are modeled with abstract ontology concepts, (light device, dimming action, switch on/off) as well as their interoperation messages and notifications (a light device is connected to a switch device). The abstract ontology is queried to generate interoperation rules, when an "off" message is received by the framework, the interoperation rule route the message to the target device using its own commands. The mapping between the abstract model and the specific model is done manually, meaning when adding a new device, the abstract ontology must be updated with abstract concepts, then, the interoperation between abstract and specific commands need to be hard coded. Their abstract ontology holds a lot of information, specially interoperation associations (light-switch) which makes it complex even for simple devices. Both approaches deal with relatively simple devices like lights with similar actions and parameters. However the abstraction of complex devices like printers with a 2000 lines of description is not trivial specially when an action on one device is equivalent to one or more actions on another device. In our approach, UPnP is chosen as a common model and ontologies reflecting UPnP and non-UPnP devices are automatically generated. Then, we apply semi-automatic techniques to find correspondences between ontologies which are used to generate proxies by filling template code. Our templates are manually written once but used with any device alignment holding the transformation rules between devices.

- Common Language: The third category uses a common language to describe devices with the same semantics, Moon provides the Universal Middleware Bridge[18] 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). Miori et al. [19] define the DomoNet framework for domotic interoperability based on Web Services. They propose DomoML a standard language to describe devices. Then *Tech-Managers* translate device capabilities as web services. The mapping is done manually and no adaptation and generation mechanism are proposed. The HomeSOA[10] approach uses 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. For example, a UPnP and DPWS light dimming services are abstracted with a DimmingSwitch interface then it is up to the developer to test the device type and invoke the underlying specific action. This category is dependent on the manual annotated mapping between the different descriptions.

III. Proposal

Our approach[20] is overviewed in Fig. 2 and detailed in Fig. 3. The first (MDE) M0 layer in Fig. 3 represents the heterogeneous plug-n-play devices descriptions expressed in different formats: UPnP uses an XML format while DPWS and IGRS use the standard Web Service Description Language (WSDL). Each description uses different semantics: UPnP uses devices, services, actions and state variables while the WSDL uses services, port types, operations and messages.
The automatic generation of ontologies lifts the device description from the M0 layer to the M1 layer, see Fig 3, *a* arrows. Each ontology represents a device using unified concepts in conformance to the meta model based on UPnP in the M2 layer. The meta model[20] defines the concepts as follows: every device has one or more service, every service has one or more action and each action has variables.
To resolve the heterogeneity in the M1 layer, semi automatic alignment techniques are applied to find the correspondences between equivalent ontologies, Fig. 3, *b*. The alignment is heuristic based, therefore an expert has to validate the correspondences. Thus, rules are applied on the alignment to detect patterns in order to assist the expert during the validation. Finally, based on the alignment, the automatic code generation techniques allow to go from an independent technology representation in the M1 layer to an executable proxy in the M0 layer, Fig 3, *c*. The proxy exposed as a standard UPnP device transfers the received invocations to non-UPnP devices, Fig. 3, *d*.

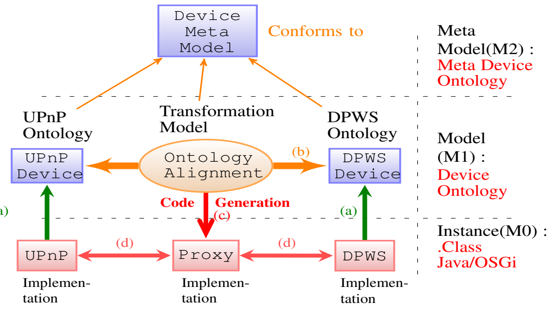


Fig. 3. Overview of the approach

A. Automatic Generation of Ontologies

Since the plug-n-play devices announce their description on the network, software entities (UPnP, DPWS *OWL Writers*) scanning the local network automatically generate ontologies (M1 layer) conformed to the meta-model to represent the device. The ontology is represented using the Ontology Web Language (OWL)[21]. Each ontology describes a device, its hosted services and actions along with the variables and their types, see Fig. 4-a, describing a part of a light ontology. An OWL writer also generates ontologies from device files description.

B. Ontology Alignment

Now that the M1 models are generated and are conformed to the M2 layer, we apply transformation techniques to go from a model to the other in the MDE M1 layer. Transformation models in the Model Driven Engineering aim to build bridges linking entities between two existing models. In our approach, the translation between equivalent devices is given by the ontology alignment (see Fig. 3, M1 Layer, b).

Fig. 4-b, shows an automatically calculated alignment using the SMOA[22] method between a UPnP and DPWS light ontologies. The matching between entities is expressed using a normalized similarity value within an $\mathbb{R}^+$[0,1] interval showed on the lines. The alignment takes two ontologies O_1 and O_2, then, applies basic matching techniques described in [8] such as Leveinshtein, SMOA and others. Such techniques are heuristic based and match two actions like "SetLevel" $\neq$ "GetLevel", therefore, we enhanced the *SMOA*[22] technique into *SMOA*++ using wordNet[23] to detect antonyms (Set $\neq$ Get) and synonyms (clock $\cong$ timer). We also applied a similarity propagation on the structure between entities, for example, we enhance two services similarity if their actions have strong similarities. A trimming is applied on the alignments, i.e. only matchings having a similarity value above a defined threshold t are kept. The result is expressed in an alignment format[24] with tuples (leftEntity, rightEntity, similarity) which can be saved in a data base. The matching between `SetTarget` and `Switch` added by the expert has a similarity set to 1 to avoid its removal when the trimming is applied.

Since the alignment is heuristic based, an expert intervention is needed to validate the mappings which is done in two steps. First the expert edits, removes or adds a matching, for example, SetTarget $\equiv$ Switch, Fig. 4, (b). However, a matching between two actions is not enough to presume that they are compatible, their input/output parameters need to be considered. Therefore, the second step consists in using patterns to detect compatible actions based on their input/output matched parameters. The alignment step and the expert validation is performed using ATOPAI [25] our Aligning and Annotation Framework for Plug-n-Play Device Interoperability.

IV. Pattern Recognition

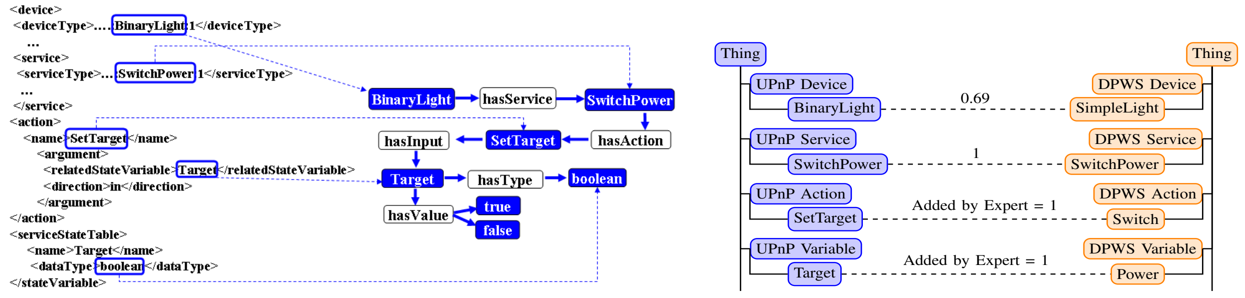
The pattern recognition is "as a classification of input data via extraction of important features from a lot of noisy data" [26]. In our approach, we use patterns to automatically classify the alignments in order to detect valid matchings between actions and let the expert focus only on non valid actions. The expert can make two actions valid by adding default values or adaptation code by referring to the specifications. In simple cases we have one-to-one mapping actions, Switch(true/false) $\equiv$ SetTarget(ON/OFF) with one parameter. However, on the standard UPnP and DPWS printers [2], [27], the UPnP action *CreateURIJob* is equivalent to the association of two DPWS actions *CreatePrintJob* and *SendDocument* and a large number of parameters. Therefore, the detected patterns guide the expert on the remaining actions to adapt. The rules are expressed in the Ontology Pre-Processing Language[28] (OPPL). Another rule language can be also be used instead. For space limitation we detailed only one rule in OPPL.

A. The Patterns

Once the ontology alignment is performed and validated, we apply rules to detect patterns and add new information to the ontology. The following paragraph presents some definitions:

Definition 1 (y = f(x)).

$$\forall x, y \in P /\ P\ set\ of\ parameters, f \in A /\ A\ set\ of\ actions$$



```
<device>
 <deviceType>....:BinaryLight:1</deviceType>
   ...
 <service>
  <serviceType>...:SwitchPower:1</serviceType>
  ...
 </service>
<action>
   <name>SetTarget</name>
     <argument>
       <relatedStateVariable>Target</relatedStateVariable>
       <direction>in</direction>
     </argument>
</action>
<serviceStateTable>
    <name>Target</name>
    <dataType>boolean</dataType>
</stateVariable>
```

Fig. 4. (a) Part of a Generated UPnP Light Ontology (b) Part of a UPnP (left) and DPWS (right) Light Alignment

$$y = f(x) \iff f \; hasInput \; x \; and \; hasOutput \; y$$

Definition 2 (f$\equiv$g).

$$\forall f, g \in A, \; f \equiv g \iff (f, \; owl\text{:}equivalentTo, \; g)$$

The actions f and g are related in the ontology with the OWL object property `owl:equivalentTo`.

The patterns used in our approach are the following:

1) ***Direct_Mapping*** is a pattern between two classes, having a one-to-one simple mapping. *For simplicity of presentation, two equivalent parameters will have the same name*. This property has the following three sub-properties:
   a) ***Direct_Mapping_Input***: two actions having at least one matching of input parameters, $f(x) \equiv g(x)$. For example, Switch(true/false) $\equiv$ SetTarget(ON/OFF). Listing 1 shows the OPPL Rule used to detect this pattern.

```
?f:CLASS, ?g:CLASS, ?x1:CLASS, ?x2:CLASS
// Select a UPnP action (f) having an input some variable x1
SELECT ?f subClassOf has_UPnP_Input some ?x1,
// Select a DPWS action (g) having an input some variable x2
?g subClassOf has_DPWS_Input some ?x2,
// The actions and the variables must be equivalent
?f equivalentTo ?g, ?x1 equivalentTo ?x2
BEGIN //if selection not empty add property between actions
ADD ?f subClassOf DirectMappingInput some ?g
```

Listing 1. OPPL Direct_Mapping_Input Rule

   a) ***Direct_Mapping_Output***: two actions having at least one matching of output parameters. For example, (Power=GetStatus()) $\equiv$ (Status=GetStatus()).
   b) ***Direct_Mapping_Input_Output***: is the association of the previous patterns.
2) ***Union_Mapping***: an action equivalent to two or more actions with no predefined order. One-to-N:

$$f(x,y) \;\equiv\; \textbf{\textit{Union\_to\_n}} \begin{cases} g(x) \\ h(y) \end{cases}$$

For example, SetClock(hour, date) is equivalent to the union of SetHour(hour) and SetDate(date).
3) ***Sequential_Union***: is a union mapping with predefined order between actions. One-to-N:

$$(y = f(x)) \;\equiv\; \textbf{\textit{Sequential\_to\_n}} \begin{cases} (1) \quad a = g(x) \\ (2) \quad y = h(a) \end{cases}$$

When an application invokes $y = f(x)$, the proxy first invokes $a = g(x)$ then $y = h(a)$ and returns y. We detected this pattern on the standard Printer devices CreateURIJob $\equiv$ **Union_Mapping (CreatePrintJob, SendDocument).** In order to detect the sequential union mappings, we first detect the union mapping then the sequential dependency (***has_Next***) between actions on the same device. ***has_Next*** is a binary relation defined as follows:

Definition 3 (has Next). $\forall f, h \in A$,
$f \; has\_Next \; h \iff$

$$\begin{cases} (1) \;\; f \neq h \;\; \text{and} \;\; Output(f) \cap Input(h) \neq \emptyset \\ \qquad\qquad \text{and} \;\; Output(h) \cap Input(f) = \emptyset \\ \quad (or) \\ (2) \;\; \exists \, g \in A \; / \; f \; has\_Next \; g \; \text{and} \; g \; has\_Next \; h \end{cases}$$

We detect this pattern by applying 3 rules, the first rule detects the first clause of the *has_Next* definition. The second rule is used to detect a cycle between two actions, (g has_Next h $\wedge$ h has_Next g). Then, the expert validating the ontology is notified and the cycle is broken by removing the *has_Next* properties. The third rule is used to detect the transitive clause. Fig. 5 shows a complex mapping between an action f and a set of union and sequential actions. The *has_Next* relation is modeled with a DAG Directed Acyclic Graph: G. Since *has_Next* is a partial order relation: irreflexive, asymmetric and transitive. G is sorted to an ordered graph (Fig. 5) using a sorting algorithm[29] during the code generation.

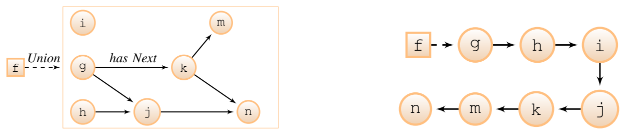


Fig. 5. (a) Complex Mappings (b) Possible Solution

The patterns N-to-One and N-to-M are ignored for now, since the invocation is unpredictable. The two actions are mapped to a single action, there is no guarantee that the application will invoke both actions g and h.

$$\begin{Bmatrix} g(x) \\ h(y) \end{Bmatrix} \;\equiv\; \textbf{\textit{Union\_to\_n}} \;\; f(x,y)$$

B. The Matching Concept

Based on the equivalent input and output parameters along with the previously detected mapping patterns relating two (simple) or more actions (union or sequential), we can automatically decide if the mapping is valid or needs to be adapted by the expert. Therefore, we define the matching concepts but first we will go through some definitions.

Definition 4 (Matching Definitions).

- $\forall a \in A,\ np_{Input}(a)$ **is the number of parameters the action *a* has as input.**
- $\forall\, a, b \in A,\ nbEqual_{Input}(a, b)$ **is the number of equivalent input parameters between actions *a* and *b*.**
- $np_{Common}(a \cap b)$ **= number of common parameters between actions *a* and *b*.**
- **S is a set of actions,** $\forall n \in \mathbb{N}^{+*}$ **number of actions,** $a_i \in A$, $S_{a_n} : \{a_1, a_2, \ldots a_n\}$.

The $MConcept_{Input}$ **takes two sets of actions** S_{a_n}, S_{b_m} **(see tab. II) and returns a value in** $\mathbb{R}^+$**[0,1] based on the equivalence of their actions and input parameters.**

Definition 5 (The Matching Concepts).

$$\forall\, S_{a_n}, S_{b_m} /\, a_i, b_j \in A,\ n, m \in \mathbb{N}^{+*},$$

$$MConcept_{Input}(S_{a_n}, S_{b_m}) = \frac{nbEqual_{Input}(S_{a_n}, S_{b_m}) * 2}{Parameters_{Input}(S_{a_n}, S_{b_m})}$$

- $nbEqual_{Input}(S_{a_n}, S_{b_m}) = \sum_{i=1}^{n} \sum_{j=1}^{m} (nbEqual_{Input}(a_i, b_j))$.
- $Parameters_{Input}(S_{a_n}, S_{b_m}) = np_{Input}(S_{a_n}) + np_{Input}(S_{b_m}) - np_{Common}(S_{a_n}) - np_{Common}(S_{b_m})$.
- $np_{Input}(S_{a_n}) = \sum_{i=1}^{n} np_{Input}(a_i)$.
- $np_{Common}(S_{a_n}) = np_{Common}(\cap_{i=1}^{n} a_i)$.

$MConcept_{Input}(S_{a_n}, S_{b_m}) = 1$. **If** S_a, S_b **have only outputs.**

We extend Paolucci's[11] four matching degrees between matched services: Exact, PlugIn, Subsumes and Fail, are applied on concepts that belong to the same ontology to detect compatibilities between services. Paolucci uses a reasoner to determine the compatibility between concepts based on the manual hierarchical classification in the common ontology. In our approach, we only have the equivalent relations provided by the alignment and the patterns. Therefore, we redefine the following matching degrees between two sets of actions S_a, S_b**, as follows:**

- $ExactMatch_{Input}(S_a, S_b)$**: for each input parameter of** S_a **there is an *equivalentTo* relation with each input parameter of** S_b**. For example, f(x,y) and g(x,y). The *ExactMatch* applies if** $MConcept_{Input}(S_a, S_b) = 1$.
- $PlugIn_{Input}(S_a, S_b)$**: for each input parameter of** S_b **there is an *equivalentTo* relation with some input parameters of** S_a**. For example f(x,y,z)** $\equiv$ **g(x,y). The parameter `z` can be ignored during the invocation since there is no equivalence on the action** g**. The *PlugIn* applies if:** $\{(MConcept_{Input}(S_a, S_b) \neq 1) \wedge (np_{Input}(S_b) = nbEqual_{Input}(S_a, S_b)) \wedge (np_{Input}(S_a) > np_{Input}(S_b))\}$.
- $Subsume_{Input}(S_a, S_b)$**: for each input parameter of** S_a **there is an *equivalentTo* relation with some input parameters of** S_b**. For example, f(x,y)** $\equiv$ **g(x,y,z). The *Subsume* matching degree do not guarantee a successful translation between actions, since some values of** S_b **are missing. The parameter `z` cannot be ignored, `g` is expecting a value. Therefore, it is up to the expert validating the alignment to verify the specifications and check if the parameter `z` can have a default value or other adaptation operations using ATOPAI. The *Subsume* applies if:** $\{(MConcept_{Input}(S_a, S_b) \neq 1) \wedge (np_{Input}(S_a) = nbEqual_{Input}(S_a, S_b)) \wedge (np_{Input}(S_a) < np_{Input}(S_b))\}$.
- $Unknown_{Input}(S_a, S_b)$**: for some input parameters of** S_a **there is an `equivalentTo` relation with some parameters of** S_b **and does not verify any previously defined matching concept. For example, f(x,y,z)** $\equiv$ **g(x,b,c).**

Table I is used to decide if the mapping between actions is a success (the translation between actions is satisfied), a failure or an undefined (the decision x/? is harder to adapt since the adaptation depends on the output parameters). Table II, shows the decision returned to the expert by ATOPAI [25]. Since, the first two actions are successful, the expert focuses then only on the undefined decisions to check if it can be turned to success. The CreateURIJob mapping (Table II) is turned to success by setting the *LastDocument* to true as a default value of the action SendDocument. The GetPrinterAttributes (Table II) is turned to success by setting the PrinterStatus as default input of the action GetPrinterElements. All the successful mappings are used for the code generation and the failures are ignored.

TABLE I
DECISION TABLE, (✓:SUCCESS, X:FAIL, ?:UNDEFINED)

(S_a, S_b)	$Exact_{In}$	$PlugIn_{In}$	$Subsume_{In}$	$Unknown_{In}$
$Exact_{Out}$	✓	✓	?	?
$PlugIn_{Out}$	x/ ?	x/ ?	x/ ?	x/ ?
$Subsume_{Out}$	✓	✓	?	?
$Unknown_{Out}$	x/ ?	x/ ?	x/ ?	x/ ?

TABLE II
EQUIVALENT ACTIONS FOR UPNP-DPWS STANDARD PRINTERS

UPnP Action (S_{a_1})	DPWS Actions (S_{b_m}, m = 1,2)	Matching	Decision
CancelJob	CancelJob	$Ex._{In}$, $Ex._{Out}$	✓
GetJobAttributes	GetJobElements	$Ex._{In}$, $Sub._{Out}$	✓
CreateURIJob	Seq.(CreatePrintJob,SendDocument)	$Sub._{In}$, $Ex._{Out}$	?
GetPrinterAttributes	∪(GetPrinterElements,GetActiveJobs)	$Sub._{In}$, $Sub._{Out}$	?

C. Expert Adaptation

Rules automatically detect patterns, however, not all the correspondences between actions are simple and can be resolved only by linking the entities. The adaptation might need data conversions and loops, for example a temperature conversion, $°C = (5/9)(°F - 32)$**. We detail in the following paragraph a use case where the alignment is insufficient and adding code is necessary. Since, the UPnP and DPWS APIs require an advanced knowledge in the protocols, we offer the expert a simple high level `Adaptation API`[25] to add the conversion operations which are injected in the templates in specific call points during the code generation.**
Use Case: let an ontology $O1$ **representing a device** $D1$**, with an action** $a1$**. An ontology** $O2$ **representing a device** $D2$**, with the actions** $a2$**,** $a3$ **and** $a4$ **which increases/decreases the volume value only by 1 unit.**

$$(a1)\ SetVolume(newVol)\ ?\ \begin{cases} (a2)\ vol = GetVolume() \\ (a3)\ VolumeUp() \\ (a4)\ VolumeDown() \end{cases}$$

The action $a1$ **`SetVolume` has no direct equivalence with the actions** $a2$**,** $a3$**,** $a4$**, however the following adaptation is possible. In our approach, the generated proxy is exposed as a device** $D1$ **and interacts with a device** $D2$**. When the `SetVolume` is invoked, the proxy first retrieves the `newVol` value, (See Listing 2, line 2) then invokes the `GetVolume` and retrieves the current volume level `vol` on the** $D2$**, (lines [3,4]). Then based on the difference between the `newVol` and the `vol`, the proxy should increment or decrement the volume value on** $D2$**. The**

expert can also invoke external services on the OSGi framework using `ExtService`[25].

```
DPWSAction a2 = getRightAction(”GetVolume”); ...
int v1 = (Integer) this.getInputValue(”newVol”) ;
a2.invoke(); // Retrieves the current volume on D2
int diff = v1 −(Integer) a2.getRetValue(”vol”);
if(diff>0) { for(int j=0; j<diff; j++) a3.invoke();}
else{ for(int j=0; j<−diff; j++) a4.invoke();}
```

Listing 2. Adaptation Code

V. Runtime Code Generation

To automatically generate a proxy for each equivalent non-UPnP device based on the alignment ontology, we propose DOXEN, a Dynamic Ontology-based proXy gENerator, installed on a *SetTopBox* for example. The alignment contains the UPnP and DPWS device descriptions and the mappings between equivalent entities (devices, services, actions and variables). Therefore, we can generate using predefined templates a proxy exposed as a UPnP and has DPWS client to interact with real devices having the same description.

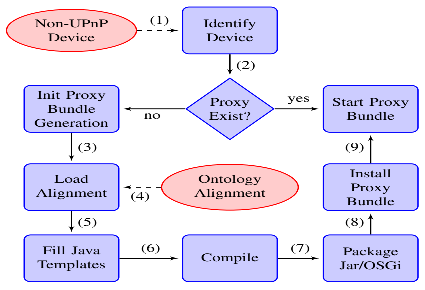


Fig. 6. Proxy Generator Diagram

DOXEN parses a configuration file containing information about the equivalent devices, the alignment file and the repository to download additional alignments. DOXEN is notified by the OSGi framework on the arrival of non-UPnP devices (Fig. 6) (step (1)), then it checks (2) the device type (”Light”, ”Printer”, etc), the version number and its hosted services. Based on such information, DOXEN loads the alignment ontology file (4) and fills (5) the pre-written Java templates based on the matched services, actions and variables in the alignment. Once the Java files are generated, DOXEN compiles the files (6) and builds an OSGi/Jar bundle (7) at runtime. Then it installs (8) and starts (9) the new generated bundle which corresponds to a UPnP proxy for the non-UPnP device. The proxy requests the general device information (manufacturer, model, friendly name, ID, etc) from the non-UPnP device and publishes the same retrieved information during its annunciation on the network. The user or the application identifies the proxy using the type and the friendly name, for example, ”HP Printer”. As soon as the non-UPnP device leaves the network, the proxy OSGi bundle goes from ”Start” state into ”Installed”. When the non-UPnP device re-appears, DOXEN starts the proxy bundle again.

VI. Global Architecture

The previously detailed modules are used as follows in the global architecture (see Fig. 7). OWL Writers can be deployed at the operator's or the client's site to generate and upload ontologies when a non identified device is discovered.
On the operator site, the expert retrieves the generated ontologies from a Data Base to align and validate using ATOPAI (2). The alignment is then deployed (3) on the home network,

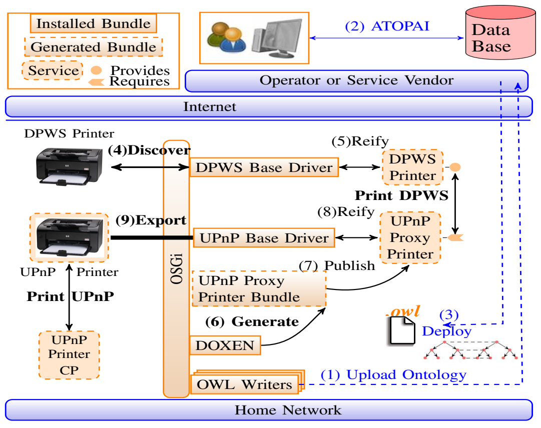


Fig. 7. Global Architecture

where DOXEN is installed. DOXEN can also be installed at the operator's site, then the automatically generated proxies (bundles) will be remotely installed in home networks.

In Fig. 7, when a DPWS device appears on the network, the DPWS Base Driver (BD) discovers (4) and reifies (5) the device as an OSGi DPWS Printer Service. DOXEN detects the OSGi DPWS Printer Service, checks the list of the equivalent DPWS devices, then, generates (6) a UPnP Proxy Printer (OSGi) bundle which publishes (7) the OSGi UPnP Proxy Printer Service. The UPnP BD reifies (8) the new UPnP Service and exposes it (9) as a UPnP device on the network. Any invocation on the UPnP exposed device is handled by the UPnP Proxy Printer OSGi Service and forwarded to the OSGi DPWS Printer Service. The invocation on the OSGi DPWS Service is reified by the DPWS BD to the DPWS Printer.

When a new device type appears, the correspondent OWL Writer generates and sends the ontology to the operator. Once matched and validated with an existing equivalent UPnP device, the alignment is deployed in the home network. If the non-UPnP device does not have an equivalent UPnP device, then, there is no need for adaptation. In this case, there would not be a UPnP application searching to interact with such device type.

VII. Implementation

To implement our approach, we used the UPnP Felix Apache [30] and the DPWS SOA4D[5] base drivers. We developed the OWL Writers on a Felix/OSGi [30] framework using the OWL API 3[21]. We implemented ATOPAI [25] to help the expert during the alignment validation. ATOPAI is based on the Alignment API 4[24] and exposes a GUI based on the Swing API, it allows to load ontologies, compute and validate alignments, detect patterns and insert adaptation code. The expert adds an alignment by selecting two entities and using the new cell button. To remove an alignment, the expert removes the line between the two entities. ATOPAI supports the SMOA and SMOA++ alignment techniques, additional techniques can be added through the Alignment API [24], it allows to use an external dictionary and to select the trimming threshold.
We implemented DOXEN using the FreeMarker [1] template engine to generate the proxy Java code and Janino[2] to compile at runtime. Currently we generate proxies at runtime using an alignment ontology for devices such as a DPWS light[5], a WS4D

[1]http://freemarker.sourceforge.net
[2]www.janino.net

[3] clock, a DPWS Standard printer. We also validated the code injection and the external OSGi calls.

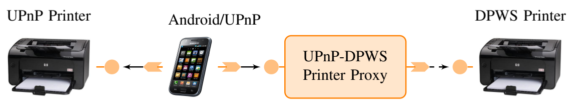


Fig. 8. Android/UPnP Control Point

We implemented a ”Home Controller” Application on a *Samsung GalaxyS Android* (2.1) smart phone which interacts only with UPnP standard Lights, Clocks and Printers. On the *GalaxyS*, we only deployed a UPnP *Cybergarage*[31] stack and deployed on a *SodaVille* (Set-Top-Box) an OSGi/Felix Framework with DOXEN along with the UPnP and DPWS Base Drivers. On the appearance of the DPWS light [5], the DPWS WS4D clock or the DPWS HP 4515x Printer, DOXEN generates at runtime a UPnP-DPWS proxy for each device. The application controlled all the DPWS devices through the generated UPnP proxies which transfer the invocations to the real DPWS devices. On the HP printer, we are successfully able to print a file(pdf, txt, ps), cancel a job, retrieve the job and the printer status.

VIII. Evaluation

We tested our approach on an Intel x86 Centrino Duo Core PC, with 2 GHz clock frequency and 1 GB RAM capacity. The OWL Writers generated ontologies for Lights, Clocks and Printers see Table III, shows the time in seconds for the ontology generation per device type, the lines of code (LoC) of the ontology and LoC of description file of each device. The difference in the building time is due to the following: first, DPWS devices have a larger description files (DPWS Printer 2237 LoC vs UPnP Printer 610 LoC) and complex hierarchical parameters [27]. Base drivers[30] (BD) represent real devices as local OSGi services, the UPnP BD represents all the device description on OSGi. However, the SOA4D DPWS BD only represents the device and service information ignoring the operations and the parameters. Consequently, the DPWS OWL Builder retrieves the WSDL files embedded on the real DPWS device, then, generates the ontology, while the UPnP OWL Writer has access to the information from the UPnP BD. An enhancement in the DPWS BD reduces the difference in the generation time.

TABLE III
Generated Ontologies on the PC

Device Type	Time (seconds)	Description (LoC)	OWL (LoC)
UPnP Printer	0.5	610	1573
DPWS Printer	187	2237	9082
UPnP Light	0.2	51	365
DPWS Light	0.8	213	245
UPnP Clock	0.05	49	161
DPWS Clock	0.25	48	123

We tested the alignment on different devices using our implemented method and algorithm SMOA++. Table IV resumes the alignment evaluation on three device types using the SMOA[22] matching technique and our proposed algorithm SMOA++. It exposes the method used, the alignment time in seconds, the success percentage for a threshold S.(t1=0.63) and S.(t2=0.25) as well as the number of false matches f.(t1) and f.(t2) detected as positive correspondences. The SMOA++ method uses SMOA with an external semantic dictionary *WordNet* to align entities. It also applies a structure enhancement and takes into account the parameter types and range. The alignment methods detect up to 78% of the correct mappings, then it is up to the expert to update the alignment using ATOPAI. SMOA++ has a higher calculation time due to the access and search in the semantic dictionary WordNet[23] and the structural enhancing. The difference in time remains acceptable since the alignment is treated off line at the operator platform. SMOA++ has an overall better performance than SMOA, the successful detected correspondances are higher and the false detected correspondances are generally lower. A detailed alignment evaluation of the ontologies can be found at [25].

[3]Web Service for Devices: http://www.ws4d.org/

TABLE IV
Alignment Evaluation on the PC

Type	meth.	time	S.(t1=0.63)	f.(t1)	S.(t2=0.25)	f.(t2)
Printer	smoa++	824	71%	2	**78%**	**8**
	smoa	77	71%	6	75%	11
Light	smoa++	7.3	50%	1	**66%**	3
	smoa	1.5	50%	4	**66%**	4
Clock	smoa++	2.5	50%	1	**66%**	1
	smoa	0.4	50%	1	66%	1

Table V resumes the time spent by DOXEN to generate and install a proxy, the lines of code of the generated Java files and the Jar bundle size. The table also shows the number of Java template files and lines of code (LoC) used. We evaluated DOXEN on the PC and a *SodaVille* Set-Top-Box (STB) with an Intel Atom 1.2 GHz, 384 MB of RAM and an open-JDK 6 implementation. DOXEN spends less than 2 seconds on a PC and 10 sec on the STB to parse the alignment and generate a UPnP-DPWS proxy for complex standard printers interoperability. The Jar size can be reduced if another optimized compiler is used instead.

TABLE V
Generated Proxies

Proxy	Time (sec) STB	Time (sec) PC	Java files	LoC	Jar(KB)
Templates	–	–	8	1112	–
Printer	10.1	1.7	31	3325	67
Light	4.7	0.94	15	1812	37
Clock	3.7	0.85	9	1151	22
UC [TV]	3.4	0.77	9	1198	25

Table VI shows the invocation time of a DPWS Client invoking directly the actions on the HP 4515x Printer and of a UPnP Client invoking through the generated UPnP Proxy. The (UPnP, DPWS) clients and the generated proxy where deployed on the same STB and connected to the printer through the local wireless network. The clients printed the same file. The results show a small difference in the invocation time due to the translation cost.

TABLE VI
Printer action invocation Time (seconds)

Actions	DPWS Client	UPnP Client
(CreatePrintJob, SendDocument)	0.84	1.14
CancelJob	0.4	0.57
(GetPrinterElements, GetActiveJobs)	0.33	0.58
GetJobElements	0.36	0.48

IX. Discussion

To achieve semantic interoperability using ontologies, there is two main types of approaches in the literature:
Integrated[32]: A global ontology is used to represent other domains (UPnP, DPWS, other). Correspondences between the global ontology and such domains are established manually. The main challenge remains in the construction which is an iterative task[32] since common and unified semantics need to be found to represent heterogeneous resources. However, a

central ontology will never be large and compatible enough to include all concepts of interest of every domain[14], so it will have to be updated, modified, extended and even matched with another ontology[14]. Each new extension will be different and can create conflicts between predefined concepts and semantics resulting with an inconsistent ontology[32]. Besides, an update in any domain requires an update of the global ontology.

Federated[33]: No common ontology is used, only correspondences between different ontologies are established using ontology alignment techniques. The challenge consists in finding correspondences between the pivot and each domain. The alignment techniques are semi-automatic and are based on the syntax, the semantics and the structure[8]. A human intervention is needed to validate the detected correspondences. Since the mappings are independent, an update in a domain requires an update of the concerned mappings. If the pivot was updated then all the mappings will be updated. There is also the *Unified* method combining both approaches and inheriting their challenges.

In the related works, the *integrated* approach with a common ontology is applied on relatively simple devices consequently ontologies proposed in their approach are relatively simple comparing to printers with a description of 2237 lines of code (LoC). Therefore the manual construction of large ontologies is time consuming and error prone regarding its complexity.
In our approach, devices announce their description, thus OWL Writers generate ontologies conformed to a meta-model. Even though, 187 sec (≈ 3 minutes) Table III is spent to generate complex DPWS Printer ontology and 824 sec (≈ 13 min) to align the printers (see Table IV), it remains acceptable and fast rather than manually building a common ontology of relatively complex devices as suggested in the related works. Using a common ontology for devices seems to be too optimistic[14] among competitors such as manufacturers and standardization committees, there had never been a unified description for the same device type. Imposing the use of a common ontology is similar to imposing a unified standard profile.

To our knowledge, we are the first to resolve heterogeneity between two standard profile printers using ontology alignment techniques and applying interoperability on a real DPWS printer. The specifications are protocol and technology independent, therefore, the expert performing the validation off line on the operator site using ATOPAI can be a technician or a domain expert. For the printers validation, the authors validated the mappings by referring to the standard printers profiles[27], [34]. The validated alignment can then be deployed on the client gateway so it can be used later by DOXEN. The main advantages of this approach are the following: first, already installed applications (Fig. 8) which targets only standard UPnP devices can now interact with other non-UPnP devices thanks to the dynamically generated proxy. The Home Application on the smart phone interacts with a DPWS HP Printer as a standard UPnP Printer. Second, there is no need to add additional networking stacks to support other protocols on devices hosting applications. The same UPnP stack already deployed is used to interact with other devices supporting a different protocol via the proxy. We only deployed a UPnP stack on the smart phone, there is no need to deploy a DPWS stack to interact with DPWS devices. Third, the proxy is automatically generated in a less than a minute without a human intervention. All the code generation, compilation and installation is automatic and transparent to the user and applications. And finally, The construction process of ontologies is relatively simpler and faster than building a global common ontology specially when dealing with complicated devices like printers. The ontologies can also be reused if another protocol is chosen as a pivot. The expert validating the alignment using our ATOPAI needs only to remove or add lines between two equivalent entities and can add default values or adaptation code using a high level API.

X. Conclusion and Future Works

In this article we propose an approach based on code generative and ontology alignment techniques to bridge device and service heterogeneity. First, we automatically generate for each device type an ontology conformed to a meta model, then we apply ontology alignment techniques to semi automatically retrieve correspondences between equivalent devices. Then an expert validates the alignments assisted with pattern recognition techniques to classify equivalent actions and compositions. The validated alignment is a set of transformation rules used to generate on the fly specific proxies from existing templates. The proxy generation is triggered on the device appearance. The specific proxy is exposed as an equivalent standard UPnP device and when an action is invoked, the proxy transfers the invocation to the correspondent device using its own semantics and syntax. We choose to expose non UPnP plug-n-play devices as UPnP since it is the most mature protocol so far. We tested the approach on an HP 4515x Printer. The solution should work on IGRS devices since it uses the same layers as UPnP and exposes the devices in WSDL. We are working on the enhancement of the alignment and exposing DPWS devices as UPnP Manageable Devices[2] for monitoring operations. We are also working on the verification of the expert code and the evaluation of DOXEN at the operator and home network sites.

References

[1] M. Weiser, “The computer for the 21st century,” *SIGMOBILE Mob. Comput. Commun. Rev.*, 1999.

[2] UPnP, http://www.upnp.org/.

[3] IGRS, http://www.igrs.org/.

[4] OASIS, “Devices profile for web services, 2009,” http://docs.oasis-open.org/ws-dd/ns/dpws/2009/01.

[5] SOA4D, https://forge.soa4d.org/.

[6] T. Spets and A. Fedosseev, “Common application layer,” Broadband Forum, Home Working Group, 2010.

[7] Y. Kalfoglou, “Exploring ontologies,” *Software Engineering and Knowledge Engineering*, 2001.

[8] J. Euzenat and P. Shvaiko, *Ontology Matching*. Springer, 2007.

[9] Y. Bromberg, L. Réveillère, J. Lawall, and G. Muller, “Automatic generation of network protocol gateways,” ser. Middleware ’09.

[10] A. Bottaro and et al, “Home soa facing protocol heterogeneity in pervasive applications,” in *ICPS*, 2008.

[11] M. Paolucci and et al, “Semantic matching of web services capabilities,” in *ISWC*, 2002.

[12] S. Ben Mokhtar, A. Kaul, and N. Georgantas, “Efficient semantic service discovery in pervasive computing environments,” in *Middleware’06*, 2006.

[13] N. Ibrahim, S. Frénot, and F. L. Mouël, “User-excentric service composition in pervasive environments,” in *Proceedings of the 2010 24th IEEE International Conference on Advanced Information Networking and Applications*, ser. AINA ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 682–689. [Online]. Available: http://dx.doi.org/10.1109/AINA.2010.129

[14] N. F. Noy, “Semantic integration: a survey of ontology-based approaches,” *SIGMOD, 2004.*

[15] Domestic-OSGi-Gateway, http://elite.polito.it/dog-tools-72.

[16] H. Chen, T. Finin, and A. Joshi, *The SOUPA Ontology for Pervasive Computing*. Springer, 2005.

[17] T. Coopman, W. Theetaert, and D. Preuveneers, “A user-oriented and context-aware service orchestration framework for dynamic home automation systems,” in *Ambient Intelligence and Future Trends - International Symposium on Ambient Intelligence.*, 2010.

[18] K. Moon, Y. Lee, and C. Lee, “Design of a universal middleware bridge for device interoperability in heterogeneous home network middleware,” in *Transactions on Consumer Electronics*, 2005.

[19] V. Miori, L. Tarrini, M. Manca, and G. Tolomei, “An open standard solution for domotic interoperability,” *Transactions on Consumer Electronics*, 2006.

[20] C. El Kaed, Y. Denneulin, F.-G. Ottogalli, and L. F. M. Mora, “Combining ontology alignment with model driven engineering techniques for home devices interoperablity,” in *Proceedings of the 8th IFIP WG 10.2 international conference on Software technologies for embedded and ubiquitous systems*, ser. SEUS’10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 71–82.

[21] “The owl api,” http://owlapi.sourceforge.net/.

[22] S. Giorgos, “A string metric for ontology alignment,” *International Semantic Web Conference*, 2005.

[23] C. Fellbaum, “Wordnet: An electronic lexical database.” Cambridge, MA: MIT Press, 1998.

[24] “Alignment api,” http://alignapi.gforge.inria.fr.

[25] “Atopai,” http://sites.google.com/site/doxenatopai/.

[26] M. Gonzalez, R.C.Thomas, *Syntatic Pattern Recognition:an Introduction*. Addison Wesley,Reading,MA, 1978.

[27] Microsoft, “Standard dpws printer specifications,” 2007.

[28] Egaa, Stevens, and Antezana, “Transforming the axiomisation of ontologies: The ontology pre-processor language,” in *Proceedigns of OWLED DC OWL*, 2008.

[29] N. Frank and et, *Structural Models: An Introduction to the Theory of Directed Graphs*. John WileySons, 1966.

[30] “Upnp base driver,” http://felix.apache.org/.

[31] “Cybergarage,” http://www.cybergarage.org/.

[32] H. Wache and U. Visser, “Ontology construction - an iterative and dynamic task.” *FLAIRS 2002*.

[33] N. Noy and D. McGuinness, “Ontology development 101: A guide to creating your first ontology, knowledge systems laboratory,” *Stanford University, CA*, 2001.

[34] UPnP, “Standard upnp printer,” October 28 2006.