Using Queries for Semantic-based Service Utilization

Mohammad-Reza Tazari

Fraunhofer-IGD, Darmstadt, Germany
saied.tazari@igd.fraunhofer.de

Abstract. A vision of the Semantic Web is to facilitate global software interoperability. Many approaches and specifications are available that work towards realization of this vision: Service-oriented architectures (SOA) provide a good level of abstraction for interoperability; Web Services provide programmatic interfaces for application to application communication in SOA; there are ontologies that can be used for machine-readable description of service semantics. What is still missing is a standard for constructing semantically formulated service requests that solely rely on shared domain ontologies without depending on programmatic or even semantically described interfaces. Such a standard would facilitate the realization of *Semantic RPC* as the whole process from issuing such a request, matchmaking with semantic profiles of available and accessible services, deriving input parameters for the matched service(s), calling the service(s), getting the results, and mapping back the results onto an appropriate response to the original request. The standard must avoid realization-specific assumptions so that frameworks supporting semantic RPC can be built for bridging the gap between the semantically formulated service requests and matched programmatic interfaces. This paper introduces a candidate solution to this problem by outlining a query language for semantic service utilization based on an extension of the OWL-S ontology for service description.

1 Introduction

A look at the visionary scenario for explaining some of the features of the Semantic Web in [1] quickly reveals the importance of semantic interoperability for the realization of those features. Taking the first paragraph of the scenario as an example where it states, "his phone turned the sound down by sending a message to all the other *local* devices that had a *volume control*", it is obvious that the phone is supposed to only broadcast a message without any dependency on the technical details of addressing and accessing the targeted devices; instead, it relies on shared concepts, here mainly (1) *turn the sound down* if you (2) *are in the vicinity of location l* and (3) *have a volume control*. We call this kind of messaging *semantic RPC*.

Research in the recent years has made significant progress towards realizing semantic RPC. The cornerstone was laid by service-oriented architectures

(SOA) that provide the needed level of abstraction for interoperability. The OASIS technical committee for SOA reference model defines SOA as "a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations."[5] From a technical point of view, the shared knowledge among the components of an SOA-based system is a set of interfaces instead of concrete references to other components and there is some mechanism to publish realizations of those interfaces, on one hand, and to find, bind, and utilize desired realizations of them, on the other hand. Then, the concept *Service* is an abstraction over the actual *value* to be provided by the realizations of the underlying interface (cf. [9] & [10]). "Value" denotes here the actually expected functionality and / or effect[1]. In compliance with this understanding, the most explicit definition of the term *service* in the virtual realm is: "an executable program function"[6]. This paper relies on this latter definition.

OSGi[2] and Web Services[3] provide two possible realizations of SOA concentrating on needed programmatic interfaces, where service registration and search are based on signature declarations and may use keyword- and /or taxonomy-based advertisement and inquiry mechanisms which restrict the process of service discovery to matches with no or little machine-interpretable semantics. OSGi was originally designed to deal with software interoperability within one physical node[4], whereas Web Services are concerned with interoperability across several networked nodes.

Ontological approaches, such as OWL-S[7], aim at providing more inference and reasoning potential based on logical implications of ontological assertions when performing tasks, such as service advertising, constructing service requests, matchmaking, invocation, enactment, composition, monitoring and recovery. In case of OWL-S, this is done by providing a domain-independent OWL-based language for describing services in an unambiguous, computer-interpretable form. A related domain ontology, on which the semantics of what a service does depends, can be imported into the OWL-S–based description of the service; then, the description of the preconditions, inputs, outputs, and results of the service (also a sort of signature specification) can be provided using the concepts from the domain ontology.

The case of semantic RPC involves the tasks advertising, on one side, and constructing service requests, matchmaking, invocation, and returning responses,

[1] In the definition provided by OASIS, this corresponds to the concept of capability. Although OASIS distinguishes between a capability as "some functionality created to address a need" and a service as "the point of access where that capability is brought to bear in the context of SOA", they admit that "in actual use the service may be talked about in terms of being the capability" (all citations from [5]).

[2] Originally an abbreviation for *Open Services Gateway initiative*. See the pages of the OSGi Alliance at www.osgi.org.

[3] See W3C Web Services Activity at www.w3.org/2002/ws/.

[4] Recent specifications also consider the cooperation between several OSGi platforms running on several distributed nodes.

on the other side. Although the OWL-S profile ontology is designated for service advertisement, but there is still no specific solution for constructing service requests, which has lead to the redundant usage of the profile ontology also in forming requests as a sort of "query by example" (cf. [2] & [9]). In reply to the submission of such a request and as a result of the matchmaking task the requester receives a set of matched profiles; then the requester may decide to invoke an appropriate matching service using an associated grounding. As the OWL-S profile ontology is based on the specification of process signatures, consequently the above solutions depend on signatures, as well. Although not based on "query by example", other approaches described in [4], [11], and [13] also base the client requests on a specification of the signatures of sought services.

In this paper, an alternative solution is introduced based on an explicit query mechanism that bundles the process of semantic RPC in one step from the view point of the requester. A brokerage mechanism comprises the matchmaking and invocation tasks altogether (cf. [8]). The broker is supposed to have access to service advertisements in form of OWL-S profiles; by receiving "service requests" in form of queries that have no dependency on any signature, the broker then tries to find the target services and invoke them while deriving the expected input parameters from the data implicitly provided by the query and return related "service responses". Figure 1 illustrates this process assuming a semantic interoperability platform that realizes both the broker and the client-side stub.

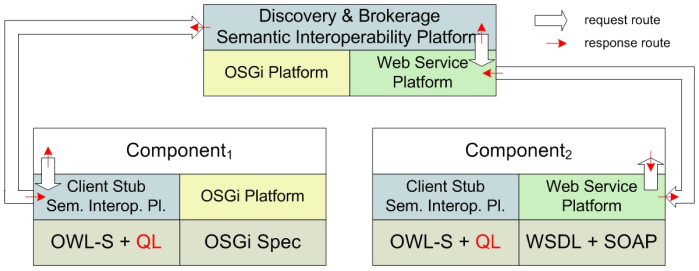

Fig. 1. The role of a semantic interoperability platform (the blue boxes) that uses the specification of a query language for "semantic RPC" and resolves a query made by component$_1$ into calling an appropriate service realized by component$_2$ returning a related result

This solution is based on a slightly new way of using OWL-S and the concept of "property paths". The understanding of OWL-S and the particular way of using it is presented in the next section. Afterwards, we introduce the concept of "property paths" and the first ideas towards the specification of the envisioned query language which is the result of work within the EU-IST project PERSONA[5]. The paper continues by introducing this research context together with concrete work done and concludes with discussing next steps towards a complete specification of an appropriate query language.

[5] www.aal-persona.org

2 OWL-S and Domain Ontologies

As mentioned in the previous section, the OWL-S profile ontology is intended for providing service advertisements. The modeling relies on the concepts from the OWL-S process ontology, which is in turn a possible ServiceModel that can be used to describe how a service works. Figure 2 summarizes the related excerpt from the OWL-S ontology.

The major concepts from the depicted process ontology are[6]:

1. `Process#hasPrecondition` for defining the requirements of the server from its clients (the client may have to fulfill some preconditions in order for the service to have a chance to proceed)
2. `Process#hasInput` for specifying the input to be provided by the clients
3. `Process#hasOutput` for specifying the possible output the service may provide
4. `Process#hasResult` for specifying the set of alternative results of a service, where the concept `process:Result` has been refined with:
   (a) `Result#inCondition` for specifying the conditions in which a result is produced
   (b) `Result#withOutput` for specifying the subset of output parameters that are relevant in a result case (for each result case, some output parameters may have a fixed value that can also be specified)
   (c) `Result#hasEffect` for specifying the set of effects of the process in a result case

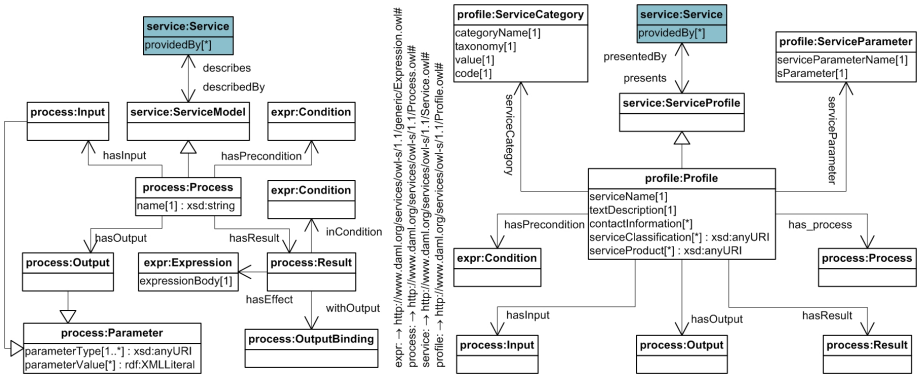

Fig. 2. The OWL-S profile ontology (right) and the related excerpt from the OWL-S process ontology (left)

Using the above service model, the OWL-S profile ontology specifies the following concepts:

– `serviceName` and `textDescription` for simple text-based searches.

[6] Using a Java-Doc like notation for relating classes and properties.

- `contactInformation` for providing descriptions of different entities that play a role in the service provision chain, e.g. the developer, distributer, and/or hosting entities; beside the informative value of this concept, a broker, for instance, could use this info to prioritize the results of the match-making process based on a ranking of its provider, e.g. by associating the service with a trust level in the sense that the default trust level for a service is equal to the trust level of the entity originally providing it. On the other hand, if quality ratings are stored for services, then an average rating can be derived for service providers based on the ratings of the services they provide.
- `serviceCategory` for categorization of services as a means for mapping different domain-specific service ontologies onto each other based on existing taxonomies using the concept of `ServiceCategory` with two concrete subclasses, namely `NAICS` and `UNSPSC` with the taxonomies from `www.naics.com` and `www.unspsc.org` respectively. Additionally, a whole class hierarchy may be defined later based on concrete code-value pairs of those taxonomies, e.g. a class UrbanLighting as subclass of `UNSPSC` with the UN-SPSC code-value pair (93142005, "Urban lighting services"). Then, the development of domain-specific service ontologies could start by choosing the desired concepts from such "global" ontologies and adding the desired specializations.
- `serviceClassification` and `serviceProduct` for the case that the original domain ontology is developed independently from such a global ontology, these two properties may refer to classes from the global ontology in order to keep the possibility of ontology mapping open.
- `serviceParameter` mainly for the provision of non-functional characteristics of concrete realizations of services. Non-functional characteristics are of general nature and may apply to arbitrary services; `MaxResponseTime`, `AverageResponseTime`, and `GeographicRadius` are three example service parameters defined in the ontology `ProfileAdditionalParameters.owl`.
- `has_process` for providing the URI of the process that models the service as the unique ID identifying the selected operation on the server side.
- `hasPrecondition`, `hasInput`, `hasOutput`, and `hasResult` for the same purposes as in case of the OWL-S process ontology.

The above summary shows that OWL-S justifiably concentrates on domain-independent aspects of modeling services. The link to the domain ontology is made by introducing the concept of service category, on one side, and the usage of the domain concepts in the description of the process preconditions, inputs, outputs, and results, on the other side. However, there is no argument against bringing these two aspects together and using the OWL-S concept "service:Service" directly as the root of explicit concepts in a doman ontology foro defining specific classes of services with relations to other domain concepts. Figure 2 shows a simple ontology defined in this way that models lighting services based on an understanding of light sources and the fact that lighting services control light sources.

We believe that this new approach of using the "service:Service" concept of OWL-S as the root of specific service classes and relating them to the domain

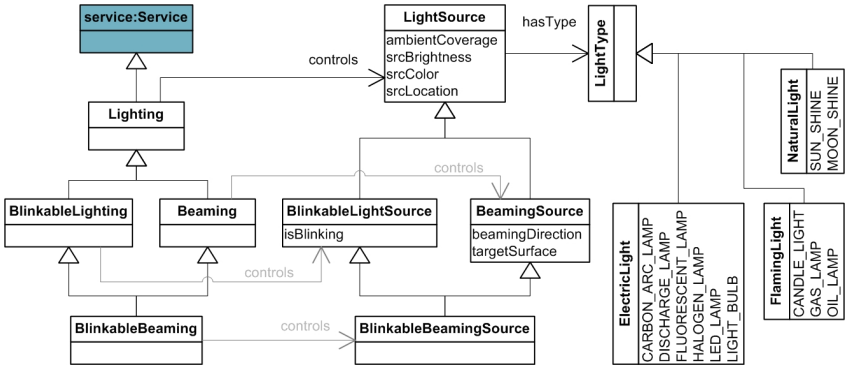

Fig. 3. Example modeling of lighting services

ontology via properties, such as "controls" in the lighting example, opens the door to more expressiveness in the specification of OWL-S processes and profiles. The basic argument is that in this way a certain domain-specific context is provided that can much easier be referred to when describing concrete services. In the next section, we dig deeper into this point and try to provide some evidence based on simple examples.

3 "Property Paths" and an Outline of Service Queries

The idea of "property paths" or "property chains" is being discussed in the Semantic Web community since some time[7]. While some restrict a property chain to contain only instances of *owl:ObjectProperty*, it seems to be possible to omit this restriction on the last property in the chain, so that the last property in the list can be either an *owl:DatatypeProperty* or an *owl:ObjectProperty*. With this freedom, we can define a property path as a closed list of property URIs that relates a resource to a set of other resources or literal values as it could be reached by conventional "join" operations over an RDF database. The following example should help to understand this concept better: Assuming that an RDF database contains the following triples (using the N3 notation):

- r_1 $:p_1$ r_2, r_3 .
- r_2 $:p_2$ r_4 .
- r_3 $:p_2$ r_5, r_6 .
- r_4 $:p_3$ o_1, o_2 .
- r_5 $:p_3$ o_3 .
- r_6 $:p_3$ o_4, o_5 .

[7] For example, see the spec of Notation 3 under www.w3.org/DesignIssues/Notation3 or the wiki pages and the concept *owl:propertyChain* proposed for the next version of OWL specification under www.w3.org/2007/OWL/.

Then, using the path notation from www.w3.org/DesignIssues/Notation3, the relation between r_1 and o_1, o_2, o_3, o_4, o_5 can be summarized as

$$r_1 \ :p_1!:p_2!:p_3 \ \ o_1, o_2, o_3, o_4, o_5 \ .$$

Combining the notion of property paths with the explicit definition of service classes that are related to the domain ontology, we can now introduce the idea of *data view* of classes of services. Figure 2 is already imparting how the *Lighting* class of services views the "world" in terms of data. All data reachable from an instance of this class can be addressed with a unique property path so that at each point in time we can virtually construct a table containing those property paths as column names and rows containing the values associated with the column names at that given time. Then, the semantic of inputs, outputs, and effects of a large set of concrete services can be summarized the following way:

- effects either change certain columns of certain rows, add a new row, or remove an existing row
- inputs either select a certain set of rows by being incorporated in a filtering function, or are used in the expressions defining an effect
- outputs return the values in certain columns after filtering and just before or after the effects has taken place; outputs may undergo some conversion after having been selected, as well

The thesis is now that both components that realize services (and want to advertise their profiles) and clients that want to issue a service request can rely on this shared "data view" of the domain, virtually based on such an imaginary table as mentioned above. Let's look at an example for a better understanding of the above: A software component that controls 4 light bulbs in a given apartment may define a subclass of the above *Lighting* class of services the following way, basically stating that, as a subclass of Lighting services, it controls only four concrete light sources of type LIGHT_BULB whose "ambient coverage"s are not known and whose "brightness"es accept values equal to only 0 or 100:

```
server:MyLighting
  a owl:Class ;
  rdfs:subClassOf l:Lighting ;
  rdfs:subClassOf
    [ a owl:Restriction ;
      owl:onProperty l:controls ;
      owl:allValuesFrom
        [ a owl:Class ;
         owl:oneOf ( server:lb1 server:lb2 server:lb3 server:lb4 ) ] ] ;
  rdfs:subClassOf
    [ a owl:Restriction ;
      owl:onProperty l:controls ;
      owl:allValuesFrom
        [ a owl:Restriction ;
          owl:onProperty l:hasType ;
         owl:hasValue l:LIGHT_BULB ] ] ;
```

```
  rdfs:subClassOf
    [ a owl:Restriction ;
      owl:onProperty l:controls ;
      owl:allValuesFrom
        [ a owl:Restriction ;
          owl:onProperty l:ambientCoverage ;
          owl:cardinality 0 ] ] ;
  rdfs:subClassOf
    [ a owl:Restriction ;
      owl:onProperty l:controls ;
      owl:allValuesFrom
        [ a owl:Restriction ;
          owl:onProperty l:srcBrightness ;
          owl:allValuesFrom
            [ a owl:DataRange ;
              owl:oneOf ( 0 100 ) ] ] ] .
```

The above mentioned virtual table in case of this service class at a time t_0 could look like the following:

Table 1. The state of the virtual table corresponding to the 'server:MyLighting' service class at a certain point in time

l:controls	l:controls ! l:hasType	l:controls ! l:srcBrightness	l:controls ! l:srcColor	l:controls ! l:srcLocation	l:controls ! l:srcLocation ! loc:locName
server:lb1	l:LIGHT_BULB	0	col:white	server:room1	'bathroom'
server:lb2	l:LIGHT_BULB	0	col:white	server:room2	'kitchen'
server:lb3	l:LIGHT_BULB	0	col:yellow	server:room3	'living room'
server:lb4	l:LIGHT_BULB	0	col:yellow	server:room4	'sleeping room'

The imaginary server component can then advertise its OWL-S profiles, below two summarized examples of them focusing on semantics[8]:

```
server:getControlledLightSources
  a server:MyLighting ;       # class-level restrictions: specify the underlying
                              # table, here Table 1
  service:presents
    [ a profile:Profile ;
      profile:has_process
        [ :- server:getCtrlLghtSrcProcess ; a process:Process ] ;
      profile:hasOutput
        [ :- server:controlledLightSources ;
          a process:Output ;
          process:parameterType "...#LightSource"^^xsd:anyURI ] ;
```

[8] In this scheme, 'ext:' denotes a first suggestion for a namespace with possible extensions to OWL-S needed to realize this approach.

```
      profile:hasResult       # all rows in scope because no instance-level
                              # restrictions specified; see next example
        [ a process:Result ;
          process:withOutput # specify columns to be returned
            [ a process:OutputBinding ;
              process:toParam server:controlledLightSources ;
              process:valueForm """
                [ a ext:PropertyPath ;
                  ext:thePath ( l:controls ) ]
              """^^xsd:string ] ] ] .

server:turnOn
  a server:MyLighting ;       # class-level restrictions: specify the underlying
                              # table, here Table 1
  a [ a owl:Restriction ;     # instance-level restrictions: select rows handled by
                              # this service
      owl:onProperty l:controls ;
      owl:hasValue
        [ a process:ValueOf ;
          process:fromProcess process:ThisPerform ;
          process:theVar server:selectedLightSource ] ] ;
  service:presents
    [ a profile:Profile ;
      profile:has_process
        [ :- server:turnOnProcess ; a process:Process ] ;
      profile:hasInput
        [ :- server:selectedLightSource ;
          a process:Input ;
          process:parameterType "...#LightSource"^^xsd:anyURI ] ;
      profile:hasResult       # the scope is assumed to be restricted to rows
                              # selected by instance-level restrictions
        [ a process:Result ;
          process:hasEffect # specify which columns of the selected rows are
                            # affected how
            [ a expr:Expression ;
              expr:expressionLanguage ext:OWL ;
              expr:expressionBody """
                [ a ext:ChangeEffect ;
                  ext:affectedProperty
                    [ a ext:PropertyPath ;
                      ext:thePath ( l:controls l:srcBrightness ) ] ;
                  ext:propertyValue 100 ]
              """^^xsd:string ] ] ] .
```

In case of 'server:getControlledLightSources' above, the presented profile simply states that the service returns the list of controlled light sources. The main point here is the binding of the output parameter using a property path that reflects the semantic of the data to be returned. In case of 'server:turnOn', a suggestion is made for stating the role of the input parameter 'server:selectedLightSource' as a filter that selects the light source to be affected. This is done by introducing

a solution for defining instance level restrictions in the context of a service profile based on the OWL-S concept 'process:ValueOf'. Furthermore, an OWL notation is introduced for stating which property is affected while specifying the new value. The specification of the affected property is done using the concept of property paths.

Similarly, a client component that wants to know the location of a given light source can formulate its request the following way: In the context of the service class `l:Lighting`, return the value in the "column" `l:controls!l:srcLocation` from a "row" whose "column" `l:controls` equals to `other:hallog1`. A request for turning off all light sources in a given location is equivalent to stating that an instance of the service class `l:Lighting` is sought that can set the "column" `l:controls!l:srcBrightness` equal to `0` in all "rows" having the value `other:loc1` in "column" `l:controls!l:srcLocation`.

Alone the two client-side examples in plain text show how a service request can be formulated as a query over the imaginary table, here Table 1. Hence, thinking in terms of SPARQL[9] and ignoring the head part with the definition of used namespaces for the sake of brevity, the envisioned query language can be outlined the following way:

```
CALL           ?s
[WITH OUTPUT   <var> {, <var>}*]      # alternative keyword: SELECT
[HAVING EFFECT <var>=<value> {, <var>=<value>}*]
WHERE          <SPARQL-style spec of the context of ?s & <var>s>
[USING FILTER  <global-selection-criteria>]
```

where

- no explicit notion of property paths is needed because a chain of SPARQL-style conditions eventually specifies a certain property path
- `HAVING EFFECT` is an abbreviation for `HAVING CHANGE EFFECT`; other variants of it are `HAVING ADD EFFECT` and `HAVING REMOVE EFFECT`; only one variant is allowed, however the `ADD` and `REMOVE` variants may appear repeatedly to add or remove several rows
- at least one of `HAVING EFFECT` or `WITH OUTPUT` is mandatory
- each variable in the `WITH OUTPUT` or `HAVING EFFECT` clause must address a certain column of the imaginary table; in case of `WITH OUTPUT`, variables may be associated with a unary aggregating operation, such as *min*, *max*, or *sum*
- the `USING FILTER` clause has a similar role like the `WHERE` clause with the difference that in the latter only property paths from the viewpoint of the referenced service class are supposed to be used, whereas <*global-selection-criteria*> are based on the OWL-S concept of 'profile:ServiceParameter' (see section 2 and service parameters such as *AverageResponseTime*) in combination with unary operations *minOf* and *maxOf*. These filters are supposed to be considered if the brokering mechanism (cf. figure 1) finds more than one matching profile. Specifying no criteria is equivalent to "broadcasting" the

[9] SPARQL Query Language for RDF – www.w3.org/TR/rdf-sparql-query/

request to all components with matching service profiles; specific standard filters, such as *random* or *best match* can be introduced for indicating that only one of the matching services should be selected on a random / best match basis.
– An implicit return value to report the query handling status can be assumed to be returned in a system variable by the broker in any case; possible values could consist of "succeeded", "no matching service found", "response timed out", and "service-specific failure".

Using this querying scheme, the above client side examples, namely querying the location of a given lamp and turning off all light sources in a given location, can be formulated as follows:

```
# get the location of other:hallog1
CALL   ?s
SELECT ?hallog1Loc
WHERE  ?s a l:Lighting
       ?s l:controls other:hallog1
       other:hallog1 l:srcLocation ?hallog1Loc

# turn off all light sources in other:loc1
CALL          ?s
HAVING EFFECT ?b = 0
WHERE         ?s a l:Lighting
              ?s l:controls ?ls
              ?ls l:srcLocation other:loc1
              ?ls l:srcBrightness ?b
```

4 Research Context

The above results are the outcome of work within the EU-IST project PERSONA (PERceptive Spaces prOmoting iNdependent Aging – www.aal-persona.org), which aims at providing an open and scalable technological platform that facilitates the development and deployment of a broad range of services in the field of Ambient-Assisted Living (AAL), in PERSONA with a focus on the "smart environment" aspects. AAL is the concept that groups the set of technological solutions, named AAL Services, targeting the extension of the time that elderly and disabled people live independently in their preferred environment. The PERSONA scenarios address the need for social integration and belonging, independence in daily life activities, security and safety at home and outdoors, and mobility.

Smart environments are treated in PERSONA as open distributed systems. The open distributed platform of PERSONA should make the formation of smart environments affordable for all. It should be possible to start with a small core and let the system evolve over time so that people can arrange the system according to their individual needs as they arise. A compact piece of software should enable the producers of diverse networking-enabled components, such as sensors and controllable gadgets and appliances, as well as developers of applications

with different levels of complexity, to produce "PERSONA-aware" components that can be plugged into the system dynamically without much effort. Instead of dealing with several complicated interfaces of different devices, users should experience an integrated world easy to interact with based on natural communication. Finally, the open and distributed nature of the system should lead to more competition on the market promoting the production of not only new components but also improved alternatives for different parts of the existing system, both at the level of application and the platform itself.

PERSONA abstracts the physical architecture of AAL Spaces (e.g., smart homes of elderly people) as a dynamic ensemble of seamlessly networked nodes. Hence, as a first step the project started to design and develop the smallest piece of software, called the PERSONA middleware, that was needed to make a node "PERSONA-aware" (cf. figure 4). To develop the middleware, a message brokering approach based on ontological match-making has been chosen. The middleware provides four virtual communication buses dedicated to dispatching captured user input (the input bus) as well as generated system output (the output bus) on one side, and enabling both event-based and call-based inter-component communication (through context and service buses respectively), on the other side. For more info on details of this design principle, please refer to [12].

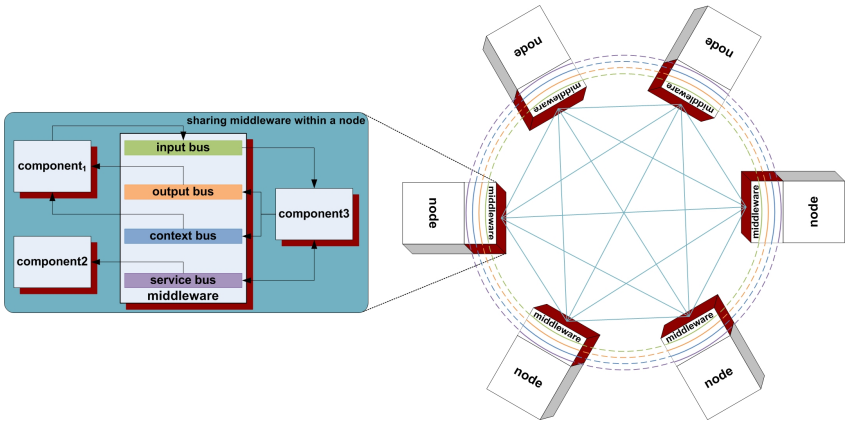

Fig. 4. The PERSONA system as a dynamic ensemble of seamlessly networked nodes

The PERSONA middleware realizes a reasoning mechanism based on the fundamental concepts of the Semantic Web, e.g. RDF resources, OWL classes, individuals, and class expressions, and OWL-S. Two pluggable components that share the same ontology can communicate using the middleware buses while the middleware itself remains neutral in regard to the used ontologies. Data exchanged on the middleware buses is serialized in RDF/XML as soon as it leaves

an instance of the middleware and again de-serialized by the receiving instance of the middleware transparently so that members of the buses see the data always in form of objects without needing to deal with XML (de-)serialization. Each of the four buses of the PERSONA middleware introduces certain RDF resources that can be used for exchanging messages on them (e.g., the RDF resource used for message exchange on the context bus is called a *context event* [3]), and works based on a distributed message brokering strategy that hides the distribution of the components on the nodes in the ensemble leading to a virtually global view on the corresponding bus.

The PERSONA service bus realizes the semantic interoperability platform sketched in figure 1. More specifically, it is responsible for handling service requests. From among different possibilities for realizing the bus strategy, we chose to promote one specific instance of the service bus as the coordinator with a global registry of service profiles. Apart from possibilities for registering service profiles, querying the profiles, and subscribing for the availability of services, the main messages exchanged on the service bus consist of *service request*s, *service call*s, and *service response*s. Currently, service requests cannot be formulated as a query with a syntax similar to the one proposed in section 3, but rather in form of certain data structures that could be the result of parsing such a query. Upon receiving such a service request, the service bus matches it against the registered service profiles using an OWL-based reasoning mechanism developed within PERSONA. As a result of this match-making, concrete service calls are inferred by extracting the OWL-S process URI and deriving the input parameters that are then forwarded to the component(s) realizing the service. The bus then gathers the responses and constructs an aggregated response that is finally returned to the original requester.

So far we could cover all sorts of interoperability needs within the PERSONA system based on the above scheme. All the practical difficulties that have arisen so far could be mastered based on improvement of the domain ontology model, normally by adding missing concepts. Results of evaluating this solution will be available in the spring 2010, after the deployment and tests with real users has taken place in the pilot sites in the fall 2009 / winter 2010.

5 Conclusions

In this paper, we introduced the idea of *semantic RPC* based on a *semantic interoperability platform* that allows software components to interact based on a query language, delegating *1)* the resolution of the query into concrete calls and *2)* constructing the query results to a broker within the platform. The framework aims at omitting all sorts of signature / syntactical dependencies between software components allowing them to interact just based on a shared understanding of the domain at hand, namely the domain ontology. A realization of this framework has been provided in the context of the EU-IST project PERSONA.

The improvements achieved by this solution are *1)* the query makes no assumptions about the set of input parameters possibly needed by the realizations of the requested service, which leads to more freedom in developing software components independently, and *2)* concrete SOA solutions can be bound once within the framework relieving individual components of dealing with heterogeneity of platforms.

The complete specification of the query language itself and the development of an appropriate parser are still open tasks. As the proof of concept is based on a special-purpose reasoner developed within PERSONA, adjustments in the concept, especially on the side of service advertisements, may be needed in order to guarantee its compliance with OWL DL.

References

1. T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001.
2. G. Denker, L. Kagal, T. Finin, M. Paolucci, and K. Sycara. Security for DAML Web Services: Annotation and Matchmaking. In *Proceedings of the 2nd International Semantic Web Conference (ISWC'03)*, volume 2870 of *LNCS*, pages 335–350. Springer, 2003.
3. A. Fides-Valero, M. Freddi, F. Furfari, and M.-R. Tazari. The PERSONA Framework for Supporting Context-Awareness in Open Distributed Systems. In *Ambient Intelligence – Proceedings of the European Conference AmI 2008*, volume 5355 of *LNCS*, pages 91–108, Nuremberg, Germany, November 2008. Springer.
4. M. Klusch and P. Kapahnke. Semantic Web Service Selection with SAWSDL-MX. In R. L. Hernandez, T. D. Noia, and I. Toma, editors, *SMRR*, volume 416 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.
5. C. MacKenzie, K. Laskey, F. McCabe, P. Brown, R. Metz, and B. Hamilton. Reference Model for Service Oriented Architecture 1.0. SOA-RM Technical Committee Specification, Organization for the Advancement of Structured Information Standards – OASIS, `http://www.oasis-open.org/committees/download.php/19679/`, August 2006.
6. D. Martin, J. Domingue, M. L. Brodie, and F. Leymann. Semantic Web Services, Part 1. *IEEE Intelligent Systems*, 22(5):12–17, 2007.
7. D. Martin, M. Paolucci, S. McIlraith, M. Burstein, D. McDermott, D. McGuinness, B. Parsia, T. Payne, M. Sabou, M. Solanki, N. Srinivasan, and K. Sycara. Bringing Semantics to Web Services: The OWL-S Approach. In *Proceedings of the 1st International Workshop on Semantic Web Services and Web Process Composition (SWSWPC'04)*, volume 3387 of *LNCS*, pages 26–42. Springer, 2004.
8. M. Paolucci, J. Soudry, N. Srinivasan, and K. Sycara. A broker for owl-s web services. In L. Cavedon, Z. Maamar, D. Martin, and B. Benatallah, editors, *Extending Web Services Technologies: the use of Multi-Agent Approaches (Multiagent Systems, Artificial Societies, and Simulated Organizations)*. Springer, March 2005.
9. C. Preist. A Conceptual Architecture for Semantic Web Services. In *Proceedings of the 3rd International Semantic Web Conference (ISWC'04)*, volume 3298 of *LNCS*, pages 395–409. Springer, 2004.
10. D. Roman, U. Keller, H. Lausen, R. L. Jos de Bruijn, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel. Web Service Modeling Ontology. *Applied Ontology*, 1(1):77–106, 2005.

11. N. Srinivasan, M. Paolucci, and K. Sycara. Adding OWL-S to UDDI, Implementation and Throughput. First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004), February 2009.
12. M.-R. Tazari, F. Furfari, J.-P. L. Ramos, and E. Ferro. The PERSONA Service Platform for AAL Spaces. In H. Nakashima, H. Aghajan, and J.-C. Augusto, editors, *Handbook of Ambient Intelligence and Smart Environments (AISE)*. Springer – To be published in December 2009, 2009.
13. G. A. Vouros, F. Dimitrokallis, and K. Kotis. Look Ma, No Hands: Supporting the Semantic Discovery of Services without Ontologies. In R. L. Hernandez, T. D. Noia, and I. Toma, editors, *SMRR*, volume 416 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.