Univ Access Inf Soc (2016) 15:85–99
DOI 10.1007/s10209-014-0376-x

LONG PAPER

Seamless semantic enrichment of services in assistive environments

Daniel Zmuda · Jacek Psiuk · Marek Psiuk

Published online: 2 September 2014

Abstract The paper proposes a methodology and a tool-based support for the development of semantic services in ambient-assisted living (AAL)-oriented assistive environments. A review of existing approaches in this area is conducted. The review covers a variety of AAL platforms from which universAAL has been chosen for the experimental implementation. The paper presented the iterative development methodology of service semantics, which facilitates the efficient creation of error-free services in AAL platforms. The features needed for the realization of the methodology are implemented as a universAAL platform extension named the annotation-based semantic enrichment. The proposed approach is assessed in the context of a sample scenario in which the services promoting universal access for elderly people or otherwise impaired persons are developed. The assessment results are used to highlight the added value of the presented work and to identify potential areas of future improvement.

Keywords Ambient-assisted living · AAL platforms · Seamless environment enrichment · Semantic annotations · Development process simplification

D. Zmuda (✉) · J. Psiuk · M. Psiuk
Department of Computer Science, Faculty of Computer Science, Electronics and Telecommunications, AGH University of Science and Technology, al. A. Mickiewicza 30, 30-059 Kraków, Poland
e-mail: zmuda@agh.edu.pl

J. Psiuk
e-mail: jacek.psiuk@agh.edu.pl

M. Psiuk
e-mail: marek@psiuk.pl

1 Introduction

An important group of people on which recent research on universal access is focusing is the elderly [2, 14]. This is caused by the demographic changes—particularly in Europe—which result in the increasing population of the elderly people in society [7]. The concept of universal access is addressed, among others, by assistive environments which—in addition to enhancing accessibility—increase the general quality of life. A research domain on assistive environments directed at elderly people is referred to as ambient-assisted living (AAL). In response to the demographic changes, AAL leverages the potential of Information and Communication Technologies (ICT) to extend the period during which elderly people and people with disabilities can enjoy a healthy, safe and happy life on their own [1, 31].

There were many research initiatives focusing on the exploitation of ICT in applications from the AAL domain [6, 18, 19, 21–23, 26, 30, 34]. Unfortunately, developing AAL applications without a common framework has proven itself to be ineffective and cumbersome [32]. Therefore, the aforementioned initiatives tried to standardize the development process by introducing the concept of a platform [3, 12]. The exact definition of the AAL platform varies, but the common denominator is that it typically provides an environment for constructing applications from a set of reusable business services, which can be easily composed and supported by the system services provided by the platform itself.

An important feature that should be provided by the AAL platform is a mechanism for expressing and handling service semantics. The semantic information is crucial for the effective composition of the services and choosing the service instance, which is the most appropriate for the current situation [11]. This is especially important in the context of universal access. For example, when a visually impaired person approaches a tourist information terminal, the terminal should detect the disability and automatically select the voice modality channel as the service providing the most appropriate user interface from the semantic point of view.

The existing AAL platforms provide support for creating and managing various aspects of service semantics. However, the process that a developer has to follow is not straightforward. It involves many steps and the resulting implementation cannot be fully tested until the process is finished. This significantly decreases the potential of semantics in AAL applications and the effective value that an AAL platform can bring to the domain of accessibility support. In order to address this issue, the following contribution is provided:

- The Iterative Development Methodology of Service Semantics which simplifies the development process and makes it less error-prone. The Methodology can be applied to any AAL platform;
- The extension of the universAAL platform, named annotation-based semantic enrichment (AAPI) that allows for realization of the Iterative Methodology;
- Critical evaluation of the Methodology and its implementation (AAPI) in a meaningful scenario from the universal access domain.

Throughout the paper, the process of adding semantic meta-data to the services is often referred to as development or implementation of the semantics. This allows the text to be more concise without losing its full meaning.

The structure of the paper is as follows. Section 2 presents a review of research in the field of AAL platforms and highlights the aspects related to semantics. The Iterative Development Methodology of Service Semantics is described in Sect. 3. The description is divided into motivation— Sect. 3.1 and the actual Methodology description— Sect. 3.2. Section 4 presents the Annotation-based Semantic Enrichment (AAPI), which allows for realization of the Iterative Methodology in practice. The evaluation of the proposed solutions in a real-life scenario is presented in Sect. 5. Finally, Sect. 6 concludes the presented research and discusses the improvements which will be addressed in the context of future work.

2 Related work

Many research initiatives attempt to address issues related to universal access with the use of semantics. Some of them, such as [9, 17], describe solutions that provide universal accessibility and interoperability in the context of user interaction allowing for adaptation of the application interfaces to specific user needs. Others [8, 25] leverage the concept of the Semantic Web to propose an approach for profile-dependent accessibility of data for end users.

Since the paper's contribution is focused on the AAL domain, the following section analyzes available AAL platforms and their features related to development of semantic aspects of services.

SOPRANO is an AAL platform oriented toward intelligent context adaptation by gluing high-level abstraction layers, such as planning and context, with lower layers, e.g., service and hardware, by means of a common SOPRANO ontology [11]. The process of gathering events, reasoning about certain situation and, finally, invoking appropriate actions is decomposed into the following architectural components [33]: (1) *Procedural Manager*—maintains high-level actions— procedures—and allows for triggering their invocation by *Composer* and *Context Manager*; (2) *Composer*— applies *Match-making* algorithms to choose service instances fitting an abstract semantic description; (3) *Context Manager*—receives events from sensors and stores them; it also enables subscribing to an event pattern.

The development process in the SOPRANO platform involves several roles, two of which are specifically related to semantics: *Device technology provider* and *Software developer*. The *Device technology provider* creates and maintains implementations of sensors and actuators, which are exposed as services inside OSGi bundles [29]. The *Provider* has to ensure that the services are semantically enriched with information from the SOPRANO ontology. This allows the *Software developer* to use sensors and actuators to combine, augment and aggregate context information into a rich and reliable semantic knowledge base which is then exposed to higher layers. Tasks related to managing the implementation of semantic aspects performed by the *Device technology provider* and the *Software developer* are demanding, yet the platform does not explicitly provide any mechanisms to reduce their complexity.

One of the main objectives of the OASIS Project [10] was to provide an implementation of an Ontology-driven Open Reference Architecture that supports interoperability, connectivity and context sharing between services relevant to the domain of the elderly. The OASIS platform comprises several building blocks directly related to semantics: (1) *Common Ontological Framework (COF)*—stores the specification of relations between different ontology modules in OOR (OASIS Ontology Repository) and allows users to define a *Hyper-Ontology*, which optimizes the integration of different ontologies; (2) *AMI Framework*—provides seamless interactivity between services, applications and ontologies stored by *COF*; (3) *Content Anchoring* *and Alignment Tool (CAAT)*—aligns the functionality of specific services with ontologies stored in the repository; (4) *Content Connector Module (CCM)*—supports automatic integration of newly created services with incoming service requests using the *AMI Framework*.

The process of developing solutions in the OASIS platform assumes the integration of services delivered by different providers in accordance with requirements specified by end users. Different functional parts of services can be composed into applications by exporting their functionality in the form of web-services that then are correlated by the AMI Framework with hyper-ontology to match the user expectations. The exposition and matchmaking of service functionality with user requirements is done automatically in a seamless way by the *CCM*. However, the overall process of services or device descriptions mapping to ontologies assumes manual approach and is time-consuming.

The approach assumed in MonAMI [15]—another OSGi-based AAL platform—focuses on simplifying the business environment for developing AAL services and, as a consequence, fostering the creation of relevant ecosystems [13]. MonAMI achieves this goal by proposing an interoperability framework—OSGi4AMI where information and context are mediated in a seamless and transparent way. OSGi4AMI defines a comprehensive ontology, covering various devices and service types which are mapped to specific Java interfaces. The ontology brings clear separation between the application logic, device logic and other system components. This is a very valuable feature that greatly simplifies the work of developers during initial implementation of services and devices, as well as during system evolution, where a given service/device implementation may need to be exchanged. Unfortunately, the granularity offered by the OSGi4AMI ontology is very coarse—only interfaces and not the specific features are defined. Therefore, managing the ontology in a large-scale system covering many networks, nodes and diverse services becomes difficult.

PERSONA project introduces a framework for supporting context awareness. This framework is implemented as an open middleware-based distributed system, based upon the OSGi technology. It offers semantic RPC and *Match-making* features [4] in the form of data buses (e.g., context bus, service bus). PERSONA enables the service provider to define ontological descriptions of services (*ServiceProfiles*) , which are then used by developers to develop business logic for specific profiles (*ServiceCallees* components). The client may use semantic information to describe the requested services and pass requests to the bus. Internal framework mechanisms perform semantic *Match-making* [27] between requests and the registered *Service-Profiles*, select an appropriate *ServiceProfile*, and communicate the response to the caller. The proposed approach is very mature and offers advanced features for describing services in a semantic manner. However, the inherent complexity of developing ontologies and mapping them to specific implementations makes it error-prone and difficult to use. Moreover, support for evaluating the validity of request/service matches remains rudimentary.

The universAAL project [5] aims at providing an open platform and reference specification of AAL on the basis of several projects (including SOPRANO, Oasis and PERSONA). One of the key goals of the project is to enable developers to easily create applications and reuse existing platform services that are shared within the developer community. To achieve this, universAAL heavily relies on ontologies and semantic descriptions of services. The process of applying semantic descriptions is simplified by providing several tools capable of transforming ontologies between different representations, such as OWL-s, UML, and Java. The platform provides mechanisms for semantic interoperability and Match-making, very similar to the ones introduces in the Persona project. However, the complexity of mapping semantic descriptions to specific business logic components remains high and requires broad knowledge about the ontologies themselves as well as the way in which they apply to a particular implementation.

The presented study of ongoing work related to assistive environments indicates that there are many existing platforms and solutions which address enrichment of services with semantic meta-data. However, in all cases the inherent development process is either too cumbersome and error-prone (e.g., Oasis, Persona) or too much simplified for supporting real-life cases (e.g., MonAMI). This problem is addressed by a methodology proposed in the following section.

3 Iterative development methodology of service semantics

This section firstly presents the paper's contribution by describing a regular process of implementing semantics in AAL platform services. All problems and inconveniences of the process are described from the developer's point of view. Subsequently, the section presents the Iterative Development Methodology of Service Semantics, which aims at automating some parts of the process and ensures constant control over the implementation's validity.

3.1 Motivation

The platforms discussed in Sect. 2 handle service semantics in various ways; however, on the abstract level, all these approaches have several essential aspects in common.

Such aspects are grasped in Fig. 1. Semantics are invariably based on an ontology. The ontology defines the *Semantic Concepts* which, in the AAL Domain, most commonly represent some fragments of the physical world [16]. The *Semantic Concepts* are used to describe a *Service* from both the provider's and the consumer's point of view. The provider defines a *Service Description* that can be perceived as a contract, whose fulfillment is guaranteed. For instance, the contract may state that the invocation of the service will change the state of a given instance of an element modeled by the ontology. In turn, the consumer defines a *Service Expectation*, which represents a contract the consumer would like to see fulfilled. For instance, by invoking the service, the Consumer may expect to read the current room temperature from an appropriate sensor covered in the ontology. Both elements— the *Service Description* and the *Service Expectation*— constitute input data upon which the *Match-making* algorithm operates. The goal of the algorithm is to process *Service Descriptions* available in the current context (e.g., on a platform node or network segment) and choose the service (or multiple services, if permitted) which represents the best semantic match for the given *Service Expectation*. Of course, a situation may arise in which no *Service Descriptions* match a given *Expectation*—in such cases the expectation simply cannot be fulfilled by the platform.

Figure 1 illustrates the elements of service semantics and their interrelations and the responsibilities for developing them. The developer at a provider side is responsible for the creation of the *Provider Implementation*. Such *Implementation* has to cover the service business logic and its *Semantic Description*. The *Service Implementation* has

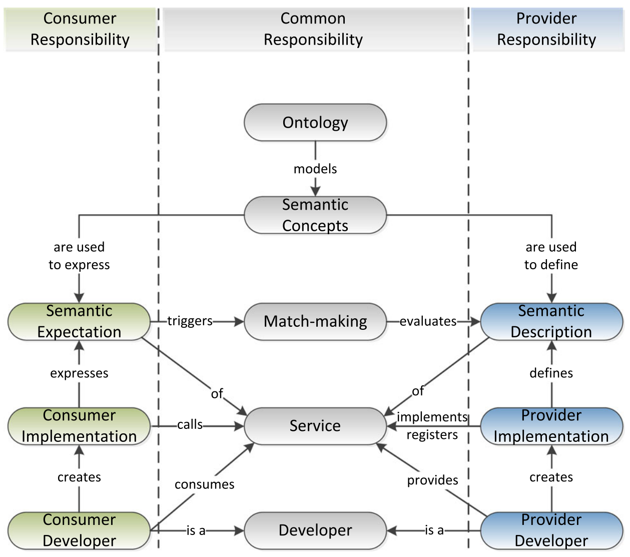

Fig. 1 Concept map presenting an abstract view of semantic service aspects

to be registered by the provider in the platform to expose the *Service Description* for the purposes of *Match-making*, and to allow for realization of service invocations. The consumer developer is responsible for implementing the relevant business logic at the consumer side and for adding there a suitable invocation taking into account the desired *Service Expectation*. The development of the ontology and the related *Semantic Concepts* is a shared responsibility. While most of the time, it would be handled by the provider, this is not mandatory.

Analysis of the development responsibilities leads to the identification of the following steps that need to be covered by different developers:

1. Implementing business logic on the provider's and consumer's side;
2. Modeling the ontology;
3. Creating *Service Description*;
4. Implementing service at provider's side;
5. Formulating *Service Expectation*;
6. Implementing service invocation at consumer's side.

Two essential problems can be identified in the context of the development steps presented. The first identified problem is that the steps may prove complex and time-consuming, and there is no clear distinction between the business logic and the implementation of semantics. Often those two elements are tightly coupled with each other which makes either of them hard to reuse for other purposes. The second problem is that only after all the process steps are finished, the implemented code can be deployed to the platform and launched for the purpose of testing its correctness. If something goes wrong, it is very hard to determine who is responsible for the error—the consumer, the provider or perhaps the creator of the ontology, who may have committed a mistake during the modeling phase.

3.2 Methodology

In order to solve the problems identified in the previous section, the *Iterative Development Methodology* is proposed. The *Methodology* is presented in Fig. 2. It divides the development process into four iterations: A–D. Following each iteration, a certain status is guaranteed, as presented on the right-hand side of the figure. The *Methodology* assumes that the application is developed with the use of the *Semantic Framework*, which itself is an integral part of the AAL platform. In order to make each iteration as developer-friendly as possible, the *Semantic Framework* should provide a set of expected features. These features are listed on the left-hand side of Fig. 2 and are mapped to iterations in which they are needed. Owing to these features, the steps comprising each iteration result in attaining the desired status.

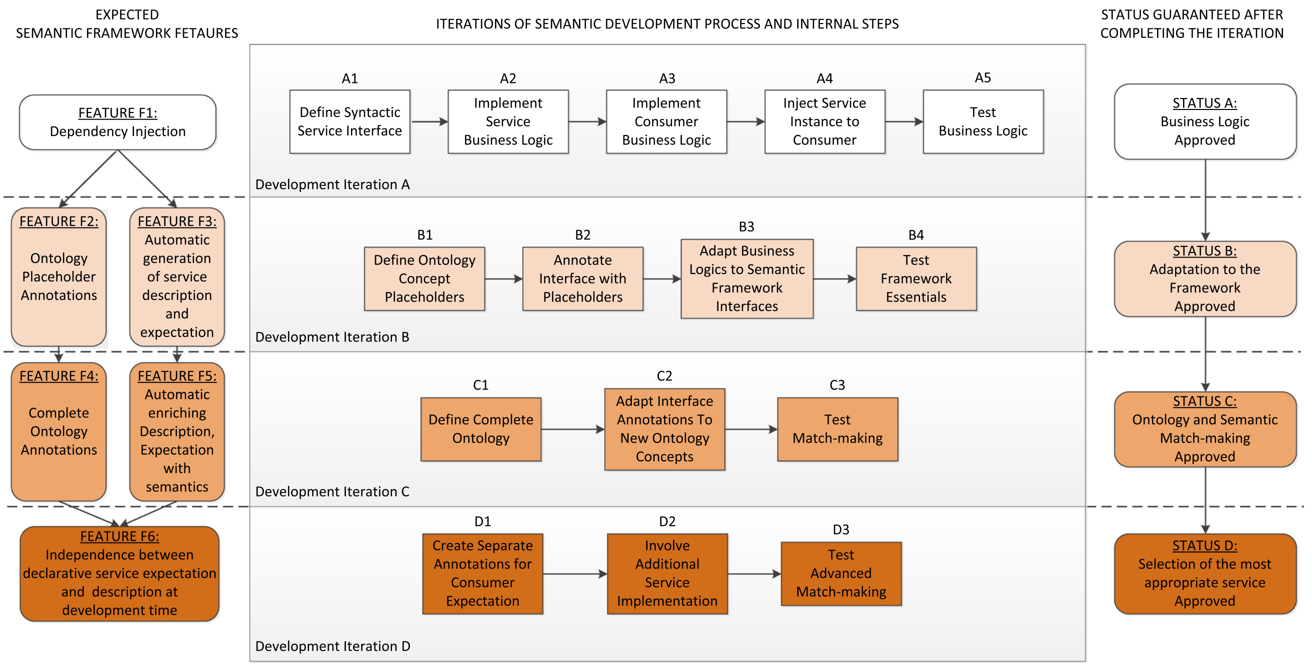
EXPECTED SEMANTIC FRAMEWORK FETAURES
ITERATIONS OF SEMANTIC DEVELOPMENT PROCESS AND INTERNAL STEPS
STATUS GUARANTEED AFTER COMPLETING THE ITERATION
FEATURE F1: Dependency Injection
A1 Define Syntactic Service Interface
A2 Implement Service Business Logic
A3 Implement Consumer Business Logic
A4 Inject Service Instance to Consumer
A5 Test Business Logic
Development Iteration A
STATUS A: Business Logic Approved
FEATURE F2: Ontology Placeholder Annotations
FEATURE F3: Automatic generation of service description and expectation
B1 Define Ontology Concept Placeholders
B2 Annotate Interface with Placeholders
B3 Adapt Business Logics to Semantic Framework Interfaces
B4 Test Framework Essentials
Development Iteration B
STATUS B: Adaptation to the Framework Approved
FEATURE F4: Complete Ontology Annotations
FEATURE F5: Automatic enriching Description, Expectation with semantics
C1 Define Complete Ontology
C2 Adapt Interface Annotations To New Ontology Concepts
C3 Test Match-making
Development Iteration C
STATUS C: Ontology and Semantic Match-making Approved
FEATURE F6: Independence between declarative service expectation and description at development time
D1 Create Separate Annotations for Consumer Expectation
D2 Involve Additional Service Implementation
D3 Test Advanced Match-making
Development Iteration D
STATUS D: Selection of the most appropriate service Approved

Fig. 2 Iterative development methodology of service semantics in the context of features expected from the semantic framework provided by an AAL Platform

The intensity of color tags in Fig. 2 reflects the amount of semantics present in each iteration. As can be seen, the first iteration does not involve semantics at all—it is white. Each subsequent iteration is tagged with a deeper shade of orange, representing a gradual enrichment of the initial iteration with semantics. Please note that following each iteration the implementation is runnable and therefore testable, which allows for the verification of additional semantic elements added in each iteration.

Iteration A, as mentioned, does not include any semantics. It comprises of five steps, A1–A5, which mostly focus on the business logic. (**A1**) The service interface is defined in terms of syntactic elements such as input and output parameters. (**A2**) The business logic of the *Service* on the provider's side is implemented in accordance with the interface defined in the previous step. (**A3**) The logic of *Consumer* handling the Service invocation is implemented. The implementation uses the dependency injection feature (**F1**) [24] to avoid binding to the provider's code— only the interface from A1 is used. (**A4**) For testing purposes, the service instance is injected into the consumer code. It is possible thanks to the dependency injection feature. (**A5**) The implementation of business logic is launched and tested. In case of errors, the logic is fixed and tests are repeated. Successful tests mean that the iteration may conclude, guaranteeing the correctness of business logic (**Status A**).

Iteration B adds minimal semantic support to the result of Iteration A in order to enable checking whether the *Semantic Framework* is used in the appropriate way. This iteration assumes that the *Framework* provides two features: F2 and F3 which are essential for the presented *Methodology*. Feature (**F2**)—*Ontology Placeholder Annotations* provides some means for annotating the interface defined in step A1. A set of possible annotations is defined to express semantic meta-data. Attaching a given annotation to some syntactic element simply gives it a certain semantic meaning. This approach is very straightforward and retains clear separation between business logic and representations of both *Semantic Expectation* and *Semantic Description*. Annotations are not coupled with any particular ontology. They simply represent the semantic expression of the given *Framework*. However, since a semantic directive often needs to relate to some ontological *Concept* (e.g., to represent its meaning), it is assumed that *Concepts* can be referenced by adding arguments to *Annotations*. The *Ontology Placeholders* could be perceived as an ontology skeleton. They define places in which ontology *Concepts* will be inserted in subsequent iterations. The assumption is that the *Placeholders* should allow for executing a simplified *Match-making* algorithm. If the *Description* and *Expectation* refer to the same *Placeholder* which is not yet filled by the ontology, the algorithm should assume that they match. Feature (**F3**) ensures that the *Semantic Framework* is able to automatically generate the *Service Description* and *Expectation* from the annotated interface. It is expected that these two elements are generated at run-time and therefore do not have to be handled at all by the developer.

There are four steps covered by Iteration B: B1–B4. (**B1**) The developer defines *Ontology Placeholders* with the use of feature F2. (**B2**) The developer annotates the interface defined in step A1 and inserts *Placeholders* as annotation arguments. (**B3**) Service exposition, on the provider's end and service invocation on the consumer's end is adapted to the API of the Semantic Framework. This, of course, implies removing direct injection of the *Service* instance from the consumer code (added in step A4). (**B4**) Both implementations (provider's and consumer's) are deployed in the platform and the service invocation is tested. If *Match-making* and the entire invocation cycle succeed, then Iteration B is complete, ensuring the validity of adapting business logic to the *Semantic Framework* API (**Status B**).

Iteration C focuses on modeling ontology for the purpose of enabling the full potential of semantics. Here, it is expected that the *Semantic Framework* will provide Annotations which allow for referring to a complete ontology model (**F4**) and that the *Framework* will be able to generate semantic-rich *Description* and *Expectation* out of these *Annotations* (**F5**). The iteration includes only three steps: C1–C3. (**C1**) The developer (which may be either the provider or the consumer) defines the ontology, including all of its *Semantic Concepts*. (**C2**) The provider improves the *Annotations* created in step **B2** by extending the *Placeholders* from step B1 and referring to the *Concepts* from the ontology. (**C3**) The following elements are deployed to the platform: the provider's code, the consumer's code and the ontology. This enables service invocation to be tested. If the invocation works, the semantically complete *Match-making* process is deemed successful, which also means that the ontology and the semantic *Annotations* of the interface are valid (**Status C**).

Iteration D focuses on more advanced *Match-making* which is not restricted to a single *Expectation* and a single *Description*. Thus far, both the *Expectation* and *Description* have been generated from the same development artifact—the interface, initially annotated with *Placeholders* and later on with ontology *Concepts*. In such circumstances, it is relatively easy to ensure that the generated elements are matched by the *Match-making* algorithm. However, in real-life scenarios, the *Description* and *Expectation* may be created independently. Therefore, Iteration D expects that the *Framework* can handle such independence at the development time (**F6**). The iteration includes the following steps (named D1–D3). (**D1**) The developer at consumer side starts with a clean syntactic interface from step A1 and annotates it with either ontology *Concepts* or *Placeholders* to formulate an independent *Service Expectation*. (**D2**) The developer creates or (if possible) uses existing service implementations that have different *Semantic Descriptions*. (**D3**) The consumer code (including the independent Expectation), all service implementations and the ontology itself are deployed to the platform. The *Service* is then invoked under various circumstances involving both the consumer and the provider. Each invocation is followed by a check whether the *Match-making* algorithm has selected the most appropriate service instance. If not, the *Service Expectation* is reworked and the test is repeated until a positive result is achieved. This concludes Iteration D (**Status D**).

In the presented description of *Methodology*, the consumer and provider code was developed from the beginning in a synchronized manner. As mentioned in Iteration D, this is not always the case. The *Methodology* takes this into account and allows iterations to be performed independently. For example, the provider may have developed the service until Iteration C before the consumer starts work on the client. Even in such cases the completion of each iteration status looks the same. In Iteration A, the service implementation is injected and only the business logic is tested. In Iteration B, only *Placeholders* are used for generating the Expectation. Finally, Iteration C provides full ontology support for the consumer's code. As presented in this short example, the potential of the *Methodology* is preserved even if the consumer and provider perform their work independently.

4 Annotation-based semantic enrichment

In Sect. 2, a review of existing AAL platforms was presented. Each platform has its own advantages as well as drawbacks; however, one solution stands out from the rest. The universAAL platform is the result of consolidation of a number of other projects (among others SOPRANO, OASIS, and PERSONA). In the scope of the universAAL project, a specification of the AAL Reference Architecture is provided, along with a fully functional distributed platform enabling seamless interoperability. Taking this into consideration, universAAL has been selected as a basis for implementation of a Semantic Framework extension that supports the *Methodology* proposed in Sect. 3.2. This section describes the relationship between universAAL components and abstract semantic aspects presented in Sect. 3.1, and introduces the universAAL platform extension named Annotation-based Semantic Enrichment (AAPI).

4.1 UniversAAL semantic framework

Figure 3 presents the mapping of universAAL components to the *Semantic concepts* defined in Fig. 1. *Resources* represent the *Semantic Concepts* used by *Providers* and *Consumers*. The *Semantic Description* is represented by a

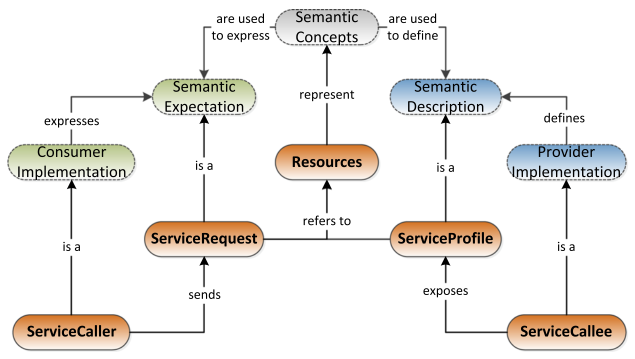

Fig. 3 Mapping of universAAL platform components to abstract semantic aspects of services

collection of *Service Profiles* and their parameters. The *Service Callee* represents the *Provider Implementation* and exposes its capabilities in the form of *Service Profiles*. The *Semantic Expectation* is represented by a *ServiceRequest* and its parameters. The *Service Caller* is acting as a *Consumer Implementation* and uses the *Service Requests* to invoke services.

From the service provider's perspective, the process of providing an implementation compliant with the concepts introduced in Fig. 1 involves preparing a *Semantic Description* of a service and its *Provider Implementation* which maps to *ServiceProfiles* and the *Service Callee*.

Listing 1 presents a sample ontological *Resource* named *LightSource* which represents a source of light. In universAAL, each *Resource* has its own specific properties and unique URIs, which distinguish it from other *Resources*. In this particular case, two properties (color and brightness) are provided and can be managed by some other entities.

The *Service Profile* describes the functional capabilities of a particular service (e.g., turning on the lights, closing the doors etc.) and its relation to specific ontological resources. Listing 2 presents a block of code that creates a profile for turning off the light sources. Line 1 introduces a class which is also described by *Semantic concepts* and represents the *Lighting* resource (an ontological service which handles light sources). Line 2 defines the URI which will be used to obtain the resource passed as input. In line 4, an object representation of our profile is created. Line 5 says that this service operation accepts an input parameter of type *LightSource*, with multiplicity equal to 1, and is accessible under a specified URI. Line 6 adds the description of the effect which will take place following the invocation of this operation. It says that the brightness of the light source will be changed to 0 which means that it will be turned off.

In the universAAL platform, all communication between service providers and consumers depends on the buses. The so-called buses are message-based components that enable message exchange as well as exposure of services to consumers. Exposing a service requires registering *Service Callee* on the bus along with previously created *Service Profiles*. This operation assures that the implemented business logic will be invoked when consumer requests match the registered *Service Profiles*.

At this point, the analysis of implementation is completed, being the provider's responsibility (cf. Fig. 1). The following steps are in the responsibility of the service consumer.

The process of enabling service semantics from the consumer's perspective is similar to the provider's task. The consumer uses the previously designed semantic concepts to perform two actions: express the *Semantic Expectation* of the required services and develop the *Consumer Implementation* which directly performs service invocations.

As mentioned before, in the universAAL platform, the *Semantic Expectation* is represented by a *Service Request*. Listing 3 presents a block of code used for this purpose. In line 3, an initial *ServiceRequest* is created specifying that it will involve services capable of controlling light sources (cf. line 1 of Listing 2). Lines 4–5 say that this request should be handled only by services that have control over the lamp at the given URI and provide the ability to change its brightness property to 0.

Having defined the *Semantic Expectations* of the consumer, it is possible to create the *Consumer Implementation*. The *Service Caller* component publishes the *Service Request* to the bus where *Match-making* occurs. In this process, the semantic information of requests is matched

Listing 1 Java representation of LightSource ontological resource

```java
public class LightSource extends Device {
    public static final String MY_URI = LightingOntology.NAMESPACE + "LightSource";
    public static final String PROP_AMBIENT_COVERAGE = LightingOntology.NAMESPACE + "ambientCoverage";
    public static final String PROP_HAS_TYPE = LightingOntology.NAMESPACE + "hasType";
    public static final String PROP_SOURCE_BRIGHTNESS = LightingOntology.NAMESPACE + "srcBrightness";
    public static final String PROP_SOURCE_COLOR = LightingOntology.NAMESPACE + "srcColor";
    ...
```

Listing 2 Code which performs service profile creation

```
public class ProvidedLightingService extends Lighting {
    static final String INPUT_LAMP_URI = LIGHTING_SERVER_NAMESPACE + "
          lampURI";
    ...
    ProvidedLightingService turnOff = new ProvidedLightingService(
          SERVICE_TURN_OFF);
    turnOff.addFilteringInput(INPUT_LAMP_URI, LightSource.MY_URI, 1, 1,
            ppControls);
    turnOff.myProfile.addChangeEffect(ppBrightness, new Integer(0));
    ...
```

Listing 3 Code creating the service request

```
public class LightingConsumer{
    ...
    ServiceRequest turnOff = new ServiceRequest(new Lighting(), null);
    turnOff.addValueFilter(new String[] { Lighting.PROP_CONTROLS }, new
            LightSource(lampURI));
    turnOff.addChangeEffect(new String[] { Lighting.PROP_CONTROLS,
          LightSource.PROP_SOURCE_BRIGHTNESS }, new Integer(0));
    ...
```

with *Service Profiles* of registered services. If any *Service Profile* matches the *Service Request*, then this request is propagated to the *Service Callee* which registered the profile. If there is more than one match, then all matched callees receive the request. Having processed the request, the *Service Callee* constructs a response (*Service Response*) containing the invocation execution status and any additional data, in accordance with the implemented business logic. Each of these responses traverses through the bus back to the consumer.

Both the consumer and provider actions according to Fig. 1 have been presented in the context of the universAAL platform. The described process involves only the semantic description of a single operation, yet it still remains complicated and error-prone. The following section describes an approach that produces the same results though in a much simpler way, exploiting the *Methodology* presented in Sect. 3.

4.2 Realization of AAPI

This section presents the contribution of the paper in the area of extending the universAAL Semantic Framework which resulted in a tool called Annotation-based Semantic Enrichment (AAPI).

As presented in Sect. 3, the Iterative Development Methodology of Service Semantics defines four iterations. Iteration A does not include any semantics therefore will not be discussed further. In Iteration B, it is assumed that AAPI provides features of the Ontology Placeholder Annotations (F2) and Automatic generation of semantic description and expectation (F3). Because the universAAL platform is written completely in Java, the proposed contribution takes into account only this programming language. Therefore, enrichment of services with semantic meta-data is realized with the use of Java annotations.

As can be seen in Listing 4, two annotations are directly related to specifying service metadata. The *@UniversAALService* annotation is used as a placeholder for service-specific properties such as its *namespace* and *name*. Depending on the usage context (provider or consumer) of the service interface, these are used either for registration or lookup actions in the bus. It is also assumed that this annotation serves as an indicator for AAPI to begin the process of semantic enrichment for a particular service. If it is not present, the service will not be enriched with semantics. The *@OntologyClasses* annotation is used to define domain ontology resources which will be used by the service business logic and therefore need to be dynamically registered by AAPI.

Listing 5 presents the annotations which can be used for each method of the business interface. Similiarly to the *@UniversAALService*, the *@ServiceOperation* indicates that this method should be enriched with semantics. The only parameter of this annotation is the *value* which specifies a unique name of the operations exposed by the service. There are three more annotations which are used in the process of semantic enrichment: *@Input*, *@Output*, and *@ChangeEffect*. The *@Input* annotation specifies that a particular parameter has semantic meaning and is accessible under *inputParameterName*. The *@Output* annotation has a similar meaning but in the context of invocation results. In the presented example, some results of method invocation are accessible under the name specified by *outputParameterName*. The final annotation, *@ChangeEffect*, is used to define the semantic effect of the execution of a given method. In the present case, it states that execution changes the value of some resource (specified by the *propertyPaths* value) to 0.

Once the process of annotating the service interface and adapting its business logic to semantic interfaces

Listing 4 Service properties and ontology resource annotations

```
@UniversAALService(namespace="ServiceNamespace",name="ServiceName")
@OntologyClasses(value = { })
public interface ServiceInterface {
...
```

Listing 5 Service method annotations

```
@ServiceOperation(value = "serviceOperationName")
@Output(name = "outputParameterName")
@ChangeEffect(value = "0", valueType = Integer.class, propertyPaths = {
        ""  })
public Object[] businessMethod(@Input(name = "inputParameterName")
        Object parameter);
```

concludes, Iteration B is nearly finished. Information provided by annotations is used by the AAPI in the process of generating semantic descriptions and expectations (F3). During the process of registering a *Provider Implementation* which implements the specified business interface, the following actions occur:

- AAPI checks whether the registered service implements an interface which contains *@UniversAALService*. If so, then:
    - for each interface method annotated with *@ServiceOperation,* a dedicated *Service Profile* is generated;
    - each *Service Profile* is automatically enriched with information on the basis of *@Input*, *@Output*, and *@ChangeEffect* annotations resulting in functionality presented in Listing 2.
- AAPI generates a *Service Callee*, enriches it with service-specific properties from *@UniversAALService* and *OntologyClasses* and registers it on the bus with previously created *Service Profiles*.

The provided business interface can also be used in the process of developing a *Consumer Implementation*. It can be passed to the AAPI lookup process which returns a proxy implementing the specified interface. When the consumer invokes a method of such a proxy, AAPI generates a proper *Service Request* using the information provided in interface annotations, and passes it to the bus. The resulting code corresponds to Listing 3.

The described process completes Iteration B. If no errors are identified at this stage, Iteration C may commence.

Listing 6 presents a sample block of code of the LightingService interface with full semantic information. For the purpose of this example, it was necessary to modify several annotations:

- *@OntologyClasses* were provided with names of ontology resources used by the service;
- *@Input*, *@Output*, and *@ChangeEffect* annotations were enriched with *propertyPaths* parameters;
- interface method arguments and return values now refer to specific ontological resources.

Such annotated interface can be used to test semantic validity and verify that the application works as expected.

During the final iteration (Iteration D), it might turn out that some of the ontology resources or business methods are not needed on the consumer side. In such a case, the business interface provided on the consumer side should be abridged, i.e., the list of resources specified in the *@OntologyClasses* annotation might be trimmed or particular methods deleted. This approach simplifies the *Consumer Implementation*; however, it does not alter the functional aspects of the consumed service. Furthermore, with the use of AAPI for semantic enrichment, one can easily change the *Provider Implementation* of the business logic while no changes have to be introduced in other parts of code (on either the consumer's or provider's side).

5 Evaluation

This section presents a specific case implementing the proposed solution. The use case focuses on providing universal access to elderly or otherwise impaired persons. The application of the methodology presented in Sect. 3 is thoroughly described. Each of the iterations is analyzed in detail, highlighting the contribution of the paper. The final part of this section contains a discussion of achieved results.

5.1 Case study

The presented case study involves an airport support system for elderly and impaired travelers. The example focuses on communication between services deployed in the airport wireless network and mobile devices carried by travelers. The service is called *AirportLocalizer* and helps people reach their intended destinations.

The methods of the service are shown in Listing 7. The assisted person can ask for precise directions that he/she has to take in order to get to his/her flight, depending on the

Listing 6 Lighting interface containing full semantical information

```
@UniversAALService(namespace = "LightingNamespace", name = "
        LightingService")
@OntologyClasses(value = { Lighting.class })
public interface LightingInterface {

    @ServiceOperation(value = "getControlledLamps")
    @Output(name = "controlledLamps", propertyPaths = { Lighting.
            PROP_CONTROLS })
    public LightSource[] getControlledLamps();

    @ServiceOperation
    @Outputs(value = {
        @Output(name = "brightness", filteringClass = Integer.class,
                propertyPaths = {
            Lighting.PROP_CONTROLS, LightSource.PROP_SOURCE_BRIGHTNESS
                }),
        @Output(name = "location", filteringClass = Location.class,
                propertyPaths = {
            Lighting.PROP_CONTROLS, Resource.uAAL_VOCABULARY_NAMESPACE
                + "hasLocation" }) })
    public Object[] getLampInfo(
        @Input(name = "lampURI", propertyPaths = { Lighting.
                PROP_CONTROLS }) LightSource lamp);

    @ServiceOperation
    @ChangeEffect(propertyPaths = { Lighting.PROP_CONTROLS, LightSource
            .PROP_SOURCE_BRIGHTNESS },
        value = "0", valueType = Integer.class)
    public void turnOff(
        @Input(name = "lampURI", propertyPaths = { Lighting.
                PROP_CONTROLS }) LightSource lamp);

    @ServiceOperation
    @ChangeEffect(propertyPaths = { Lighting.PROP_CONTROLS, LightSource
            .PROP_SOURCE_BRIGHTNESS },
        value = "100", valueType = Integer.class)
    public void turnOn(
        @Input(name = "lampURI", propertyPaths = { Lighting.
                PROP_CONTROLS }) LightSource lamp);
}
```

current location and type of impairment (`getDirections()` method). Different implementations of the service could be installed in the airport's infrastructure, each covering a different type of disability. For example, a blind person can obtain directions in the form of instructions read aloud by the mobile device; a person with visual impairment will be directed along routes equipped with Braille inscriptions and tactile pavings; a wheelchair-bound traveler will be directed to elevators rather than staircases, etc. Apart from obtaining directions, the service client may also request the system to call a nearby elevator (methods `getElevatorNearby()` and `callElevator()`)—in this way, people with serious handicaps do not need to push a call button (which might be invisible or unreachable for them).

5.2 Iterative development methodology execution

The service is implemented in accordance with the Iterative Development Methodology, using the universAAL framework extended with AAPI. The following section explains what tasks are undertaken during each iteration. Both providers and consumers differ with respect to the types of targeted handicaps (e.g., directions read by a text-to-speech module or a map displayed on the device screen; avoiding stairs etc.) As a proof of concept, one specific implementation is assumed in iterations A–C (its specific type is irrelevant for evaluation purposes).

Iteration A focuses on creating proper business logic. First, an interface of the service is developed in line with the expected business process. In this case, the interface from Listing 7 is used and two implementations created. On the provider's side the implementation includes the following:

- algorithms to determine the best route, starting from the caller's current location, to the given destination, depending on the caller's impairment (`get-Directions()` method);
- choosing the next elevator along the current route (`getElevatorNearby()`); and
- controlling the state of the elevator upon client request (`callElevator()`).

Listing 7 Bare service interface

```
package org.universAAL.ontology.airport.localization;

public interface AirportLocalizer {
    public Directions getDirections(Location myLocation, Location
           destination);
    public Elevator getElevatorNearby(Location myLocation);
    public void callElevator(Elevator elevator, int level);
}
```

On the consumer's side, the following implementations have to be provided:

- presenting directions which suit the user's impairment type (`getDirections()`);
- retrieving information describing the target location (not related to this service);
- discovering the nearest elevator (`getElevatorNearby()`);
- when approaching the elevator, sending a call request (`callElevator()`).

Finally, by using the framework's dependency injection mechanisms, the service instance can be injected into consumer code. This allows the business logic to be executed and thus properly tested. At this point, it is assumed that the business logic has been approved.

Iteration B adds a minimal amount of semantics into the created interface. First, Ontology Concept Placeholders have to be identified and annotated. The following elements have to be defined (by adding annotations provided with the framework to certain elements of the service interface):

- name and namespace (`@UniversAALService` annotation);
- ontology class names (`@OntologyClasses`)—for now these classes are merely mockups of the actual ontology that will be created in the next iteration;
- service methods (`@ServiceOperation`);
- input and output parameters, together with their names (`@Input` and `@Output`).

Once this step is complete, Service Description and Service Expectation are automatically generated by the framework. During this process, the service is automatically configured to be exposed in the AAL platform. The system also generates a consumer's invocation, capable of finding the service, wrapping the call into a request, sending it over the platform and unwrapping the received response. Finally, the generated implementations are deployed in the platform and tested again. The *Match-making* process can now be invoked for the first time.

Iteration C covers the ontology development and plugs it into the service. All Semantic Concepts of the *AirportOntology* are created at this point (this includes all the routes that passengers can take to their flights, information on which routes can be taken by travelers depending on their impairments, elevators and their possible states, location in the airport area) As this paper does not cover ontology definition, the process will not be discussed in detail. Having created the ontology, the placeholders in the interface can be linked to the relevant Concepts (the `@ChangeEffect` annotation, `propertyPaths` and `filteringClass` parameters are added to the interface). The final form of the annotated interface is presented in Listing 8. Lastly, the ontology, along with the provider's and the consumer's generated code, is deployed in the platform. The semantically complete *Match-making* process can now be executed and tested.

Iteration D deals with creating different Service Expectations and Descriptions. One additional implementation of the provider and consumer code, targeting a different type of disability, is created. Iterations A–C are repeated using a different business logic and a single differing annotation.[1]

The development phase of the *AirportLocalizer* service concludes with the end of Iteration D.

5.3 Discussion

Having the service development process completed, a summary and evaluation of the contribution and comparison with *status quo* solutions can be performed. The input of the development process included:

- a specific use case involving a service which can be deployed at an airport to support the elderly and impaired people;
- the universAAL framework designed to provide an efficient platform for designing, developing and deploying ready-to-use services within an existing infrastructure.

As the paper's contribution, the presented environment was enriched to add value to the development process in order to facilitate universal access provisioning. This enrichment covers the following elements:

[1] The `propertyPaths` parameter has `Airport.MOVEMENT_IMPAIRMENT` instead of `Airport.BLIND_IMPAIRMENT` in `@Output` annotation of the `getDirections()` method because the semantics of that method change according to the impairment type.

Listing 8 Fully annotated service interface

```
package org.universAAL.ontology.airport.localization;

import org.universAAL.middleware.api.annotation.*;
import org.universAAL.middleware.rdf.Resource;
import org.universAAL.ontology.airport.*;

@UniversAALService(namespace = "LocalizationNamespace",
  name = "LocalizationService")
@OntologyClasses({ Localizer.class })
public interface AirportLocalizer {

  @ServiceOperation
  @Output(name = "directions", filteringClass = Directions.class,
    propertyPaths = { Localizer.PROP_CONTROLS, Airport.
          LOCALIZATION_SYSTEM, Airport.BLIND_IMPAIRMENT })
  public Directions getDirections(
    @Input(name = "myLocation", propertyPaths = { Localizer.
          PROP_CONTROLS, Airport.LOCALIZATION_SYSTEM })
      Location myLocation,
    @Input(name = "destination", propertyPaths = { Localizer.
          PROP_CONTROLS, Airport.LOCALIZATION_SYSTEM })
      Location destination
  );

  @ServiceOperation
  @Output(name = "elevator", filteringClass = Elevator.class,
    propertyPaths = { Localizer.PROP_CONTROLS, Airport.
          LOCALIZATION_SYSTEM })
  public Elevator getElevatorNearby(
    @Input(name = "myLocation", propertyPaths = { Localizer.
          PROP_CONTROLS, Airport.LOCALIZATION_SYSTEM })
      Location myLocation
  );

  @ServiceOperation
  @ChangeEffect(propertyPaths = { Localizer.PROP_CONTROLS, Airport.
        LOCALIZATION_SYSTEM, Airport.ELEVATOR, Elevator.MOVING }, value
         = "true", valueType = Boolean.class)
  public void callElevator(
    @Input(name = "elevator", propertyPaths = { Localizer.PROP_CONTROLS
          , Airport.LOCALIZATION_SYSTEM })
      Elevator elevator,
    @Input(name = "level", propertyPaths = { Localizer.PROP_CONTROLS,
          Airport.LOCALIZATION_SYSTEM, Airport.ELEVATOR })
      int level
  );
}
```

- the Iterative Development Methodology presented in Sect. 3;
- AAPI presented in Sect. 4.

Based on the enriched environment, a service supporting universal access was created. Two differing implementations were provided, targeting different types of disability. Both the provider's and the consumer's side were linked with a common ontology and synchronized with each other, ensuring seamless communication.

The most important advantage of the proposed solution is the division of work into four distinct iterations, making it easier for the developer to handle this otherwise complex and error-prone task. Such incremental development also enables progressive validation of results. Simple mistakes can be easily diagnosed and their causes eliminated. Moreover, the proposed methodology introduces clear separation of concerns. Tasks concerning business logic, ontology modeling and development work on the provider's and the consumer's side remain separate.

Another advantage is the fact that during the development process, the provider's and the consumer's code remain fully synchronized. Implementation of the ontology is easily shared and simultaneously distributed to both sides. This ensures code compatibility and mitigates pernicious programming errors. AAPI delivers these advantages in a fully automatic way, enabling the developer to focus on creating error-free code.

The final advantage worth mentioning is that the burden of creating many boilerplate code segments is lifted from the developer. AAPI takes care of automatic code generation, guaranteeing its syntactical and semantical correctness. The amount of code created by the developer using the proposed solution was compared to a scenario without AAPI. The results are shown in Table 1.

Table 1 Comparison of the amount of code created by the developer using different approaches

	Standard approach	Proposed approach
Lines of code	~380	~150
Number of classes containing semantic information	3	1

Regardless of the benefits mentioned above, one drawback of the proposed approach must be acknowledged. Iteration D involves steps similar to iterations A–C, focusing on a different implementation. A mechanism which would synchronize this process and handle large numbers of parallel implementations of the same service would be a significant improvement.

In addition to the discussion presented in this paper, a real-life evaluation of the proposed approach was undertaken. AAPI was contributed directly to the universAAL project and met with very positive reception from the developers' community.[2] Unfortunately, this is not evidenced by any citeable scientific document.

Taking the above into consideration, AAPI is a highly promising solution. The approach represented by the AAL environment have thus far lacked appropriate tooling support. The system presented in this paper remedies this situation.

6 Conclusions and future work

Several conclusions can be drawn on the basis of the presented study. The proposed Iterative Development Methodology of Service Semantics tackles the inherent complexity of the development process. Each development task, such as implementation of business logic or ontology modeling, is enclosed in a separate iteration. Results of each iteration are directly deployable and testable, which reduces the risk of errors. The features identified by the Methodology could be perceived as a recipe for a simple and user-friendly Semantic Framework. They free the developer from having to manually develop and maintain *Service Descriptions* and *Service Expectations*. They also automate extensive parts of consumer's and provider's implementations.

In order to assess the novelty of the presented work, the AAL platforms presented in Sect. 2 were analyzed in the context of the features expected by the proposed Iterative Development Methodology. Table 2 contains the summary of the support for these features provided by the reviewed platforms. The feature fulfillment is marked with (1) "+"—when a feature is fully supported; (2) "−"—when a feature is not supported; or (3) "+/−"—when a feature is supported partially. The cases of full and partial feature support are now elaborated.

The SOPRANO platform provides its middleware components exposed via well defined APIs, though does not let developers describe services semantically in a declarative manner, e.g., with annotations (F4). The *Composer* component of the SOPRANO platform performs match-making between service instance and abstract semantic description. Thanks to that, feature F6 is fully supported. The OASIS project provides the mechanisms for tagging the business services with semantic information. The mechanism involves mapping specific web-services operations to the ontologies stored in repositories.[3] Such approach allows for achieving complete support for feature F2. The need for the development of an additional layer in the form of web-services which are well isolated from specific business logic but contain semantic information enables full support for F6.

Regarding the MonAMI approach, F2 is partially supported by introducing a two-level mapping: services to functions and functions to devices (sensors/actuators) [12]. Having this, the independence between *Service Expectation* and the *Description* at the development time (F6) could also be achieved. In the PERSONA and universAAL projects, the F4 and F6 features are partially supported (both projects present the same fulfillment of the features because PERSONA was the input project on which universAAL was heavily based)—the functionalities are implemented though are not usable in a straightforward manner. Developers are unable to specify the semantics declaratively and a significant amount of boiler-plate code still needs to be written manually, which can lead to hard to diagnose run-time errors.

The feature of Dependency Injection is fully supported in almost all analyzed projects. All projects, beside OASIS, are based on the OSGi technology which provides a capability of Declarative Services and, starting from version 4.2 of OSGi specification [28], the Blueprint Container. Both capabilities allow for the realization of the Dependency Injection pattern. In the case of the OASIS project, there are mechanisms that partially support the feature of Dependency Injection. It is realized not by means of injecting dependencies in the business logic, but injecting the services necessary for given application on the basis of ontological description. The whole process is

[2] Documentation of AAPI is provided on the following site http://forge.universaal.org/wiki/support:RD_Core_AAPI In order to access it a registration of a free account is needed.

[3] The mapping is performed with the use of Content Anchoring and Alignment Tool.

Table 2 Summary of the support for the features of iterative development methodology provided by different AAL platforms

	F1	F2	F3	F4	F5	F6
SOPRANO [11, 33]	+	–	–	±	–	+
OASIS [10]	±	+	–	–	–	+
MonAMI [13, 15]	+	±	–	–	–	±
PERSONA [4, 27]	+	–	–	±	–	±
universAAL [5]	+	–	–	±	–	±

F1—Dependency injection

F2— Ontology placeholder annotation

F3—Automatic generation of service description and expectation

F4— Complete ontology annotations

F5—Automatic enriching description, expectation with semantics

F6 —Independence between declarative service expectation and the description at the development time

realized in an automated way by an internal component called AMI Framework.

The analysis of the reviewed AAL platforms in the context of features expected by Iterative Development Methodology shows that the support for these features was not extensively provided. Therefore, none of the platforms allows for direct realization of the proposed Methodology. In particular, features related to automatic enrichment (F5) and code generation (F3) specific for the given *Semantic Descriptions* and *Expectations* (which significantly simplifies the development process of services semantics) are not supported by the current AAL platforms. Moreover, it is apparent that the separation between semantics and the business logic (F2, F4) is not handled properly in most of the projects where those two elements are tightly coupled with each other. The conclusions resulting from the AAL platforms analysis ensure that the proposed approach for seamless semantic service enrichment is a novel concept. The proposed Methodology aids the development of service semantics to the extent which was not achievable before.

The applicability of the proposed Methodology has been verified using a fully featured implementation—the Annotation-based Semantic Enrichment (AAPI)—an extension of the universAAL platform Semantic Framework. Evaluation performed using a sample scenario related to accessibility support shows that the development of semantics becomes simpler and less error-prone compared to existing tools. Additional good feedback from the universAAL community strengthens these evaluation results. Taking all of the above into consideration, it can be said that the proposed Methodology and AAPI directly enhance the potential of the universAAL platform and increase its chances for gaining broad acceptance on the AAL market.

Although the presented Methodology was implemented for universAAL, this is just one of its possible realizations. The Methodology is founded on abstract semantic aspects, common to many different AAL platforms. Thus, it can conceivably be reimplemented for other AAL platforms.

Current work on the Methodology and AAPI focuses on service interactions. In the context of future work, the authors would like to extend proposed solutions with asynchronous event-oriented interactions and with interactions focusing strictly on the user interface layer. Especially the latter improvement would allow for addressing recent challenges of universal access [20] and therefore increasing the extent to which the solutions proposed support the developer in the implementation and maintenance of truly accessible applications.

Acknowledgments The research presented in this paper was partially supported by the Polish Ministry of Science and Higher Education grants no. 15.11.120.264, 15.11.120.268 and also by the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 247950.

References

1. Ambient Assisted Living Strategic Research Agenda (2010). http://www.aaliance.eu/
2. Emiliani, P.L., Stephanidis, C.: Universal access to ambient intelligence environments: opportunities and challenges for people with disabilities. IBM Syst. J. **44**(3), 605–619 (2005). doi:10.1147/sj.443.0605
3. Fagerberg, G., Kung, A., Wichert, R., Tazari, M.R., Jean-Bart, B., Bauer, G., Zimmermann, G., Furfari, F., Potortì, F., Chessa, S., Hellenschmidt, M., Gorman, J., Alexandersson, J., Bund, J., Carrasco, E., Epelde, G., Klima, M., Urdaneta, E., Vanderheiden, G., Zinnikus, I.: Platforms for AAL applications. Proceedings of the 5th European conference on Smart sensing and context. EuroSSC'10, pp. 177–201. Springer, Berlin (2010)
4. Fides-Valero, A., Freddi, M., Furfari, F., Tazari, M.R.: The persona framework for supporting context-awareness in open distributed systems. Proceedings of the European Conference on ambient intelligence. Am I '08, pp. 91–108. Springer, Berlin, (2008)
5. Hanke, S., Mayer, C., Hoeftberger, O., Boos, H., Wichert, R., Tazari, M.R., Wolf, P., Furfari, F.: Universaal an open and consolidated aal platform. In: Wichert, R., Eberhardt, B. (eds.) Ambient Assisted Living, pp. 127–140. Springer, Berlin (2011)
6. I2Home IST project. http://www.i2home.org
7. ICT & Aging—European study on users, markets and technologies. Tech. Rep., empirica and WRC on behalf of the European Commission, Directorate General for Information Society and Media (2010). http://www.ict-ageing.eu/
8. Karim, S., Latif, K., Tjoa, A.M.: Providing universal accessibility using connecting ontologies: a holistic approach. In: Proceedings of the 4th international conference on Universal access in human–computer interaction: applications and services (2007)
9. Karim, S., Tjoa, A.M.: Towards the use of ontologies for improving user interaction for people with special needs. In: ICCHP, Lecture notes in computer science. Springer (2006)

10. Kehagias, D.D., Tzovaras, D., Mavridou, E., Kalogirou, K., Becker, M.: Implementing an open reference architecture based on web service mining for the integration of distributed applications and multi-agent systems. In: Proceedings of the 6th international conference on agents and data mining interaction, ADMI'10, pp. 162–177. Springer, Berlin (2010). http://dl.acm.org/citation.cfm?id=1880493.1880511
11. Klein, M., Schmidt, A., Lauer, R.: Ontology-centred design of an ambient middleware for assisted living: the case of SOPRANO. In: Kirste, T., Knig-Ries, B., Salomon, R. (eds.) Towards ambient intelligence: methods for cooperating ensembles in ubiquitous environments (AIM-CU), 30th Annual German Conference on artificial intelligence (KI 2007), Osnabrck, (2007)
12. Kung, A., Fagerberg, G.: Alliance for an AAL open service platform. In: AALIANCE conference—Malaga, Spain (2010)
13. Kung, A., Jean-Bart, B.: Making aal platforms a reality. Proceedings of the first international joint conference on ambient intelligence. Am I'10, pp. 187–196. Springer, Berlin (2010)
14. Leitner, M., Subasi, O., Höller, N., Geven, A., Tscheligi, M.: User requirement analysis for a railway ticketing portal with emphasis on semantic accessibility for older users. In: Proceedings of the 2009 international cross-disciplinary conference on web accessibililty (W4A), W4A '09, pp. 114–122. ACM, New York (2009). doi:10.1145/1535654.1535683
15. Marco, l., Casas, R., Bauer, G., Marn, R.B., Asensio, N., Jean-Bart, B., Ibane, M.: Common OSGi interface for ambient assisted living scenarios. In: Gottfried, B., Aghajan, H.K. (eds.) BMI Book, Ambient Intelligence and Smart Environments, vol. 3, pp. 336–357. IOS Press (2009)
16. Marinc, A., Stockloew, C., Tazari, M.R.: 3d interaction in aal environments based on ontologies. In: Ambient Assisted Living, pp. 289–302. SpringerLink (2012)
17. Minon, R., Aizpurua, A., Cearreta, I., Garay, N., Abascal, J.: Ontology-driven adaptive accessible interfaces in the inredis project. In: Int. Workshop on Architectures and Building Blocks of Web-Based User-Adaptive Systems (2010)
18. MonAMI IST project. http://www.monami.info
19. MPower IST project. http://www.sintef.no/Projectweb/MPOWER
20. Newell, A.F., Gregor, P.: User sensitive inclusive design in search of a new paradigm. Proceedings on the 2000 conference on Universal Usability. CUU '00, pp. 39–44. ACM, New York, NY, USA (2000)
21. Oasis IST project. http://www.oasis-project.eu
22. Persona, IST project. http://www.aal-persona.org
23. Pöttner, W.B., Wolf, L.: Ieee 802.15.4 packet analysis with wireshark and off-the-shelf hardware. In: Proceedings of the Seventh International Conference on Networked Sensing Systems (INSS2010). Kassel, Germany (2010)
24. Prasanna, D.R.: Dependency Injection, 1st edn. Manning Publications Co., Greenwich, CT, USA (2009)
25. Signore, D.O.: Ontology driven access to museum information (2005)
26. Soprano, IST project. http://www.soprano-ip.org
27. Tazari, M.R.: Using queries for semantic-based service utilization. In: Proceedings of CEUR Workshop (2009)
28. The OSGi Alliance: OSGi Service Platform Compendium Specification —Release 4, Version 4.2 (2009)
29. The OSGi Alliance: OSGi Service Platform Core Specification—Release 4, Version 4.2 (2009)
30. UniversAAL IST project. http://www.universaal.org/
31. Van Den Broek, G., Cavallo, F., Wehrmann, C.: AALIANCE Ambient Assisted Living Roadmap. IOS Press, Amsterdam (2010)
32. Wichert, R.: Configuration and Dynamic Adaptation of AAL Environments to Personal Requirements and Medical Conditions. Proceedings of the 5th international on conference universal access in human–computer interaction. Part II: Intelligent and ubiquitous interaction environments, UAHCI '09, pp. 267–276. Springer, Berlin (2009)
33. Wolf, P., Schmidt, A., Klein, M.: SOPRANO—An extensible, open AAL platform for elderly people based on semantical contracts. In: 3rd Workshop on Artificial Intelligence Techniques for Ambient Intelligence (AITAm I08), Patras, Greece (2008)
34. Wolf, P., Schmidt, A., Otte, J.P., Klein, M., Rollwage, S., Knig-Ries, B., Dettborn, T., Gabdulkhakova, A.: Openaal—the open source middleware for ambient-assisted living (aal). In: AALIANCE conference, Malaga, Spain, March 11–12, (2010)