universAAL IOT – a Technical Overview

Topics

Semantic Discovery & Interoperability

Service Broker & Orchestrator

Context Broker, Context History Entrepôt, & Semantic Reasoning

Human-Environment Interaction

Openness

Maturity

Status w.r.t. major Requirements

universAAL IoT is a mature open platform[1] for the integration of open distributed systems of systems, with over 15 years of development history, from general conceptual work in German research projects EMBASSI and DynAmITE, over proof of concept with unique problem solving approaches in FP6 PERSONA, consolidation and first tooling in FP7 universAAL, and stress testing in real life in CIP ReAAL, which has led to the creation of the initial universAAL IoT ecosystem. By implementing semantic interoperability for service-oriented architectures at the level of communication protocols, it provides an open horizontal service integration layer at the highest abstraction layer, across all possible verticals, which makes it a solution for many of today's system problems. universAAL IoT is distributed with the Apache Software License 2.0, available under https://github.com/universAAL/.

In the core of universAAL IoT is a middleware that understands data and functionality only through the definition of ontologies. A set of ontologies is already available in the universAAL code base, primarily covering the domains of Smart Environments and Ambient Assisted Living (AAL). The usefulness of universAAL does not limit though to these domains: it is enough to add the ontologies that cover the desired new application domain in order for universAAL to adapt.

The universAAL middleware is basically an enabler for the exchange of messages among autonomous components. It is comprised of (1) a common container interface (for portability, configuration, logging and localization – in terms of adapting to a local context), (2) the essential data representation and serialization needed for unified messaging using the machine-readable RDF/Turtle standard for exchanging both models (ontologies) and data instances (the standard simultaneously serves as the basis for enabling semantic interoperability), (3) node discovery and messaging within an authorized group of middleware instances for easy connectivity and interaction of universAAL enabled nodes within the same network, which forms the basis for universAAL to act as the operating system of an ensemble of devices, and (4) a set of virtual communication buses as semantic brokers for context events, semantic service requests (and responses), and user interaction functions. As a result, the universAAL middleware is able to hide distribution and heterogeneity within the device ensemble, and facilitate the integration of components as well as the communication among them at a semantic level.

[1] http://universaal.info/blog/post/3487/ What-is-anopen-platform/

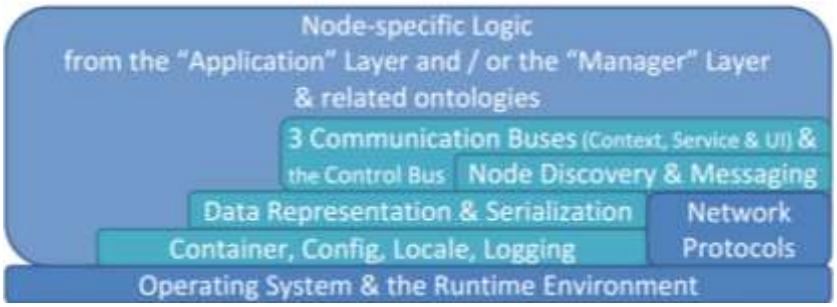

Figure 1. universAAL middleware stack.

universAAL IoT ships the middleware along with a set of managers, mostly optional, that help implement common use cases, such as storing, reasoning, service composition, remote interoperability (connecting with services and entities outside the uSpace – the universAAL-based intelligent space), user interaction extensions, security functions, device (sensors and actuators) protocol interconnections, etc.

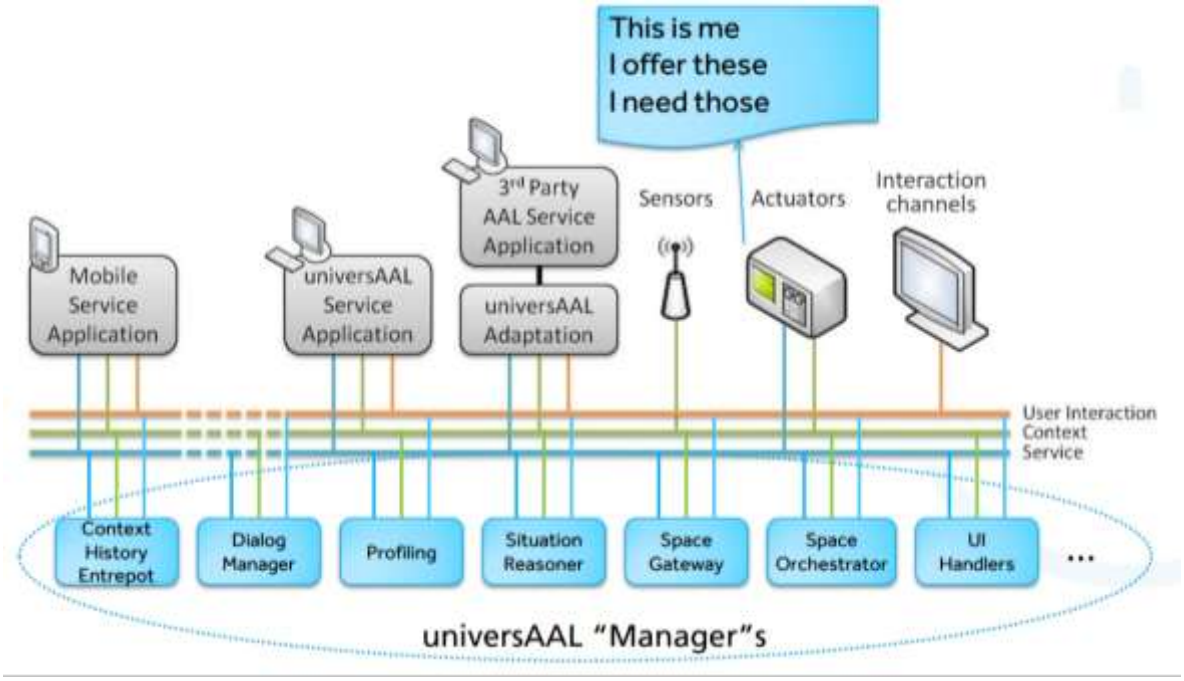

Figure 2. Like applications, universAAL Managers connect to the buses of the universAAL middleware.

As part of the platform, the universAAL IoT community also provides a set of tools to ease the development, deployment and maintenance of the systems developed with universAAL.

Semantic Discovery & Interoperability

The first discovery level in universAAL is related to the mutual discovery of middleware instances within a subnet, which is based on network broadcast; concretely, universAAL uses java SLP framework for this node discovery level. Once a node is discovered, the peering phase starts in order to check compatibility and authorization. Among compatible and authorized peers, communication is established based on the concept of a "connector", which enables the creation of a logical network for message distribution, called a uSpace (universAAL Space). Currently, the only implementation of the "connector" concept is based on the jGroups framework. However, all modules involved in this layer, no matter if for discovery, peering or establishing communication, are based on well-defined APIs so that different frameworks and solutions can be used in diverse combinations to provide alternative implementations for this layer.

The uSpace groups all nodes in a smart environment and enables them to communicate in a secure way. Messages in a uSpace are secured using a common symmetric cryptographic key. The distribution of messages can be direct (node to node), multicast (node to set of nodes), or broadcast (node to all nodes). These messages are classified according to patterns of communication (publish/subscribe, general request/response, or request/response for interacting with human users); thus, there are different "buses", each serving its specific purpose using an appropriate own brokerage strategy, where a bus is a message broker in charge of routing the messages to the appropriate software modules on appropriate nodes based on its brokerage strategy.

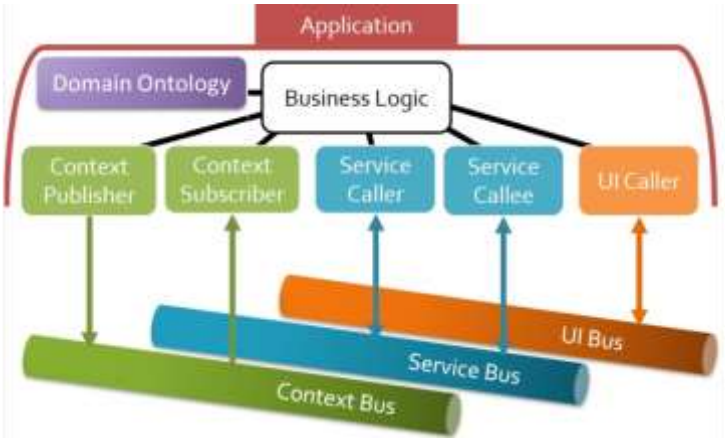

Figure 3. Integration with universAAL is equivalent to "talking" to its buses with specific roles.

The second level of discovery is the semantic discovery provided by the middleware buses. Modules that connect to the buses must describe their capabilities and requirements semantically and register them with (when they connect to the buses) / send them to (dynamically during runtime) the buses. The routing of messages with semantically formulated content is done by the buses based on a sematic matchmaking, which effectively discovers the proper provider (of functionality) or consumer (of data).

The bus abstraction is the primary concept behind the paradigm of semantic interoperability as realized by universAAL. In universAAL, developers use the provided data structures and protocols of a bus to talk to it in a semantic manner and the bus responds accordingly. This is different from the conventional understanding that limits semantic interoperability to linking data and enabling semantic search, often in a parallel world of annotations. Consequently, the extent of semantic interoperability is usually limited to finding "objects" by semantic search and then calling their functions based on an a-priori agreed API in addition to the agreed ontology.

As an example, for a component in such environments that wants to reach the goal that, say, the entrance door is opened, it may perform a semantic search based on (1) a given ontology to find the object representing the entrance door, but eventually it has to rely on (2) a given API and for example call the predefined operation "open" in order to reach its original goal. In the universAAL world, however, the ontology is utilized to directly instruct the service bus to "open the entrance door", eliminating the need for an additional API. As ontologies may exist independently from all the three parties (including the universAAL buses performing the brokerage), this totally decouples the requester / subscriber from the responder / publisher in terms of both the technical details of communication and programming syntax.

Semantic interoperability as realized by universAAL causes that all dependencies between the participants in a communication are shifted to models shared by them (shared ontologies or compatible ones with an existing mapping between them), this way reducing all syntactical dependencies in the communication to the usage of a single standard API of the brokerage mechanism implemented by the middleware buses.

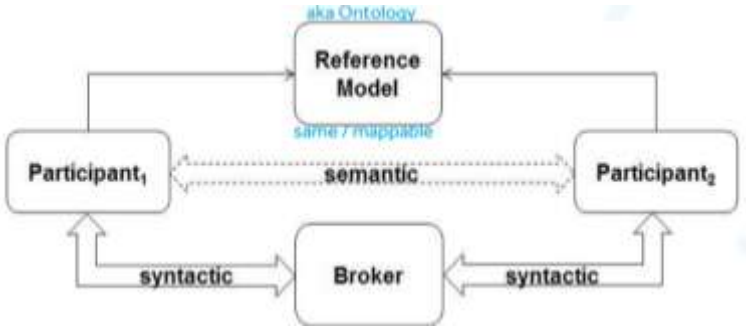

Figure 4. Direct syntactic dependency is eliminated using the single API of a broker, which enables semantic interoperability through shared (or mapped) ontologies.

In the above view, interoperability is the ability of two communicating entities to share data and functionality despite having been developed independently. Semantic Interoperability enables interoperability at a semantic level, avoiding domain-specific APIs and reducing syntactical dependencies to one single brokerage API. Specific use case dependencies can be resolved through common (or specific) shared ontologies which are pluggable into the system.

Service Broker & Orchestrator

The first broker is the Service Bus responsible for request / response pattern, also known as “call-based” communication. This bus enables the provision and consumption of services in a service-oriented environment. Both the registration of provided services (using a data structure called “ServiceProfile”) and the request of services (using the construct called “ServiceRequest”) are concentrated on the concept of “service results”, i.e. `some info returned´, or `an effect achieved´ or a combination of them. In service profiles, providers can formulate which service results can be achieved by initiating which processes; requesters formulate the expected service results in their service requests. By semantic matchmaking, the service bus as the broker tries to find out which registered processes should be initiated in order to get the expected service results. Then it asks the provider to start the process of providing the underlying service, waits to get the response, maps this response back to the structures usable by the requester, and finally returns the adapted response to the original requester. If more than one provider were asked to execute their appropriate processes (which could happen under certain circumstances), the responses would be merged into a single response.

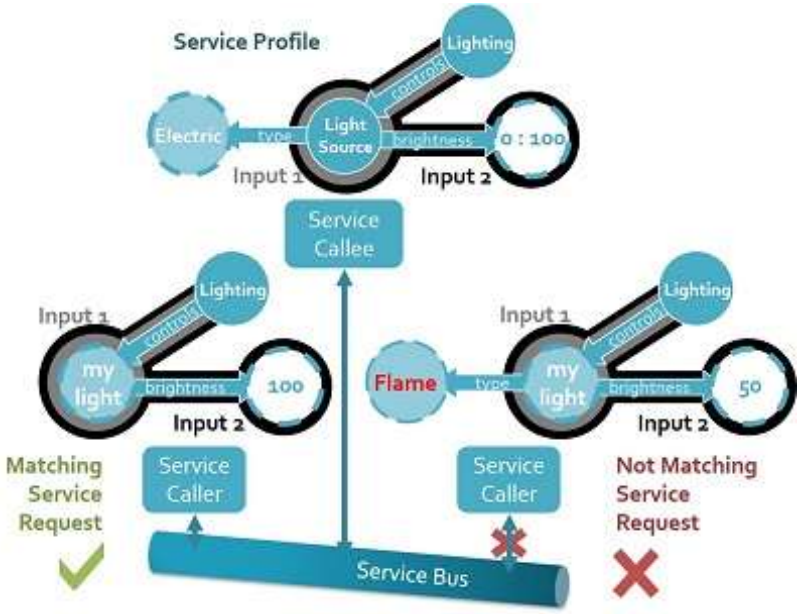

Figure 5. Visualization of Service bus matchmaking.

As a matter of fact, registered processes may be parametrized in order to achieve a range of different results depending on the provided parameters. The interesting thing is that the semantic matchmaking in universAAL can find out which of the info items in the request should be passed as parameter in order to get the expected result.

Composite services can be defined using JavaScript – currently the only scripting language supported. The universAAL workflow engine – that interprets and executes such scripts – is called the Service Orchestrator.

Context Broker, Context History Entrepôt, & Semantic Reasoning

Context awareness is a key aspect of intelligent environments. Context is any information reflecting the state of any parameter inside such environments, no matter if this state is provided by a "controller", estimated by a sensor, derived by a reasoner, or explicitly provided by a human user. Context awareness is achieved through ontology-based data sharing, which can be both pushed through the "context broker" or pulled through the service broker.

Thus, the context bus that serves as the context broker, is the bus responsible for the publish/subscribe pattern, also known as the event-based communication; it distributes context information within the uSpace. Messages accepted by the context bus are either context events (sent by context publishers) or context event patterns (sent by context subscribers).

Each context event represents a change in the uSpace context, namely the change of the state value of a given "parameter" in the uSpace. Inspired by the RDF model, such parameters are identified in universAAL by a pair of URIs, one identifying a resource in the uSpace and the other a specific property of that resource whose value has changed. It should be obvious now that the triple constructed from the two URIs and the related new state value in a context event is equivalent to the concept of a statement in RDF; the first URI serves as the subject of the statement, the second one as the predicate and the state value as the object of the statement. Each context event has additional metadata, such as the provider info, a timestamp, and a possible expiration stamp.

Figure 6. The anatomy of a context event.

With context event patterns, context subscribers can specify the class of context events in which they are interested. This is done by introducing conditions on any of the properties of the accepted context events, no matter if subject, predicate, object or provider[2]. The semantic matchmaking on

[2] Assuming an ontology that defines "Person" as a class and "hasLocation" as a possible property of "persons", when a component using this ontology would like to be notified whenever a human changes the location, it

the context bus is simply based on checking if the context event at hand is an instance of the class of context events specified in each of the registered context event patterns. The context event will be forwarded to those components that have registered the matching patterns.

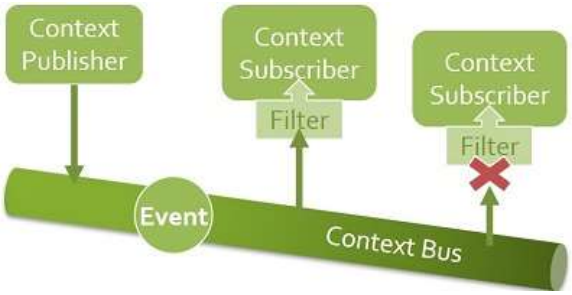

Figure 7. Context Broker strategy.

To make sure that also the context pull mechanism over service bus works, universAAL provides a special manager, called the Context History Entrepôt (CHE), that guaranties the persistence of context. CHE subscribes for all context events by registering a pattern without any restriction, and stores all context events received in an RDF database. This database is always applying the ontological inference mechanisms in the background, this way also acting as a reasoner that produces new knowledge based on known context. For providing access to this database, it registers service profiles on the service bus to offer for processing context queries. Two major types of queries can be processed by CHE: (1) SPARQL queries, and (2) those based on context event patterns. This allows applications to extract statistics and analyse the evolution of the context.

Additionally, two context reasoners enrich the universAAL support for context-awareness. One is using the drools rule engine to provide applications with the possibility of defining rules to be evaluated with every context event; these rules may produce new context events that are published to the context bus, make a service request to the service bus, or both. Another reasoner is called the Situation Reasoner, which uses SPARQL "construct" queries for inferring new facts that are then published to the context bus as a context event. Sending a SPARQL construct query to the situation reasoner causes a persistent storage of the query in order to be able to run the query whenever there is a chance that it can infer new facts.

Finally, the open and modular approach of universAAL to supporting context-awareness allows to add further context reasoners in addition to the above mentioned facilities for **semantic reasoning** (OWL-based inference in CHE, the drools rule engine and the Situation Reasoner) pretty easily.

Human-Environment Interaction

With its user interaction (UI[3]) framework, universAAL prepares for a shift of paradigm from human-computer (all interaction channels connected to one single computing device) to human-environment interaction (interaction channels distributed in the environment, not necessarily connected to one single computing device). By giving a recognized role to interaction channels, universAAL differentiates between two distinct layers with a brokerage mechanism in between: (1) the application layer consisting of all components that may have the need to interact with human users and (2) the presentation layer consisting of a new type of components called "UI Handlers" that are experts in utilizing interaction channels for handling interaction requests.

---

must register a context event pattern that restricts (1) the subject of the context events to be of type "Person" and (2) the predicate of the context events to be exactly equal to "hasLocation".

[3] In this section, the abbreviation "UI" stands for "user interaction" (and not "user interface"), especially in the compound names "UI Framework", "UI Bus", "UI Handler", "UI Request", and "UI Response".

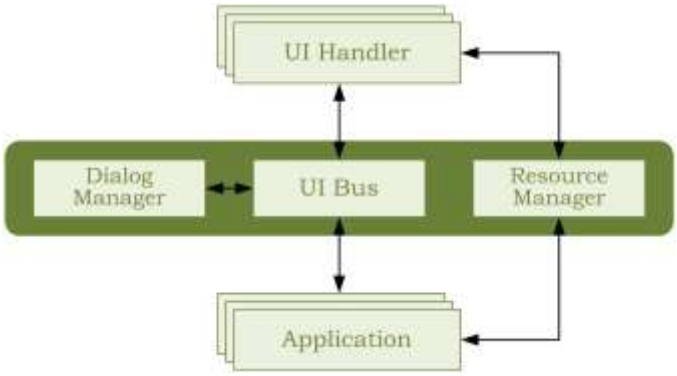

Figure 8. Components of the universAAL UI framework.

The brokerage mechanism between the application and presentation layers brokers the UI requests of applications to appropriate UI handlers in a situation-aware way. It consists of

- the “UI broker”, again as a virtual communication bus of the middleware called the UI bus,
- a model for specifying interaction requests in a modality neutral way and without any layout bias (inspired by the “Form Controls” of the W3C specifications for XForms),
- a “Resource Manager” for storing and retrieving alternative presentation resources (like visual icons and signs for different display sizes and their acoustic equivalents), and
- a “Dialog Manager” that, among others, (1) represents the whole uSpace by providing different flavours of system menus and standard dialogs, (2) assists the UI bus by being responsible for the incorporation of user context in analysing the interaction situations, (3) assists the UI Bus also by providing a persistent mechanism for user-specific dialog management, and (4) handles user instructions that are not in the context of a running dialog. Due to roles such as in (1) and (4), it is logically also playing the role of an application that creates own UI requests.

Based on this framework, which is promoted by ISO/IEC as a Publically Available Specification (PAS) with the number 62883:2014[4], applications send their UI requests consisting of the dialog to be performed, the addressed user, as well as the language, privacy level and priority of the dialog, in a declarative way to the UI bus and then wait until they receive the corresponding UI response from the bus. By receiving a UI request, the bus asks the Dialog Manager to enrich the request with the user context, such as the user’s current location and his/her interaction preferences and capabilities.

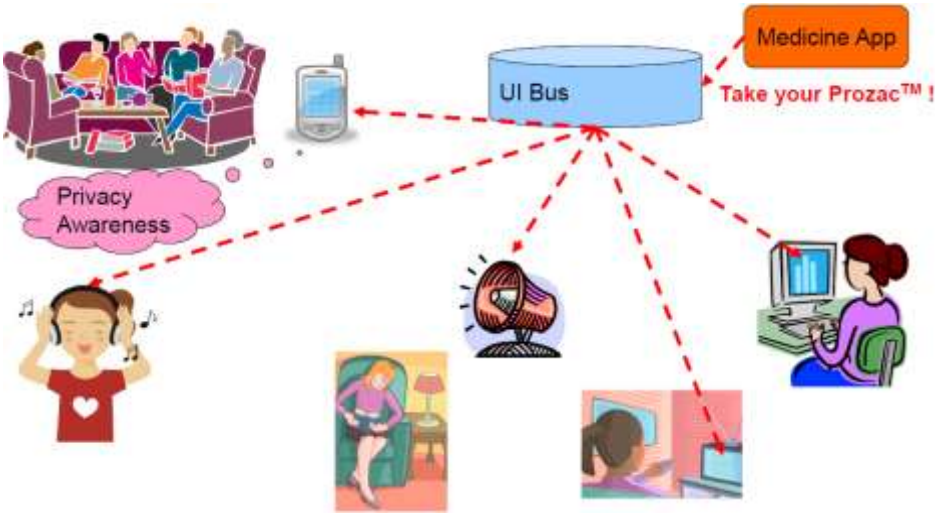

Figure 9. Context-aware selection of appropriate interaction channels.

[4] https://webstore.iec.ch/publication/7577

It then starts with the matchmaking of the enriched UI request against the registered UI handler profiles, trying to find the most appropriate UI handler for handling the UI request, taking into account different criteria like proximity to the user, the supported languages, and the privacy of the managed channels compared to the privacy required by the contents of the UI request. The bus then waits for the UI response from the selected UI handler, which it will forward to the application that had originally sent the corresponding UI request.

As a result, the UI framework of universAAL makes applications multi-modal automatically. Additionally, it provides for loss-less change of interaction channels in location-based sessions with users. If after the selection of the appropriate UI handler and before receiving the related response, the user context changes in a way that the currently selected UI handler loses its relevance (e.g., change of user location or the entrance of other people in the same location as the user), the bus will ask that UI handler to cut the execution of the dialog and return the UI request with all data collected from the user so far; it then forwards the request to the next UI handler which is the better match for the new situation. This way, scenarios, such as "follow me" where the dialog follows the user from one room to another one, are realized seamlessly.

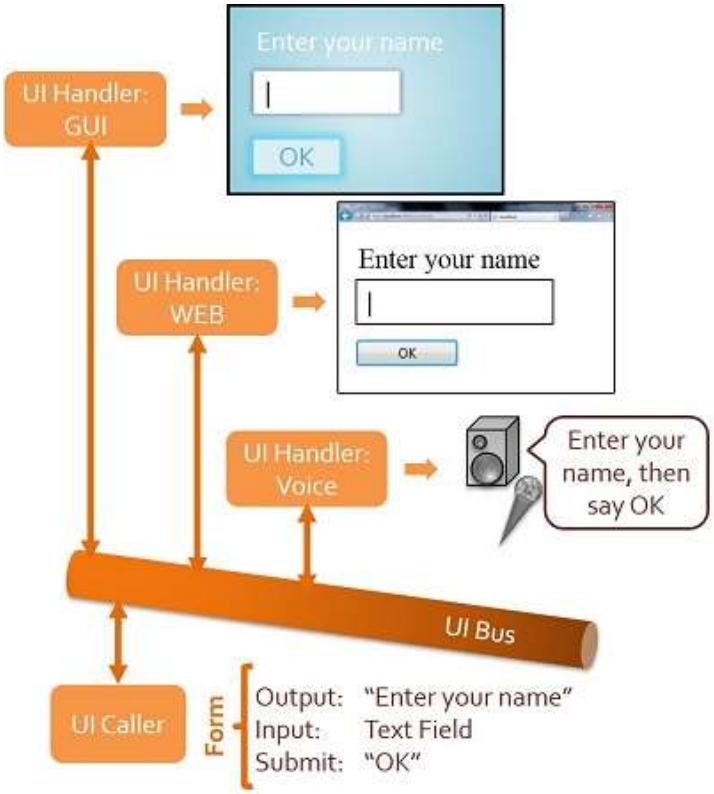

Figure 10. The UI bus with different UI Handlers each with a different modality and / or layout policy, all showing how the rendering would be performed for the same interaction request.

Openness[5]

Open API	https://github.com/universAAL/platform/wiki
Open Scope	universAAL has been developed in the context of European projects on "ICT for ageing well", as a platform for Ambient Assisted Living (AAL). Therefore, most of the applications developed so far on top of universAAL aim at supporting active and independent living of older people in smart living environments.

[5] See http://www.universaal.info/blog/post/3487/What-is-anopen-platform/ for more information.

	Nevertheless, universAAL does support the development of several other types of applications beyond AAL. This claim is based on the following facts: – The lower part of universAAL known as the universAAL middleware is an enabler for the integration of distributed components and communication among them while hiding distribution and heterogeneity. The use cases supported by the universAAL middleware are mostly about sending and receiving messages, enabling semantic interoperability without any domain-specific bias. This characteristic makes the universAAL middleware appropriate for the integration of any open distributed system of systems. – Beyond the general support for communication with global applicability, there is one specific feature of the universAAL middleware that is actually restricted to smart environments because it is concentrated on explicit interaction among human users and smart environments in a situation-aware way. However, it is still without any application bias, only based on an abstract representation of users, locations, and "interaction channels". The usage of this feature is totally optional. – Beyond the middleware, there is a set of optional universAAL "manager" components, most of which are again totally without application bias. This has been possible due to the reliance on the Semantic Web technologies for unified representation of data regardless of domain and extraction technology, unified cross-domain query language, and externalizable and sharable domain models. Only the profiling solution relies on specific ontologies that model the physical world and users in a specific way. The modular design and implementation of the universAAL architecture, however, allows to substitute this profiling solution with alternative solutions relying on alternative models.
Open Source license	https://www.apache.org/licenses/LICENSE-2.0
Open Provision	The licensing with Apache License 2.0 allows any form of redistribution, also when bundled with a host platform in terms of hardware and / or operating system.
Open Operation	The licensing with Apache License 2.0 introduces no restriction in using universAAL in operation in any environment.
Open adaptation	1. The licensing with Apache License 2.0 introduces no restriction in changing the source code. 2. The modular implementation with clear dependency specification allows to bypass or substitute not needed / desired modules without affecting the operation of the modules in use as long as the dependencies of the used modules are satisfied.

Maturity

TRL 1 – basic principles observed	Done 1999 – 2006 in the German projects EMBASSI and DynAMITE
TRL 2 – technology concept formulated	
TRL 3 – experimental proof of concept	Achieved 2007-2013 in the FP6 and FP7 projects PERSONA and universAAL
TRL 4 – technology validated in lab	
TRL 5 – technology validated in relevant environment	
TRL 6 – technology demonstrated in relevant environment	Achieved 2013-2016 in the FP7 CIP project ReAAL in 13 pilot sites from eight different EU countries with 31 different applications providing 100+ use cases to 5000+ users
TRL 7 – system prototype demonstration in operational environment	
TRL 8 – system complete and qualified	Complete deployments qualified in few sites of ReAAL
TRL 9 – actual system proven in operational environment	A core set of universAAL modules is since 2 years in operation in real life business, without interruption, seven days a week, 24 hours a day

Status w.r.t. major Requirements

Semantic Interoperability	Worldwide unique support for semantic interoperability at the level of communication protocols in a service-oriented environment, by eliminating all syntactic dependencies between communicating parties, through limiting all syntactical dependencies to the standard API of a brokerage layer that provides three virtual communication buses: the context bus for event-based communication, the service bus for call-based communication, and the UI bus as a special case of the service bus where the service providers are able to handle the interaction with human users.
Security / Privacy	The execution environment of universAAL is based on Java with OSGi or Android. In addition to benefiting from the native security mechanisms of these technologies, such as signed archives / bundles, policy files, and a controlled way of resolving dependencies, there are the following levels of security introduced by universAAL itself: – Groups of universAAL nodes may connect to each other, only when they share the same symmetric key; within such a group (a uSpace), messages are encrypted and decrypted using the shared key of the group / uSpace, resulting in end-to-end security within the group / uSpace. – Communication between two groups (uSpaces) as well as between a group (a uSpace) and a non-universAAL component has to pass the uSpace gateway, where end-to-end security can be achieved based on standard public key infrastructure. – Permissions are definable at the level of any type of message on any of the three communication buses as well as on the uSpace gateways. – Identifiers are all based on the concept of URIs exchanged within the trusted environment of universAAL as described above. – For the specific case of smart environments, a location-based concept for sessions, invalidated by contextual knowledge, minimizes the frequency of explicit authentication of human users. – Privacy can be preserved not only by having the above controls on the exchange of data and functionality but also due to the fact that the design and implementation of universAAL is not bound to any predefined deployment strategy or operation policy governed by predefined roles, such as Could servers or backend systems, but can be organized according to the requirements in several different ways.
Developer tools	In addition to benefiting from the eclipse and maven plugin mechanisms, eclipse update sites and maven repositories, there are a few developer tools, such as a dashboard, project templates with associated code generators, ontology modelling tools with code generating, integrated tests with eclipse debugging, and continuous integration.
Deployment tools	The few existing tools for installation, commissioning, administration, customization and personalization tools are all proprietary and mostly immature.
Data analysis tools	Data is stored in terms of RDF databases, using Virtuoso and Sesame. There are two immature context reasoning tools which use rules defined for them to derive new facts from known facts; one is based on the power of first order logic provided by the SPARQL Construct queries, and the other based on java programming within a Drools engine. There is no pattern recognition tool yet. There are no real visual analytics tools, except of a proprietary basic tool for visualizing the history of data from single sources.
Marketplace	During the FP7 project universAAL a component called uStore was developed using the IBM WebSphere as the eCommerce engine. After the end of that project, the maintenance of uStore was stopped due to two major issues: on one side, the lack of either real demand or any operation plan for uStore, and on the other side, the proprietary licensing of WebSphere.