Contract No. IST 2005-034891

Hydra

Networked Embedded System middleware for Heterogeneous physical devices in a distributed architecture

D6.6 Updated MDA Design Document

Integrated Project
SO 2.5.3 Embedded systems

Project start date: 1st July 2006 **Duration: 48 months**

Published by the Hydra Consortium **2009-08-20 version 1.0**
Coordinating Partner: Fraunhofer FIT

Project co-funded by the European Commission
within the Sixth Framework Programme (2002 -2006)

Dissemination Level: Public

Document file: D6.6 Updated MDA Design Document v1.0.doc

Work package: WP6

Tasks: T – 6.6

Document owner: CNet

Document history:

Version	Author(s)	Date	Changes made
0.10	Matts Ahlsén (CNET)	2009-05-07	Document outline, TOC
0.2	Matts Ahlsén, Peter Rosengren, Peeter Kool (CNET)	2009-0515	Introduction & overview
0.3	Matts Ahlsén, Peter Rosengren, Peeter Kool (CNET)	2009-05-20	MDA in Hydra overview
0.35	Matts Ahlsén, Peter Rosengren, Peeter Kool (CNET)	2009-06-10	Semantic device descriptions
0.50	Matts Ahlsén, Peter Rosengren, Peeter Kool (CNET)	2009-06-15	MDA Middleware components update
0.70	Peter Kostelnik (TUC), Matts Ahlsén (CNET)	2009-06-20	Device Ontology update
0.8	Matts Ahlsén, Peter Rosengren, Peeter Kool (CNET)	2009-06-23	Discovery process update
0.95	Matts Ahlsén, Peter Rosengren, Peeter Kool (CNET)	2009-06-30	Draft 1
0.96	Peter Kostelnik (TUC), Matts Ahlsen (CNET)	2009-07-16	Revision of ontology chapter
0. 97	Matts Ahlsén, Peter Rosengren, Peeter Kool (CNET)	2009-08-04	Draft for peer-review
1.0	Matts Ahlsén, Peter Rosengren, Peeter Kool (CNET), Peter Kostelnik (TUC)	2009-08-20	Final for submission

Internal review history:

Reviewed by	Date	Comments
Julian Schütte (SIT)	2009-08-15	Approved with comments
Adedayo Adetoye (UR)	2009-08-15	Approved with comments

Index:

Figures:

1. Introduction

1.1 Background

The Hydra project aims to research, develop, and validate middleware for networked embedded systems that allows developers to develop cost-effective, high-performance ambient intelligence applications for heterogeneous physical devices.

The first objective is to develop middleware based on a Service-oriented Architecture (SoA), which makes the underlying communication layers transparent to the applications built on top of the middleware. The middleware will include support for distributed as well as centralised architectures, security and trust, reflective properties and model-driven development of applications.

The Hydra middleware will be deployable on both new and existing networks of distributed wireless and wired devices, which operate with limited resources in terms of computing power, energy and memory usage. It will allow for secure, trustworthy, and fault tolerant applications through the use of novel distributed security and social trust components.

The embedded and mobile Service-oriented Architecture will provide interoperable access to data, information and knowledge across heterogeneous platforms, including web services, and support true ambient intelligence for ubiquitous networked devices.

The second objective of the Hydra project is to develop an Integrated Development Environment (IDE). The IDE will be used by developers to develop innovative semantic model driven applications with embedded ambient intelligence using the Hydra middleware.

1.2 Purpose, context and scope of this deliverable

This document (D6.6) describes the Semantic Model Driven Architecture (MDA) of Hydra as it has evolved from the initial specifications and design until the implementation in the middleware, as per end of iteration 3 in month 36.

This document is an update of the initial MDA design document (D6.2) with inputs from the intermediate software deliverables D6.4 to D6.8.

Hydra aims to interconnect devices, people, terminals, buildings, etc. The Service-Oriented Architecture and its related standards provide interoperability at a syntactic level. However, in Hydra we also aim at providing interoperability at a *semantic level*. The objective of WP6 is to extend this syntactic interoperability to the application level, i.e., in terms of semantic interoperability. This is done by combining the use of ontologies with semantic web services.

In order to cope with the huge variety of capabilities of the devices to be integrated in Hydra, the Hydra middleware should provide adaptations to whatever interface the devices offer. To achieve this, Hydra aims to be able to describe the capabilities of the devices in such way that a software agent can understand these capabilities and use them, e.g., in an automated discovery process.

The objective of the Hydra MDA is to facilitate application development and to promote semantic interoperability for services and devices. The semantic MDA of Hydra includes a set of models, represented by ontologies, and enables these to be used both in design-time and in run-time.

1.3 Hydra Innovations and Contributions

Hydra's technological innovations in semantic MDA are in the following areas:

- To develop tools for (semi-)automatic building of device ontologies - evolving ontologies, generalisation of concepts (knowledge generalisation)
- Techniques for automatic device classification and ontology updating.
- Ontologies over the middleware components themselves.

- Application of ontology-based semantic technologies on privacy and security issues
- Application of ontologies in enabling intelligent services (personalisation, alerting etc.) and search.

The following highlighted extract from table 5 in the DOW section 4.5 "Technologies to be used, researched and developed" summarizes the intended contributions from WP6 with respect to the semantic model-driven architecture.

WP 6 SoA and MDA middleware

Technology area	Use of existing technologies	New technologies to be developed	New technologies to be researched
Embedded and mobile service-oriented architectures for AmI	The Hydra middleware will be based on mature web service technologies such as SOA, SOAP, WSDL, BPEL etc. to the furthest extend possible Embedded web services will be built using standard WS technologies including: • Web services stack • Fast evaluation of WS • Semantic stack	Technologies for bringing semantic web service technology down to device level to provide semantic interoperability between devices.	New technologies for integration of WS with the device level will be researched. This will include: • Automatic generation of web services device proxies. • Caching principles
Semantic Model-Driven Architecture for AmI	The model driven architecture will be build with standard web service technologies including domain model meta descriptors such as IFC and HL7 classes Ontology frameworks will be based on standards such as OWL Horizontal standards such as WS-Coordination and WS-Transaction will be considered	New technologies for maintaining and accessing distributed domain meta models will be developed Semantic cooperative instantiation of devices, personas and services will be developed	Technologies for Automatic Device classification Technologies for Semantic-cooperative reasoning. New techniques based on combination UML and OWL for automatic construction and maintenance of ontologies will be researched. Research of principles and technologies for Intelligent Rules Processing to allow for configuration of device behaviour.
Semantics and knowledge management	Prototype semantic approaches will be used, e.g., inspired by OWL-S or SWS based on the Semantic Web, to support properties such as discovery, context awareness, self-* properties Standard Knowledge Management (KM) techniques for knowledge capture, indexing and re-use will be deployed where needed and applicable	New technologies to provide interoperability at the semantic level will be developed including profiling knowledge repository technologies for preference engineering	

Table 1: WP6 contribution objectives from the initial DOW

1.3.1 OMG Model-Driven Architecture

The MDA represents a major evolutionary step in the way the OMG (The Object Management Group, www.omg.org) defines interoperability standards. For a very long time, interoperability had been based largely on CORBA standards and services. Heterogeneous software systems interoperate at the level of standard component interfaces. The MDA process, on the other hand, places formal

system models at the core of the interoperability problem. What is most significant about this approach in relation to Hydra is the independence of the system specification from the implementation technology or platform. The system definition exists independently of any implementation model and has formal mappings to many possible platform infrastructures (e.g., Java, XML, and SOAP).

The MDA has significant implications for the disciplines of Meta modelling and Adaptive Object Models (AOMs). Meta modelling is the primary activity in the specification, or modelling, of metadata. Interoperability in heterogeneous environments is ultimately achieved via shared metadata and the overall strategy for sharing and understanding metadata consists of the automated development, publishing, management, and interpretation of models. AOM technology provides dynamic system behaviour based on run-time interpretation of such models. Architectures based on AOMs are highly interoperable, easily extended at run-time, and completely dynamic in terms of their overall behavioural specifications (i.e., their range of behaviour is not bound by hard-coded logic).

The main contribution of Hydra will is in the use of ontologies both for the application developer and the device developer. For the latter we support a model-driven process at design time through the use of ontologies and semi-automatic code generation for devices. Ontologies are also an integral part of the run-time environment, i.e. program execution is based on the models and descriptions in the ontologies, providing an easy to configure and dynamic extensible middleware.

1.3.2 Automatic Device Classification and Ontology Design

In order to cope with the huge variety of capabilities of the devices to be integrated in Hydra, two broad options can be considered: a) to force every device to be compliant to some set of more or less flexible interfaces, or b) to have Hydra middleware layer provide adaptation to whatever interface the devices offer.

Since choice a) will probably not be applicable neither to the present nor to the future world, Hydra has opted for choice b), so it will try to be able to adapt to all the variety of interfaces, information and operations that the devices offer. And given the vast amount of devices, the only viable option to address this issue is to try to do it in some automatic way.

In order to achieve this, Hydra relies on semantic descriptions/annotations about device capabilities (using ontologies) in such way that applications can understand these capabilities and use them. Once the semantics describing the model of a peer device has been found, the device capabilities could be accessed.

1.3.3 Embedded device semantics and rule engines

A final issue, which involves the adoption of semantic facilities into a novel platform such as the envisaged one, comprises the development of reasoning rules and components that will make use of dynamic meta-data to take advanced real-time decisions. It is clear that web services composition is the technology envisaged obtaining complex functionality from atomic operations of heterogeneous end-points (services, interfaces provided by any entity: user agents, servers, devices, etc.). The reasoning over available data (not only services but also network status, context information, availability of resources, etc.) becomes a critical task that should be solved to obtain later successful compositions. However, reasoners must rely on query languages over meta-data and there are several initiatives and languages that allow for queries over RDF annotated data.

In the Hydra MDA the choice fell upon SPARQL as the query interface to the Device Ontology. The latter expressed on OWL/Owl-s, with the Pellet reasoning engine. For diagnostics purposes the SWRL language has been used to express diagnosis rules (c.f. WP4).

2. Executive Summary

This work package applies Service Oriented and Model Driven Architecture techniques to AmI systems. All of the devices and services comprising a Hydra network will be integrated in a *Service Oriented Architecture* (SoA), which will provide, among other things, interoperability. The Hydra middleware thus also becomes the link between web services and devices. Interoperability, which here is taken as the capability of components of Hydra to talk to each other no matter what is the technology used to implement them or their physical location, is achieved by means of the usage of many specifications in the context of the web services world, including XML, SOAP, WSDL, XML Schema, WS-Security, WS-Addressing and several others. To summarise, the main purpose of the Service-Oriented Architecture in Hydra is to provide interoperability between devices at a *syntactic level*.

Hydra aims to interconnect devices, people, terminals, buildings, etc. As mentioned above, the Service-Oriented Architecture and its related standards provide interoperability at a syntactic level. However, one of the goals of Hydra is to provide interoperability at the *semantic level*. This is achieved through a modelling infrastructure in the middleware, whereby services exposed by devices can be described and consumed by Hydra applications.

A main contribution of this work package to is that it brings semantic web technologies down to the device level, i.e., each device can act as a semantic web service accessible by other devices, users and software application. This is achieved in close cooperation with WP4 who have developed techniques for embedding web services into devices. In this WP we are concerned with automating the generation of web services code for devices based on meta data and ontology descriptions.

In order to cope with the huge variety of capabilities of the devices to be integrated in Hydra, two broad options can be considered: a) to force every device to be compliant to some set of more or less flexible interfaces, or b) to have Hydra middle layer provide adaptation to whatever interface the devices offer.

Since choice a) will probably not be applicable neither to the present nor to the future world, Hydra has opted for choice b), so that the middleware is able to adapt to the variety of interfaces, information and operations that devices offer. And given the vast amount of devices, the only viable option to address this issue is to try to do it in an automatic way.

In order to achieve this, Hydra has introduced descriptions for the devices (ontologies) in such way that an automatic agent can understand these capabilities and use them. Once the semantic description of the device mode has been found, then its device capabilities could be accessed.

Hydra's technological innovations in semantic MDA are in the following areas:

- To develop tools for (semi-)automatic building of device ontologies - evolving ontologies, generalisation of concepts (knowledge generalisation).
- Techniques for automatic device classification and ontology updating.
- Ontologies over the middleware components themselves.
- Application of ontology-based semantic technologies on privacy and security issues
- Application of ontologies in enabling intelligent services (personalisation, alerting etc.) and search.

A final issue, which involves the adoption of semantic facilities into a novel platform such as the envisaged one, comprises the development of reasoning rules and components that will make use of dynamic meta-data to take advanced real-time decisions. It is clear that web services composition is the technology envisaged obtaining complex functionality from atomic operations of heterogeneous end-points (services, interfaces provided by any entity: user agents, servers, devices, etc.).

3. Requirements for the Hydra Semantic Model-driven Architecture

3.1 User requirements

Below we present the current set of user requirements with relevance for the MDA, and which are part of the specification at the end of iteration 3. The requirements are maintained and elicitated using the Volere method in WP2 with the Jira requirements database A subset of the requirements are selected for assessment in validation (WP10) of each iterations prototype (c.f. D10.2 and D10.3 for details).

Table 2: WP6 requirements summary list

Key	Summary	Rationale	Fit Criteria
Hydra-91	Any Hydra device should have an associated description	For management, search and discovery purposes, all Hydra enabled devices should be described (classified) according to the Hydra device ontology.	Any device associated to a Hydra application is also included in the Hydra device ontology, and its description can be retrieved.
Hydra-92	Rule-based configuration of devices	The possibility for the developer to specify device behaviour using rules. It should be possible to derive and re-use rules from pre-existing or generic rule sets for application domains. Possibility to hide device specific details.	The functionality (services) of a device is accessible (by user or application) thru a rule-based interface.
Hydra-94	Simulation environment	Use of a simulation environment is important for validating the rules/software interaction with devices. It can also be used for replaying the event log in order to examine unwanted system behaviour.	Simulation environment is available
Hydra-101	Manual device ontology definition	The developer should be able to define and extend device ontologies. The IDE is required to provide descriptors for devices and device classes	The Hydra IDE supports the manual editing of devices in the framework of device ontology.
Hydra-102	Device Ontology with user interface	Tool that allows browsing, searching, navigating device classes and their capabilities.	Tool for browsing device ontology exists
Hydra-103	Automatic device ontology construction	The construction of a device ontology should be facilitated through finding and parsing product or device descriptions to annotate and produce ontology entries. The component should handle different input formats like Word, PDF, HTML, databases.	5 of 10 device descriptions can be successfully processed
Hydra-104	Automatic Discovery of Services	It should be possible to configure the middleware to discover available services that meets defined criteria.	8 of 10 services are automatically discovered.

Hydra-108	Device discovery	Middleware should be able to detect new device that enters the network	7 of 10 devices are discovered
Hydra-110	Device Categorisation in runtime	Middleware should after discovery of device be able to categorise a device based on device ontology information.	7 of 10 devices are correctly categorised and described.
Hydra-111	Dynamic Web Service Binding	Middleware should be able to after device discovery and categorisation expose a new device as a web service that can be called without re-compilation.	New devices are callable and controllable in 7 out of 10 cases.
Hydra-112	Dynamic Web Service Generation	Configuration tool that is able to generate the necessary interfaces to wrap the device functionality as a web service.	7 of 10 device functionalities are automatically represented as web services
Hydra-113	Composition (of services and devices)	In order to enhance or replace application level functions it will be useful to be able to compose services and devices from different providers and/or manufacturers into high level services/devices	Service composition during design-time is possible.
Hydra-114	Semantic enabling of device web services	Middleware should be able to attach semantic descriptions to device web services based on device ontology.	7 of 10 devices are semantically enabled.
Hydra-117	Hydra component ontology	In order to support automatic device proxy creation, a Hydra middleware manager's ontology is needed. The ontology will facilitate the selection of the appropriate device and service managers to implement the proxy, depending on the discovery protocol and device types.	Hydra device and service managers can be identified and selected through a software component ontology
Hydra-119	Domain modelling support	The middleware and IDE should be able to interface with application domain frameworks representing core concepts and functions of specific application domains. These could in the most basic form be represented by UML Profiles, or domain ontologies.	The Hydra IDE supports at min 2 defined domain modelling frameworks.
Hydra-120	Multiple Device Virtualisations	It should be possible to have several different views/virtualisations of a device depending on context and applications.	At least 2 different virtualisations are provided
Hydra-126	Automatic Device ontology updates	The device ontology should automatically update its device descriptions.	The device ontology can detect device updates and handle that in 7 of 10 cases.
Hydra-129	Support for Semantic Web Standards for Device	Middleware should support different semantic web standards, including OWL-S, WSMO, and selected parts of WS-*	Support for at least OWL-S and WSMO

	Communication		
Hydra-210	Middleware should support different architectural styles	It must be possible to build systems with different architectures such as fully decentralised vs. centralised. De/centralization can pertain to: - data/knowledge - control - computation	Supports at least two different architecture styles
Hydra-376	Security requirements must be part of the Hydra MDA	Security must be defined to be resolved semantically	Security model can be defined semantically
Hydra-389	Service browsing in device ontology	It must be possible to view services as central building blocks, thus an application developer should be able to browse the device ontology from a service perspective, in addition to a device perspective.	A developer can find services and use them in development, without an a priori knowledge of the devices that implement the services.
Hydra-392	Rules for selection of alternative devices	The developer user should be able to specify how devices can replace or complement each other. This is relevant in situations when a device fails and another device exists which can provide a replacement service, or, when different levels of quality of service are available.	In the SDK, contructs are available that allow the developer to specify rules for when and how devices and services can be interchanged and combined.
Hydra-477	Device proxies should make use of available security features for "last mile" communication	If non-Hydra-enabled devices are communicate to the Hydra network by a proxy, security features of the protocol supported by the device should be used.	Device proxies must support WEP and WPA for Wi-Fi connections as well as Bluetooth authentication and encryption
Hydra-500	Semantic annotations of devices using SAWSDL	Device developers should via the DDK be able to produce (SAWSDL) annotations for devices, in order to facilitate device discovery and ontology update.	For a given UPnP discoverable device, it is possible to create an SAWSDL annotation which can be accessed from the UPnP discovery information.
Hydra-501	A Hydra enabled device must support UPnP discovery	UPnP has been proven as a well-functioning network discovery mechanism in Hydra.	All Hydra enables devices support UPnP

3.2 Quality attributes scenarios

As a complement to the Volere requirements process, a set of Quality Attribute Scenarios were also developed. These are based on a number of ISO Quality Attributes that can be used to characterize different architecture qualities of the Hydra architecture (e.g., portability, adaptability). The Quality Scenarios relate some of the Volere requirements to the corresponding Quality attributes, by describing how particular quality attribute can be identified in the system architecture and possibly also measured. These results were reported in deliverable D6.1 [Hydra, 2007].

4. Hydra approach to Semantic MDA

4.1 Motivation: Applications with heterogeneous physical device networks

Hydra applications are based on networks of embedded devices, which may be geographically distributed and possibly heterogeneous in the technologies supported.

The Hydra MDA supports the design and run-time of such applications by providing a set of models, transformations and component assemblies.

The objectives are to facilitate programming with devices for application developers to thru the Hydra SDK, and for device manufacturers to Hydra enable physical devices through the Hydra DDK. The semantic model-driven architecture of Hydra is based on combination of ontologies and other semantic web technologies to support the design of applications of device networks in different application domains. The MDA is both a design-time and a run-time resource.

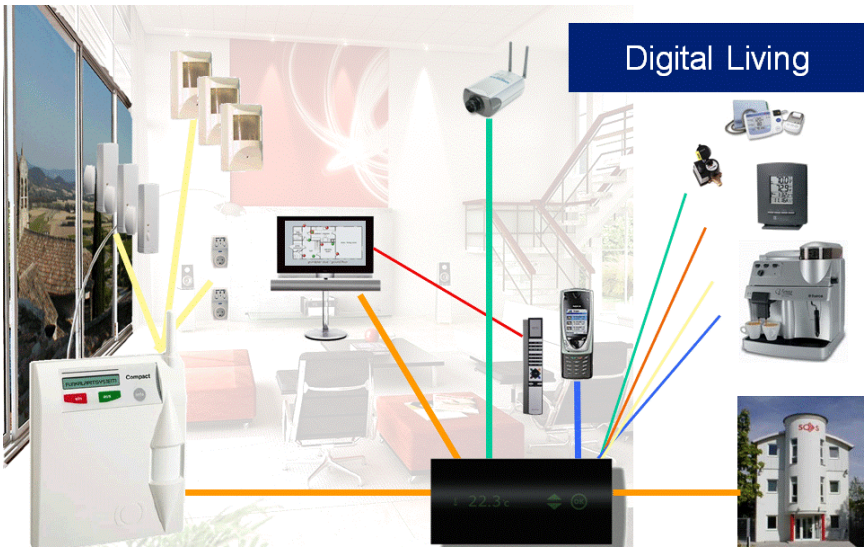


Figure 1: Device networks for home automation may include large numbers of heterogeneous devices.

Even though Hydra is a middleware and the MDA is a part of the middleware, the platform is tested in three different application domains.

Various types of devices usable in these domains have been the sources for requirements on the semantic representations of device and service descriptions and development support and tools for programmers.

Recent developments in home automation has resulted in new types of home appliances that are DLNA-compatible (a further development of UPnP), current mainly various media management devices. These devices should be able to coexist in a Hydra network, with other types of sensors and actuators based on various wireless technologies like ZigBee, Bluetooth and RF, which are supported by Hydra.

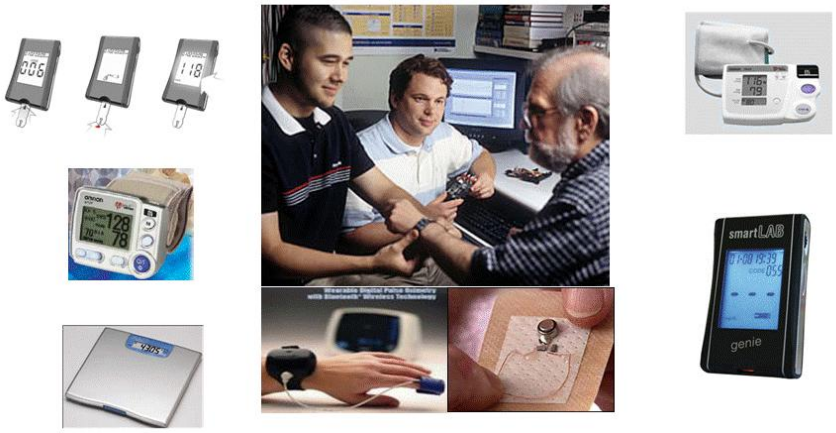


Figure 2: Devices for home health-care and remote monitoring

In the areas of health and agriculture, Hydra is supporting the use of various monitoring and sensor devices, foot vital signs monitoring and environment sensors.

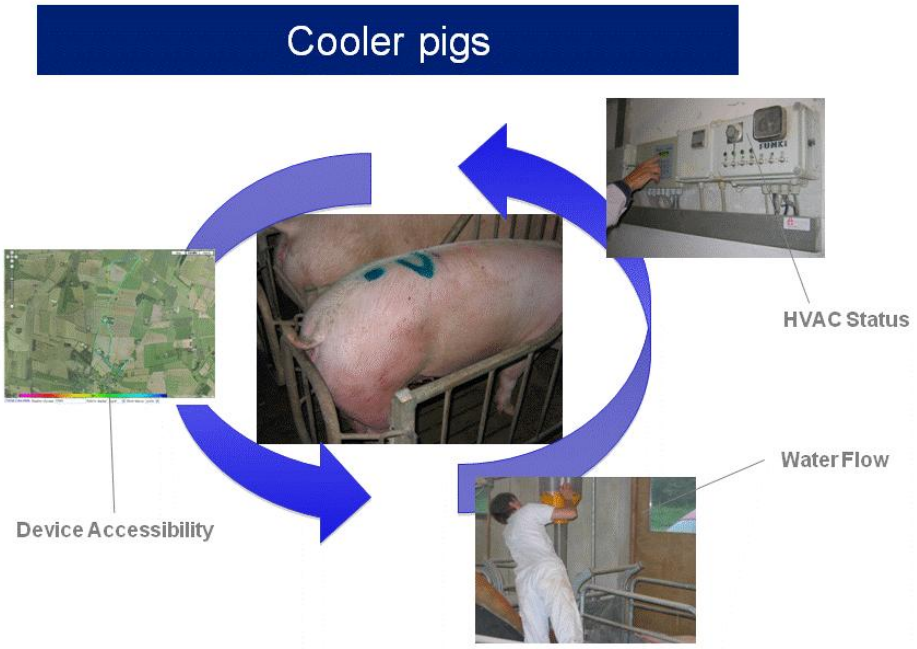


Figure 3: Agriculture and farming is another domain for Hydra applications

In order to connect these different devices, Hydra implements the necessary network platform infrastructure (based on SOA over P2P). On top of this platform the MDA provides the tools and mechanisms allowing programmers to develop model driven applications with the above mentioned device types.

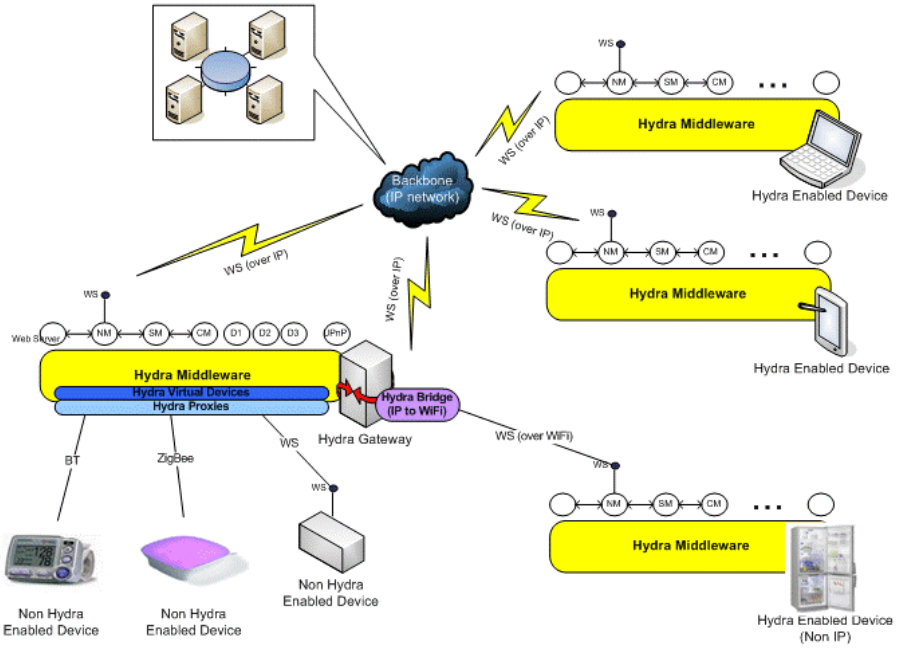


Figure 4: The Hydra middleware implementing Hydra Device networks

To achieve our vision of a Semantic Model Driven Architecture we have chosen to base our approach on ontologies and related semantic technologies. In Hydra there are three major ontologies used - Device Ontology, Security Ontology and Software Components Ontology.

The Hydra Device Ontology presents the basic high level concepts describing basic device related information, which will be used in both development and run-time process.

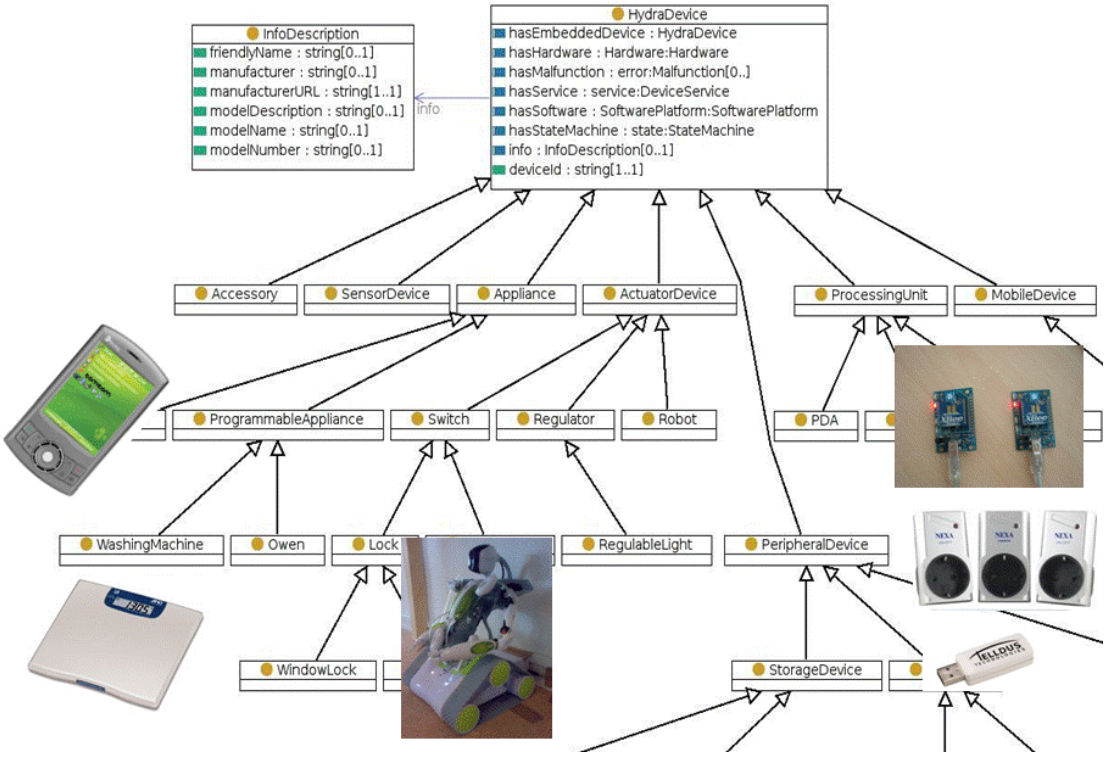


Figure 5: Device taxonomy subset of the Hydra Device Ontology

The device ontology is divided into four interconnected modules:

- Basic device information and taxonomy
- Device services
- Device malfunctions
- Device SW and HW capabilities and state machine

The content and structure of the Device Ontology is described in more detail in chapter 5.

To summarise, there are two uses of the semantic MDA in Hydra is used in two phases. Firstly, it is relevant at design-time, and it will support both device developers as well as application developers. Secondly, at run-time any Hydra application is driven from the semantic MDA.

4.2 Physical Devices and Hydra Devices

The basic idea behind the Hydra Semantic MDA is to differentiate between the physical devices and the application´s view of the device.

A Hydra Device is the software representation of a physical device. This representation is either implemented by a proxy running on a gateway device, or, by embedded Hydra managers on the actual device. A Hydra Device is said to Hydra-enable a physical device.

The MDA run-time includes a Device Service Generator which creates the service interfaces for discovered devices. Each Hydra device will thus get a web service as well as a UPnP service interface.

There are five categories of Web (UPnP) services generated for a Hydra Device,

- A Generic Hydra web service, exposing metadata and management functions common to all Hydra Devices
- An Energy web service, providing a set of functions for the monitoring and control of energy consumption of devices.
- A Memory Service which allows logging and storing of device internal data such as state variables and energy consumption data.
- A Location Service which can be used to query the device about is location and position.
- A device type specific web service, exposing the device type specific functions

4.3 Semantic Devices

Based on Hydra Devices, we introduce the concept of *Semantic Devices* as a programming construct. This allows a programmer to develop new applications specific adaptations of the available Hydra Devices.

The Hydra Devices offers a set of services, a lamp might offer "on/off" and "dimming" as two services while a pump might offer "increase flow" and "get water temperature" as two services.

The services offered by the physical devices have been designed independently of particular applications in which the device might be used. A semantic device on the other hand represents what the particular application would like to have. For instance, when we are designing the lighting system for a building it would be more appropriate to model the application as working with a logical lighting system that provides services like "working light", "presentation light", and "comfort light" rather than working with a set of independent lamps that can be turned on/off. These logical devices might in fact consist of aggregates of physical devices, and use different devices to deliver the service depending on the situation. The service "Working light" might be achieved during daytime by pulling up the blind (if it is down) and during evening by turning of a lamp (blind and lamp being Hydra Devices). We call these logical aggregates of devices and their services for *Semantic Devices*.

Semantic Devices should be seen as a programming concept. The application programmer designs and programs his application using semantic devices. Figure 6 below illustrates the concept. The

semantic device "Heating System" consist of three physical devices: a pump that circulates the water, a thermometer that delivers the temperature and a light that flashes when something is wrong.

The developer will only have to use the services offered by the semantic device "Heating System", for instance "Keep temperature:20 degrees Celsius" and "Set warning level:17 degrees Celsius", and does not need to know the underlying implementation of this particular heating system.

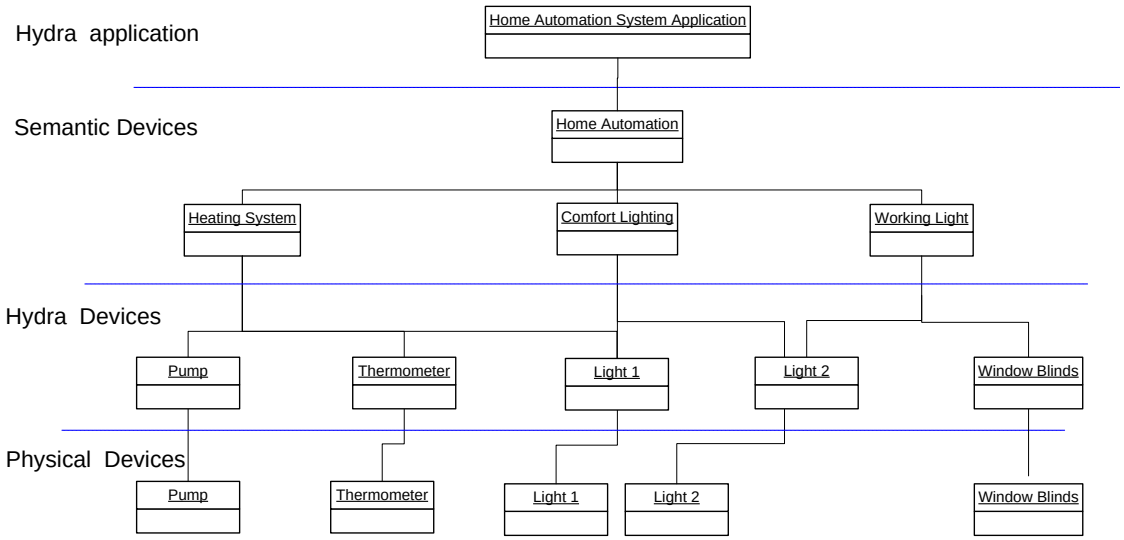


Figure 6: Semantic Devices provide a high-level programming construct on top of Hydra Devices.

The Semantic Device concept is flexible and will support both static mappings as well as dynamic mappings to physical devices.

Static mappings can be both 1-to-1 from a semantic device to a physical device or mappings that allow composition.

- An example of a 1-to-1 mapping would be a "semantic pump" that is exposed with all its services to the programmer.
- An example of a composed mapping is a semantic heating system that is mapped to three different underlying devices – a pump, a thermometer and a digital lamp.

Static mappings will require knowledge about which devices exists in the runtime environment, for instance the heating system mentioned above will require the existence of the three underlying devices – pump, thermometer and lamp – in for instance a building.

Dynamic mappings will allow semantic devices to be instantiated at runtime. Consider the heating system above. We might define it as consisting of the following devices/services:

- a device that can circulate the water and increase its temperature
- a device that can measure and deliver temperature
- a device that can create an alarm/alert signal if temperature is out of range.

When such a device is entered into the runtime environment it will use service discovery to instantiate itself and it will query the physical devices it discovers as to which can provide the services/functions the semantic device requires. In this example the semantic device most probably starts by finding a circulation pump.

But then it might find two different thermometers which both claims they can measure temperature. The semantic device could then query about which of the thermometers can deliver the temperature in Celsius, with what resolution and how often. In this case it might only be one of the thermometers that meet the requirements. Finally the semantic device could search the network if there is a physical device that can be used to generate an alarm if the temperature drops below a threshold or increases to much. By some reasoning the semantic device can deduct that by flashing the lamp repeatedly it can generate an alarm signal, so the lamp is included as part of the semantic heating system.

The basic idea behind semantic devices is to hide all the underlying complexity of the mapping to, discovery of and access to physical devices. The programmer just uses it as a normal object in his application focusing on solving the application's problems rather then the intrinsic of the physical devices.

We note also that Semantic devices can be subject to device discovery, and will in that case be discovered as Hydra Devices by other Hydra applications.

4.4 Semantic MDA at design-time

4.4.1 Model-driven code generation for Hydra Devices

The different ontologies in the semantic MDA are used at design time to generate web service code for devices. This work is carried out as a part of WP 4 "Embedded AmI Architecture". While WP4 is concerned with generating small and efficient web service code that can be embedded into devices, WP6 is concerned with utilising these device web services by mapping semantic devices to them to provide programmers with a high level semantic interface to the devices. It should be noted that in both WP4 and WP6 the same Device Ontology is used to ensure maximum re-use and a truly semantic MDA approach. It is the responsibility of WP6 to define the structure and content of the Device Ontology, as is described in chapter 5.

The details of the Hydra approach to web service code generation for devices are described in Deliverable 4.2 [Hydra, 2007b]. This section thus briefly summarizes the approach.

The figure below shows the generation strategy for web services for devices. We have developed a tool, *Limbo*, which takes as inputs an interface description ("Provide WSDL file") and a semantic description of the device on which a web service should run ("Provide OWL description"). The interface description is assumed to be in the form of a WSDL file and the semantic description is a link to an OWL description of the device (part of the Device Ontology).

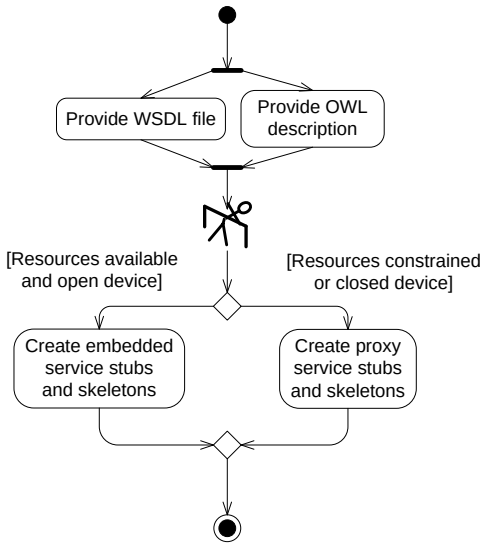


Figure 7: Automatic generation of web service code for devices.

The semantic description is used to

- *Determine the compilation target.* Depending on the available resources of a device, either embedded stubs and skeletons are created for the web service (to run on the target device) or proxy stubs and skeletons are created for the web service (to run on an OSGi gateway).
- *Provide support for reporting device status.* Based on a description of the device states at runtime (through a state machine), support code is generated for reporting state changes through the Hydra Event Manager. Eventually this also supports the self-* properties of Hydra.

In both cases, refer to D4.2/D4.7 "Embedded Service SDK/DDK Prototype and Report" for more detail.

Model-driven code generation for Semantic Devices

The descriptions of services in the Device Ontology can be used at design time to find suitable services for the application that the Hydra developer is working on. The descriptions of these services will be used to generate code to call the service, query the device that implements the service, and manipulate the data that the service operates on.

The Hydra SDK and DDK are made available in an object-oriented language environment, and IDEs. Thus, the objects a developer can use to access the services (service proxies) as well as objects from the Device ontology connected to the service (in its simplest form, the parameters to the service operations) will be generated from the Device Ontology and discovery information retrieved from the devices at discovery time.

These device objects can then be used when creating a semantic device or Hydra application from the selected Hydra devices and their services.

An example of this is a heating control system, where device proxies to represent the heating system devices and classes representing the domain classes (Temperature, TemperatureRange), will be generated for the Hydra developer.

Some devices have a certain set of services built in, e.g. a thermometer that provides a thermometer reading service. The thermometer service is not upgradeable and no other services can be added to the device. In this case we can find out which services the device provides by looking up the device in the ontology.

Some advanced devices such as smart phones and PDAs, however, are capable of installing and hosting any number of services. This is a capability of devices that will be represented in the Device Ontology. There are physical devices that come with a static set of services, devices that are programmable and thus can host (almost) any service and devices that can host Hydra proxies for physical devices. A Hydra developer can specify a service to be used, and leave the device as generic as possible - any device that is capable of implementing the service. The necessary code will be generated both for the service and the device.

Applications can decide how to use the Device Ontology, so that some applications will only use the Device Ontology at design whereas others will always query the Device Ontology for new types of services that match certain goals.

4.5 Semantic MDA at Run-time

4.5.1 Device Discovery Architecture

The Hydra MDA includes a Discovery Architecture which implements the device discovery process. This architecture is structured in three layers abstracting the discovery functions.

The discovery process operates both locally and remotely, so that devices that are discovered in a local Hydra network can also be discovered in a peer Hydra network over the P2P protocol implemented by the Hydra Network Manager.

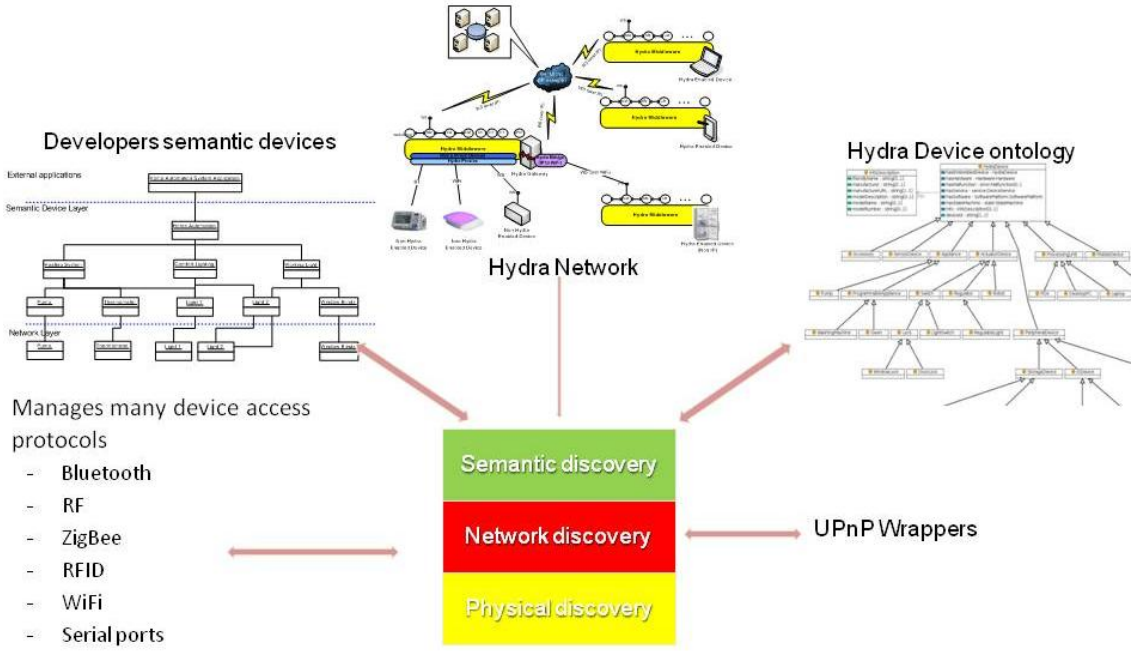


Figure 8: The 3-layered Discovery Architecture is part of the Hydra MDA.

The lowest discovery layer implements the protocol specific discovery of physical devices. This is performed by a set of specialized discovery managers listening for new devices at gateways in a Hydra network. The second layer uses UPnP/DLNA technology to announce discovered physical devices in the local network and to peer networks.

At the top most layer the device type is resolved against the Device Ontology and is mapped to some Hydra Device type. It is then placed in the *Device Application Catalogue* (DAC). If an application subscribes to events regarding this type of device, it will be notified that the device is available and has been placed in the Device Application Catalogue.

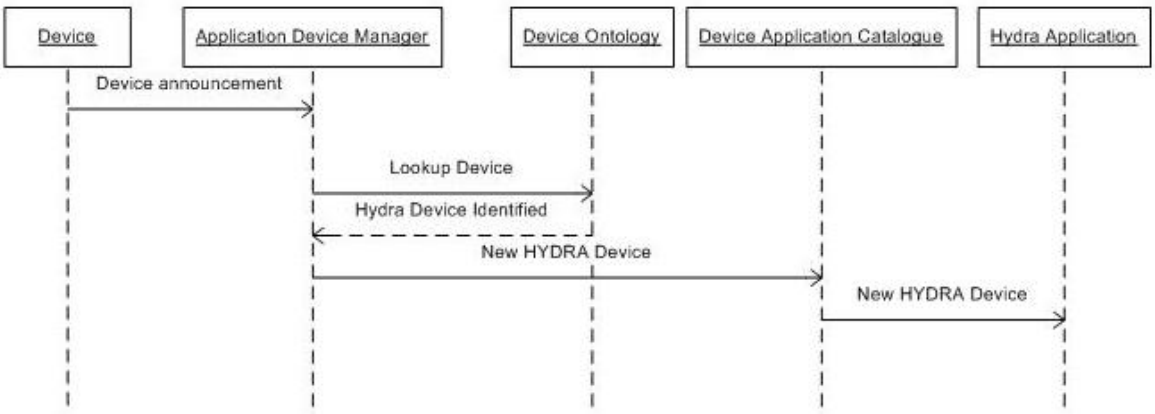


Figure 9: In the final step of discovery, the device type is resolved against the device ontology, and then entered into the DAC notifying the Hydra application.

4.5.2 The Device Application Catalogue (DAC)

The *Device Application Catalogue* (DAC) is a fundamental part in every Hydra application. It is a runtime component that keeps track of and manages all devices that are currently active within an

application. The DAC is managed by the Application Device Manager. The DAC serves all Hydra middleware managers with the information and metadata they need regarding devices, their services, and their status.

The Application Device Manager uses the Hydra Device Ontology and models for discovery to recognise new devices when they enter into a Hydra network. Based on the discovery model it queries the Device Ontology to deduce what type of device has entered the network. The Hydra DAC can be queried by different middleware managers to retrieve a service interface for different devices.

A Hydra browser has been developed to allow a user/developer to graphically browse the Hydra network and inspect properties and services of devices. The browser tool also allows the user to invoke the different services offered by devices. To illustrate the functionality of the Device Application Catalogue we can view the figure below that shows the *Hydra DAC Browser* which allows browsing of the different devices that have currently been discovered by the Hydra DAC.

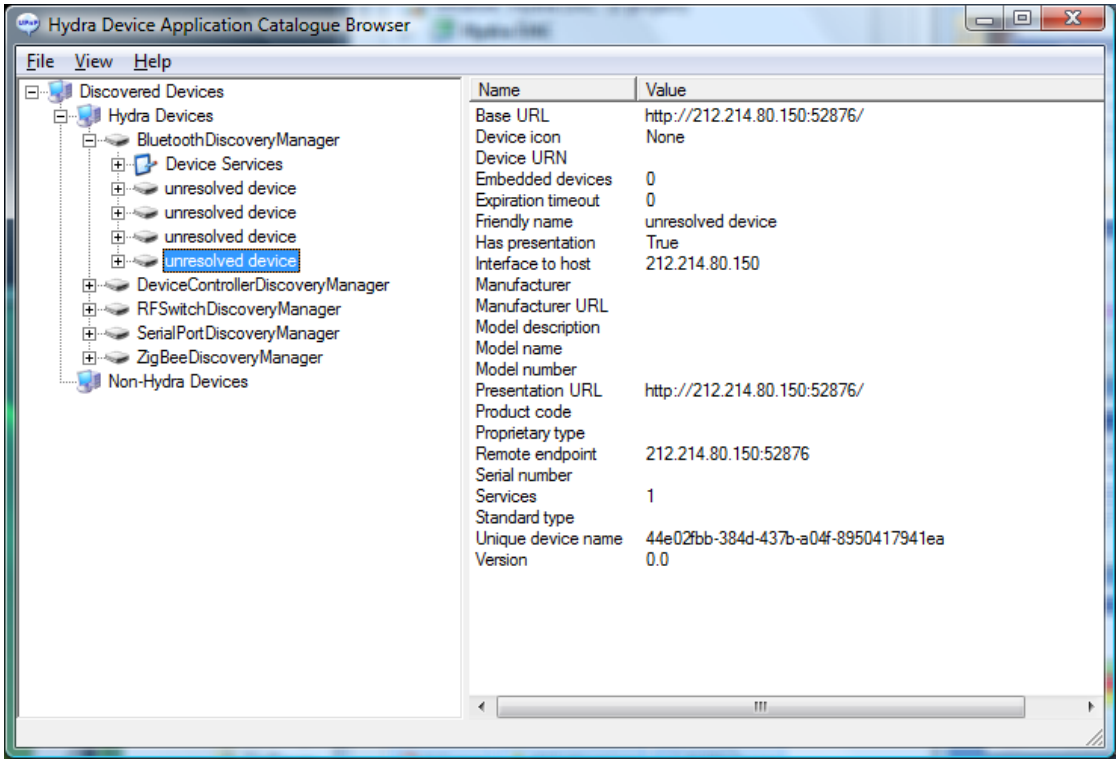


Figure 10: The Hydra Browser

By manually invoking the different services we can also actually illustrate the role the Device Application Catalogue plays in the Hydra middleware. As can be seen above 5 different Discovery Managers are available in the network, each of them is dedicated to discover a certain type of physical device (Bluetooth, RF Switches, and ZigBee etc).

Each Discovery Manager keeps track of the device it has discovered and tries to elicit as much information as possible from the device. All this physical discovery information can be accessed by calling the service "Get Device Physical Discovery".

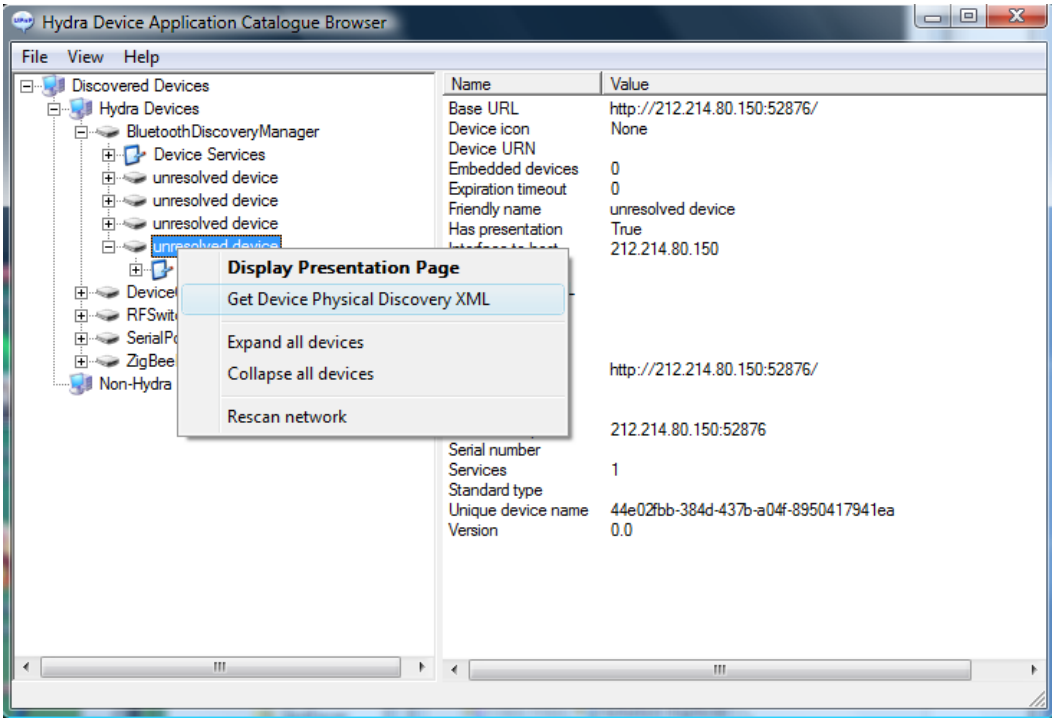


Figure 11: Retrieving discovery information from the physical device

This discovery information is returned as an XML document, which can be seen in the figure below:

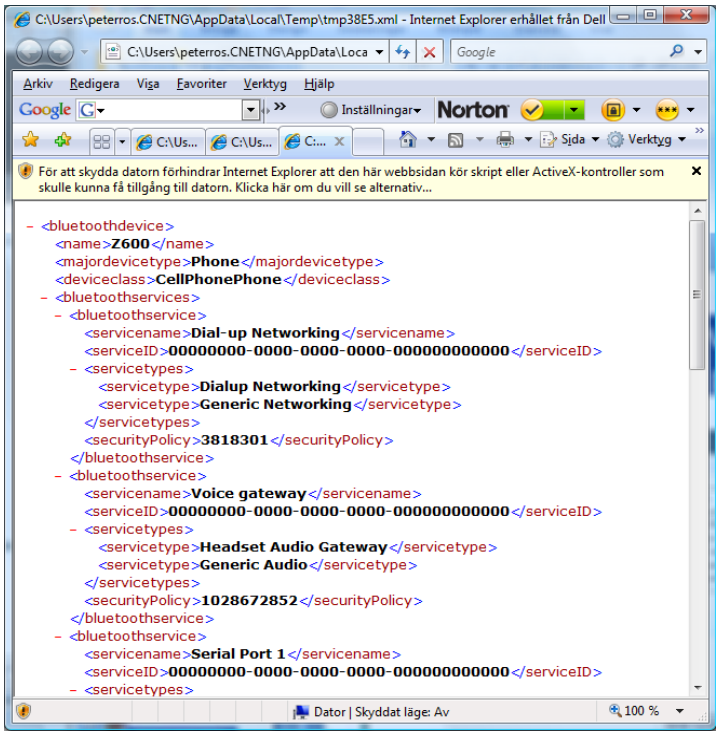


Figure 12: Discovery information from a Bluetooth Device

In the figure we can see that it is a Bluetooth Device that has been discovered, it has the Bluetooth Major DeviceType "Phone" and Minor DeviceType "CellPhonePhone" (Major DeviceType and Minor DeviceType are part of the Bluetooth standard.

The Bluetooth Discovery Manager has also managed to extract the different Bluetooth services offered by the device. This discovery information can now be used to reason about what type of device has been discovered. The physical discovery XML is given to the Device Ontology which deducts that this device corresponds to a "Basic Phone" in the Hydra Device Ontology.

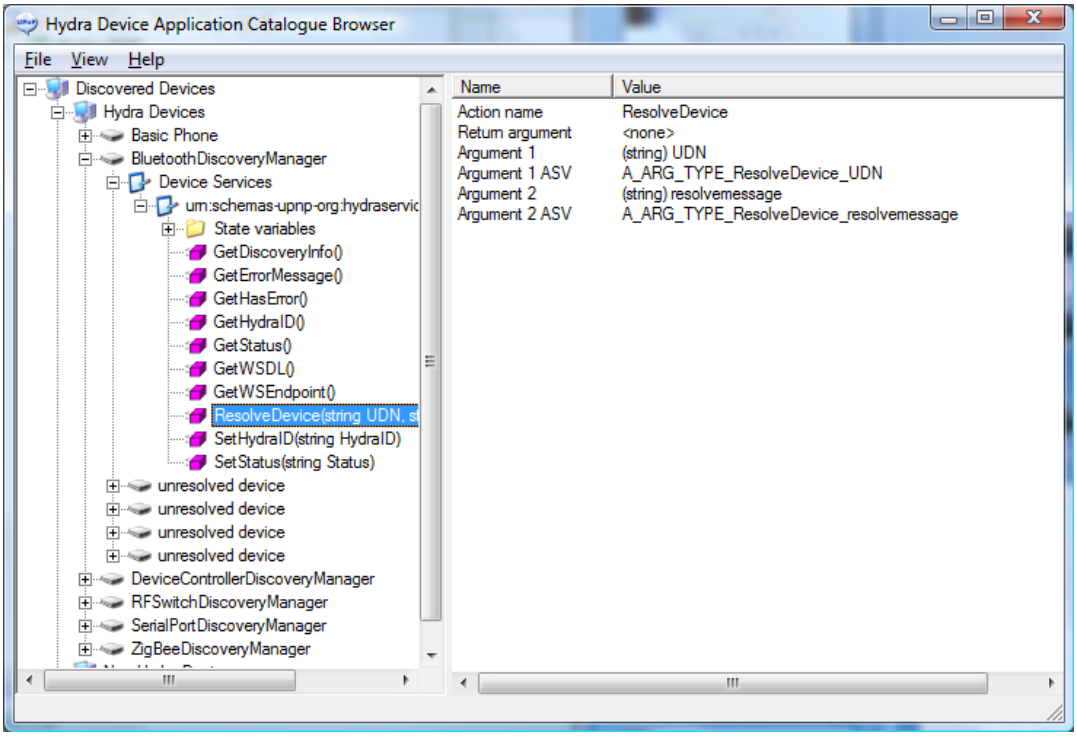


Figure 13: Resolving a physical device into a Hydra Device.

By invoking the service "Resolve Device" we can now tell the Bluetooth Discovery Manager that this is a "Basic Phone". The idea is of course to do this programmatically, but here we do it manually for illustration purposes.

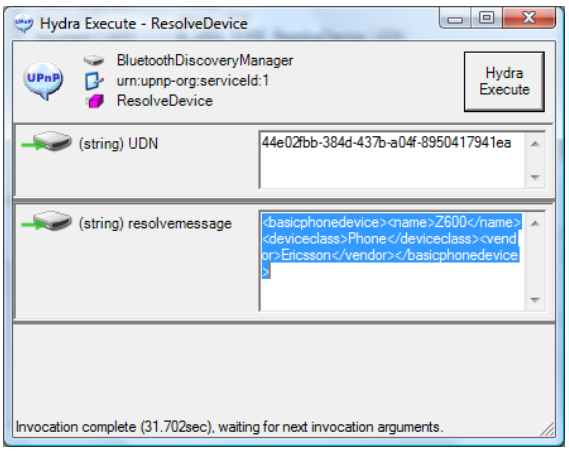


Figure 14: Resolve information is sent as an XML structure to the Discovery Manager

The Discovery Manager then creates and publishes the Device to the network as a "Basic Phone" device. The Basic Phone device is now available together with the services offered by a Basic Phone (in this case a set of SMS read/send functions).

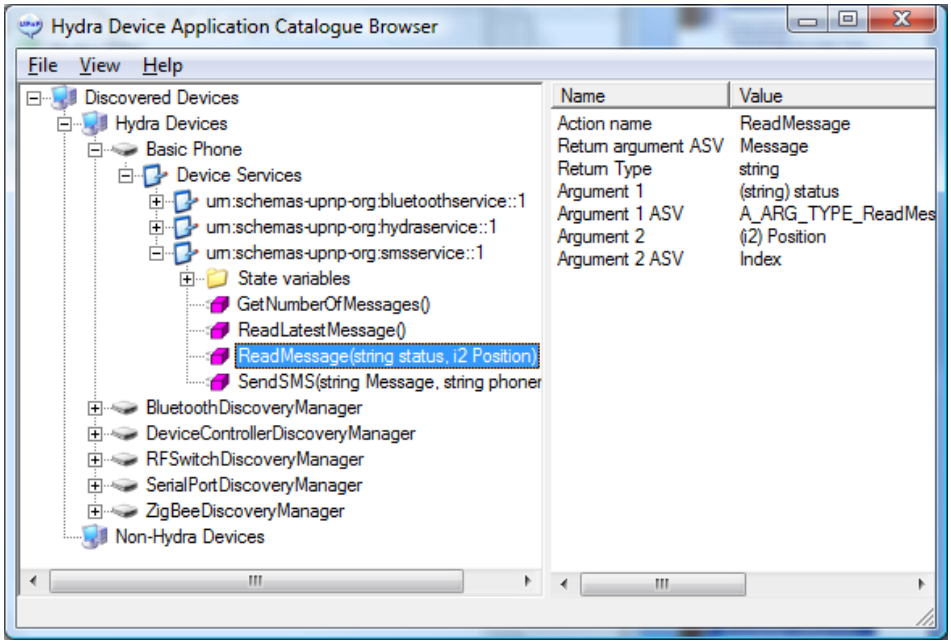


Figure 15: A physical device with unknown functionality has been transformed into Basic Phone Device with services for reading/sending SMS.

These services are now directly invokable from the Browser, and we can now for instance send an SMS.

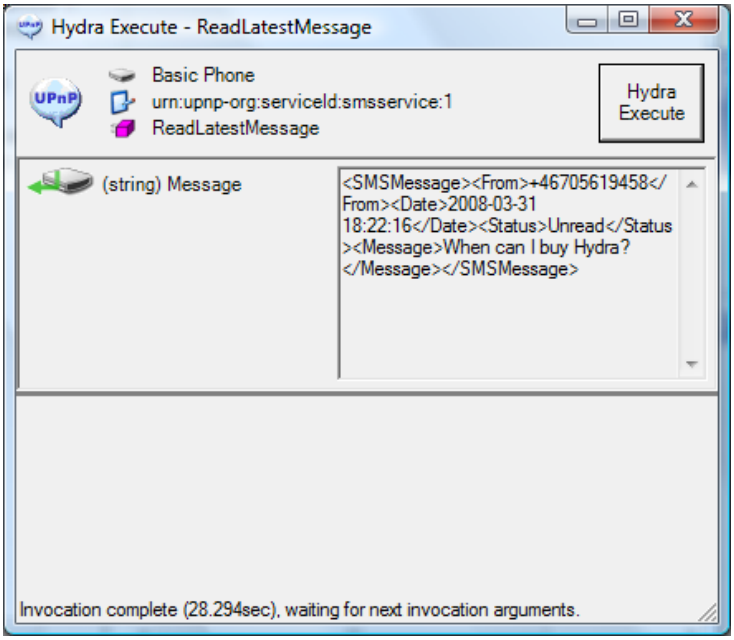


Figure 16: Sending an SMS through the Basic Phone Device

Finally we can also use the Browser to retrieve a service description for a web service that allows us to access the device programmatically:

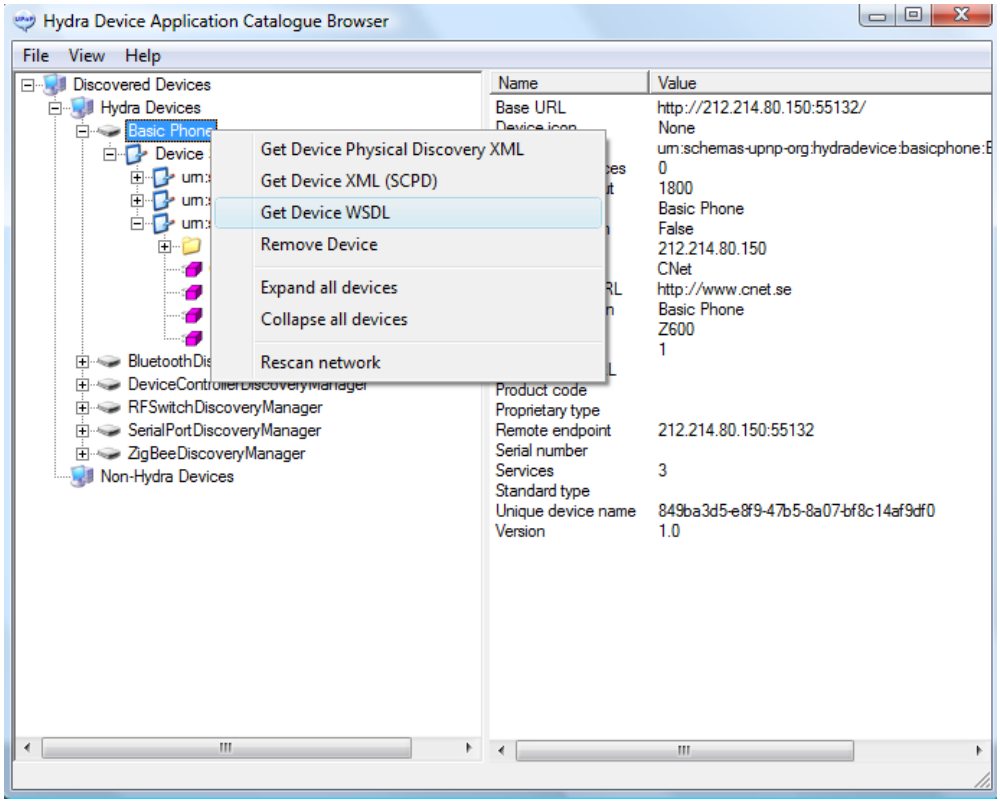


Figure 17: Using the DAC to retrieve a WSDL description for the device.

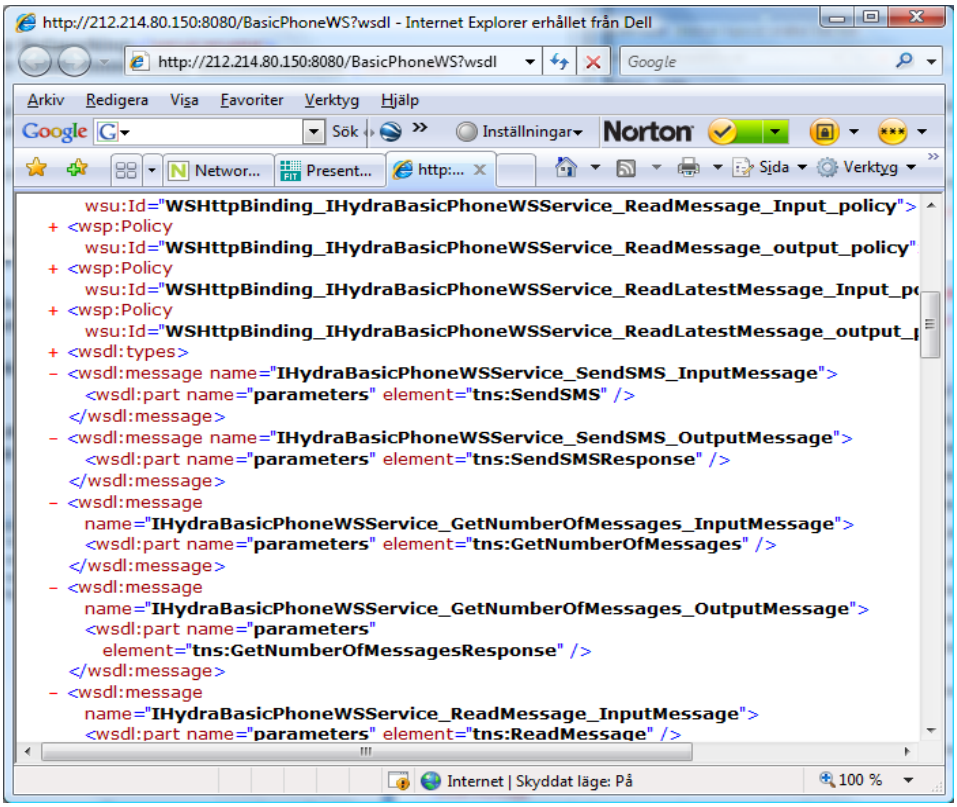


Figure 18: A WSDL (Web Service Description Language) for the device

The models used at design time are also used in the discovery of devices. At design time, the Hydra application developer selects the Hydra devices and services that will be used to implement the application. This subset of the Device Ontology will form the basis for the Device Application Catalogue. These devices may be defined at a fairly general level, e.g. the application may only be interested in "Hydra SMS Service" or "Hydra Generic Smartphone Device" and any device entering the network/(application context) that fits in these general categories will be presented to the application. The application will then work against the more general device descriptions.

This means that an application should only know of the (types of) devices and services selected by the developer when it was defined. Although other devices may be registered at the network level, an application gets notified on a "needs/wants to know" basis. Note that this still means that the application could use a device that was designed and built after the application was deployed, as long as the device can be classified through the Device Ontology as being of a device type or using a service that is known to the application, e.g., a Hydra application built in 2008 could specify the use of "Hydra Generic Smartphone" and "Hydra SMS Service" and thus use a "Nokia N2010 Smartphone" released two years later.

The above scenario means that although the Device Application Catalogue is defined at design time as a selection from the Device Ontology at a specific point in time, the Device Ontology used at runtime will be constantly updated. Whether the Ontology Manager always will use a full ontology or in some cases a subset that is useful to the application for optimization is to be further investigated. This will require solutions for versioning, caching and evolution of the Device Ontology.

If there are any non-Hydra-enabled devices that the developer wants to use, these will have to be Hydra enabled first using the (Hydra device mapping tools) e.g. LIMBO [Hydra, 2007b]. The Hydra developer will also have to define the application level events that are of interest to the application, e.g. devices entering or leaving the network, error states, and so on.

In the SDK, only Hydra Devices are used. If the developer needs information about the specific device at run time, this will be available on request (analogous to reflection capabilities in various programming languages), but in most cases, the only objects that the application handles are Hydra devices.

The DDK (Device Development Kit) is used to Hydra-enable physical devices, while the SDK (Software Development Kit) is used to build more advanced Hydra applications using other Hydra Devices.

4.5.3 Use of models for resolving security requirements and capabilities

The MDA also plays a role in the design and enforcement of security in Hydra. The modelling framework, requirements and design principles are based on the Hydra Security Meta Model, which elaborates the set of building blocks: context, trust, virtualisation and semantic resolution (deliverable D7.9).

A Security Ontology (reported in deliverables D7.6 and D7.7) is used to provide a vocabulary for protection goals and capabilities for Hydra Devices and applications. The ontology is a Hydra adaptation of an existing ontology framework (the NRL security ontology). The security ontology can be referenced from the devices descriptions in the Device Ontology.

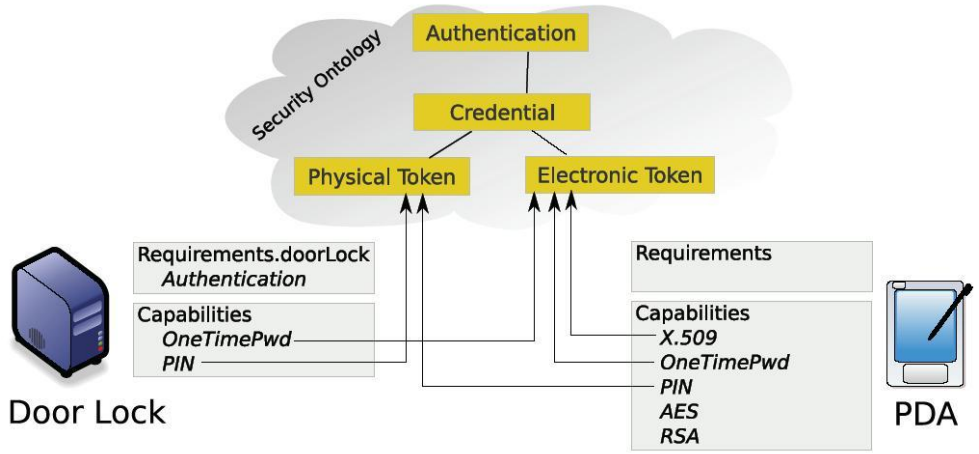


Figure 19: (from D7.9) Security Ontology can be used to resolve common security capabilities

We refer to the corresponding WP7 deliverables for in-depth descriptions of the Security Meta Model and its implementation in Hydra.

4.6 Standards used

An important objective in the design of Hydra as an open source middleware has been the use of standards, such as the various industry standards for semantic technologies. In the development process, three such language standards were used:

- The Web Ontology Language (OWL) allows semantic description of the several elements in the middleware environment. OWL was used as the main modelling language for ontology specification, capturing the most important requirements for achieving semantic interoperability in the Hydra
- The SPARQL query language for RDF was used to retrieve the information from ontologies in the development process to test the representation capabilities of developed models and also in Application Ontology Manager Implementation.
- The Semantic Web Rule Language (SWRL) allows definition of rules, which were used to extend the models of device state-machines. SWRL is primarily used in the devices diagnostics process.

A short overview of used standards and reasoners with their possibilities of usage in the Hydra is presented in the appendix.

5. Hydra ontologies

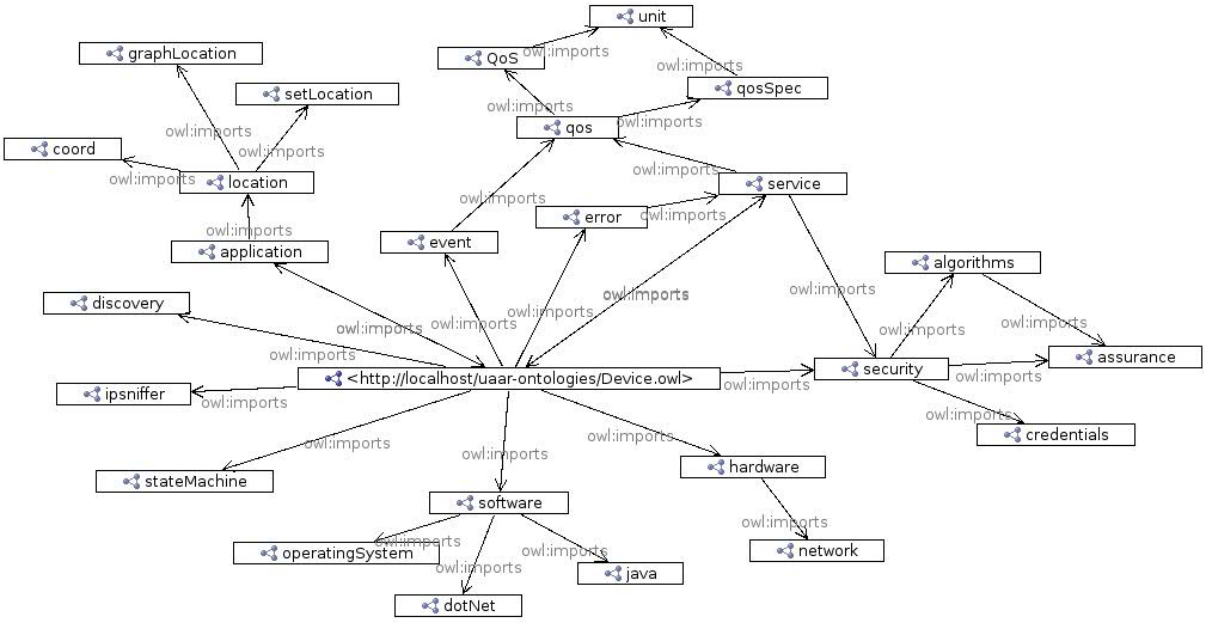
5.1 Hydra ontology architecture

In Hydra, ontologies are used to model devices, security requirements and parts of the middleware itself.

The Hydra Device Ontology represents the concepts describing device related information, which can be used in both design and run time. The basic ontology is composed of several partial models representing specific device information. The initial device ontology structure was extended from the FIPA device ontology specification [FIPA 2002]. The initial device taxonomy was extended from AMIGO project vocabularies for device descriptions [AMIGO, 2006].

The relation between the Device Ontology components is shown in

FiFigure 20.



i

Figure 20: Relations between main Device Ontology components.

The components of the Device Ontology can be shortly described as follows:

- Core Ontology (Device.owl): contains a taxonomy of various device types and the basic device description, manufacturer and model information.
- Device Capabilities: represent the hardware properties (Hardware.owl, Network.owl) and software description (SoftwarePlatform.owl divided into DotNet.owl, Java.owl and OperatingSystem.owl ontologies)
- Device Services (Service.owl): describes the models of device services in the terms of operation names, inputs and outputs. The device services are connected to the Quality of Service ontology (QoS.owl, QoS.owl, QoSSpec.owl, Unit.owl) used to annotate the services and their parameters to several quality factors.

The Hydra services of a device are further divided in different categories, which are made available to the developer in the DAC:

- A generic set of services providing access to various device and service metadata.
- A number of device type specific services.

- Device events (Event.owl): provides the descriptions of events, which can be generated by the simple devices, as the alternative of providing the functionality. Events can be annotated to the quality of service ontology in the similar way as the services.
- Device Malfunctions (Error.owl): represents the various types of errors and failures which may occur when using the device at run-time
- Self-* Properties supporting models: models of state-machines tracking the run-time device/service state changes, model of device run-time request/response tracking (IPSniffer.owl, StateMachine.owl) and SWRL rules supporting mainly the self-monitoring and self-diagnosis processes. Detailed in WP4, D4.8.
- Security Ontology (securityMain.owl): represents the various security properties, such as protocols, algorithms (securityAlgorithms.owl), objectives and assurances (securityAssurance.owl), which may be attached to devices or services. To describe the security properties, the third party NRL ontology was reused, modified and connected to the device model. Detailed in WP7, D7.3-D7.9.
- Discovery models (Discovery.owl): used for semantic resolution in the semantic discovery process.
- Application model (Application.owl): describes the model of application and the entities used in various applications, such as locations or persons (Location.owl, Coord.owl, SetLocation.owl, GraphLocation.owl)

The Hydra ontology architecture was designed to support the maintainability and future extensions of used concepts. The ontologies have been developed using the OWL language. The references between more general and specific concepts and modules (related ontologies) is realised using the OWL import mechanism. In design-time, every ontology module can be further extended by creating new concepts according to the needs of representation of the new information about new device types and models. The concepts can also be further specialized. For example, if the new device type is needed, the adequate concept in the device classification module can be further sub-classed by more specialised concepts and the new properties can be added.

The device or application developer may use the maintenance/update tools to extend or modify the ontology structure or for populating the ontology with instances – i.e., models of specific devices. Device instances created at design time are represented by description templates, which are used at run-time to create application specific device instances. Each time a new device is discovered, the ontology is used to infer the most suitable device template and the Ontology Manager creates the device specific clone – the run-time instance. Each real device has its own run-time instance, which is used to track the device properties continually changing at run-time. Tracking of run-time properties is used e.g. for self-monitoring, self-diagnosis, updating state-machines purposes or context-related computations. The ontology models can also be used as the semantic support for model-driven application development.

5.2 The Device Model

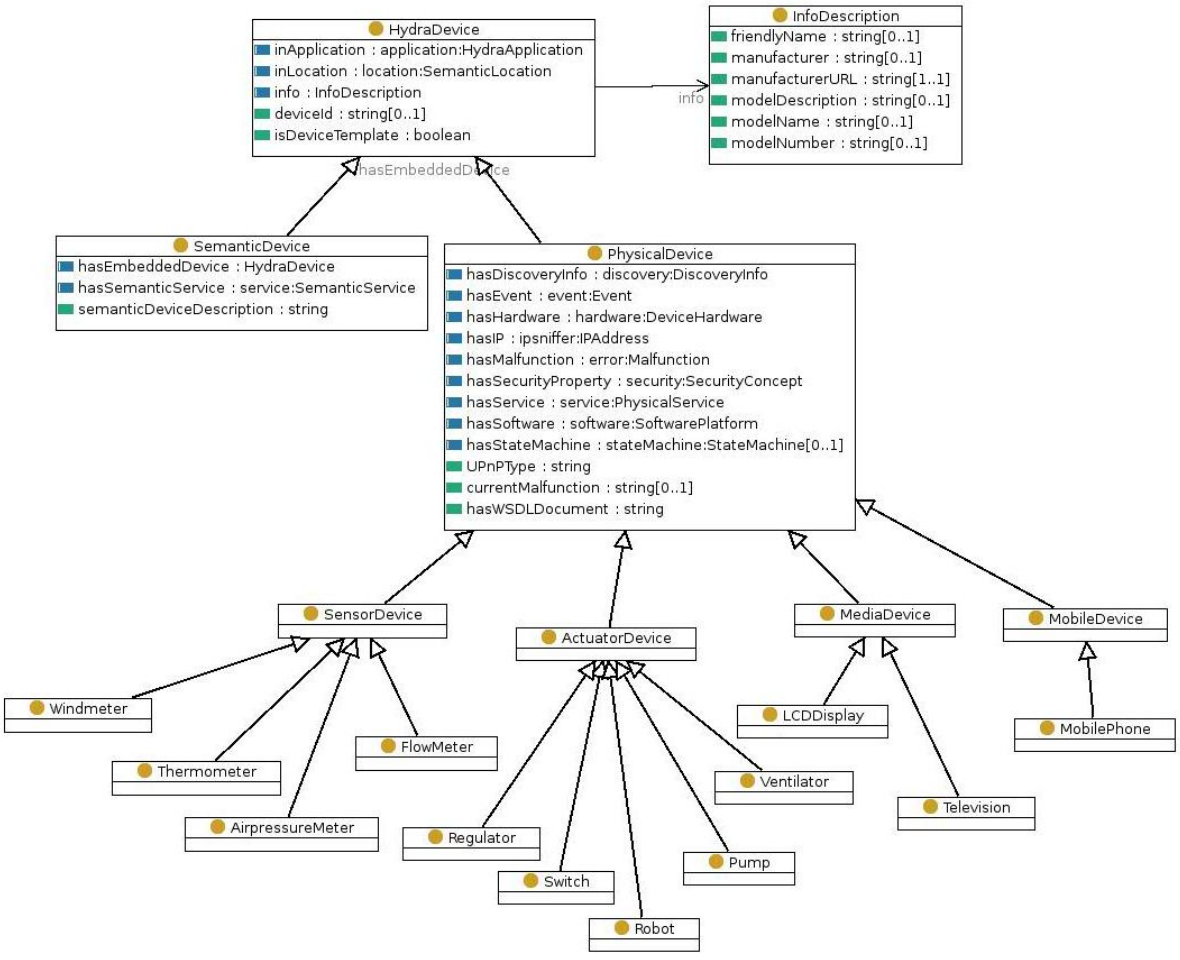
5.2.1 Basic device information

Basic device information represents general device information. The *HydraDevice* concept presents the main ontology class. The *HydraDevice* is further subclassed to the model of the *PhysicalDevice* and the *SemanticDevice*, which share the common device properties (such as deviceId or location), but have different semantic interpretation and behaviour.

The concept *InfoDescription* contains basic information about device friendly name, manufacturer data (such as manufacturer name and URL) and device model data, namely model name, model description and model number. The information is represented as OWL data type properties. The *InfoDescription* class is referred from the *HydraDevice* concept using the *info* OWL object property.

An important part of the basic device information is the representation of device type. The type of device is modelled as the OWL *is-a* hierarchy by sub classing the *PhycicalDevice* concept. This approach leads to the model of flexible device taxonomy, which can be further modified and extended by newly manufactured or not yet used device descriptions. The main purpose of device taxonomy is to reduce the whole model complexity by distributing the device information into smaller units. Each device type should refer only to relevant parts of all possible device information, for example relevant device capabilities, service types, malfunctions, etc. The Device taxonomy should also reduce the information complexity in both development and run-time process by selecting only the set of device information relevant to actual context. The hierarchy is defined for the physical devices, which are used as the lowest (physical and executable) level in semantic devices composition.

The semantic model of the basic device description is illustrated in FFigure 21. The initial device taxonomy was taken from AMIGO project vocabularies for device descriptions [AMIGO, 2006].



F

Figure 21: Device Taxonomy

Further, the OWL object property *hasEmbeddedDevice* of *SemanticDevice* concept recursively refers to *HydraDevice* concept. This property enables the creation of models of composite devices, such as in case of *HeatingSystem* device used in first system prototype application. *HeatingSystem* can be,

for example, composed of *Thermometer* and *Pump* devices. Property *hasEmbeddedDevice* enables to access information on several subsumption levels according to actual needs in dependence on actual context, run-time properties, required services, etc.

5.2.2 Device services

The device services ontology component presents the semantic description of device services on the higher, technology independent level. Hydra service model enables the interoperability between devices and services, employing the service capabilities, input and output parameters and supported communication protocols supporting the device interaction.

The semantic service specification is based on the OWL-S [OWL-S, 2004] standard, which is currently the most complete description of semantic mark-up for services following web service architecture (the overview of related standards for semantic web service mark-up is presented in D6.3 deliverable). The OWL-S approach was taken as the starting point for Hydra service model.

HydraService concept serves as the container for the two different service types *SemanticService* of *PhysicalService* assigned to the semantic or physical devices. *HydraService* concept is linked to the common service properties, such as quality of service, security properties, additional service capabilities or I/O parameters. *ServiceInput* and *ServiceOutput* parameters are specific subclasses of general *ServiceParameter* class and should be annotated to a semantic model describing various input and output types in the syntactic (for example, string, number) and semantic (for example, address, and user name) way. For more, the *ServiceOutput* contains the information of actual value and value range of the output parameter. This information should hold the actual value returned by the service and should be continually updated. Actual values can be used for example for diagnostics or various kinds of context-based decisions. Capabilities and input/output descriptions can be used for suitable service discovery or service composition, but also for semi-automatic or fully automatic generation of self-descriptive service user interfaces.

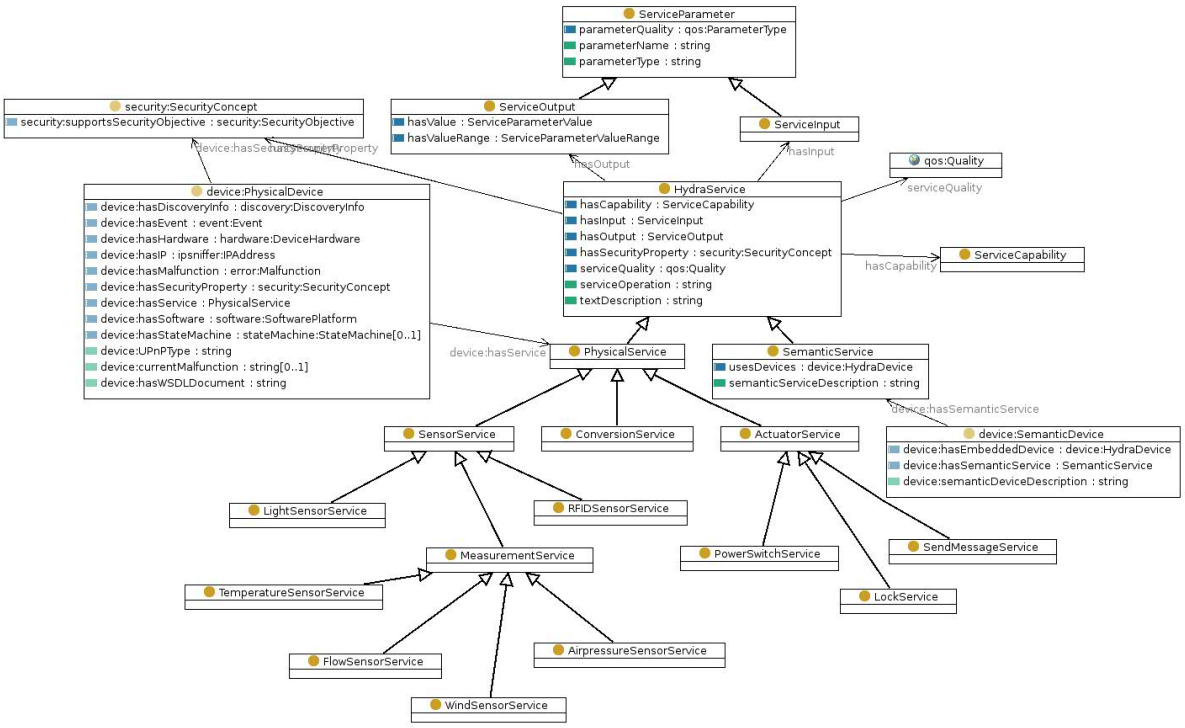


Figure 22: Modelling of services in the Hydra Device Ontology.

The *PhysicalService* concept represents the properties of real device services and contains the taxonomy of services. The taxonomy is also used to classify the services by their capabilities or

usage purposes. Using the service categorisation tends to reduced complexity of service discovery and development process by selection of only services of specified type or usage.

SemanticService represent the model of composition of devices and services used by the semantic device.

The proposed Hydra device services model represents one possible approach to service modelling and may be subject to further investigation and research related to possible existing and future semantic service mark-up standards (such as WSMO) and the system architecture requirements.

5.2.3 Device Events

Some simple devices, which are not able to provide a service interface, may instead provide simple functionalities in the form of generated events. The events are, similarly to the description of services, subclassed to the taxonomy of possible event types. Each *Event* is described by the *MetaInformation* providing the basic event description (frequency of event generation, trigger, event human-readable description). Each event contains the set of *EventKeys*, which are sent in key-value pairs by the device, when the event is generated. The event keys can be annotated to the quality of service ontology, which specifies the units of values. The model of events is illustrated in Figure 23.

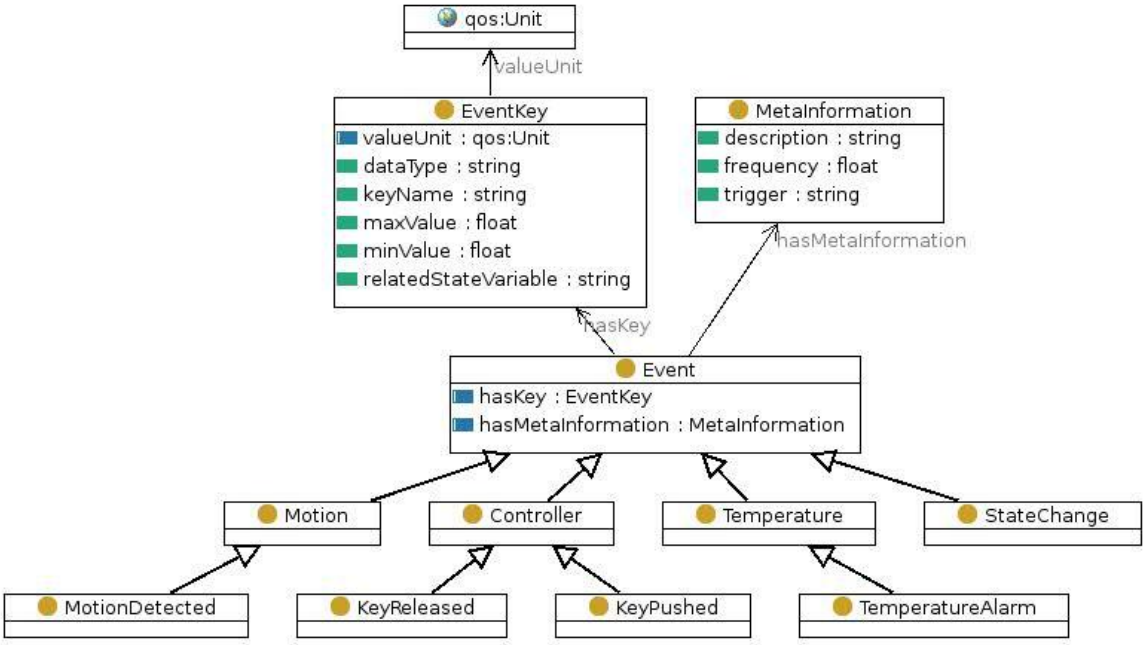


Figure 23: Event model in the Hydra Device Ontology

Actually, the event ontology is used as the semantic support for the developer, when creating the application. Using the event ontology, the developer can easily check, what should be taken into account, when device generating the events should be used in the application.

5.2.4 Device malfunctions

The semantic model of device malfunctions represents possible errors that may occur on devices. The concept *Malfunction* is referred from the *HydraDevice* concept using the *hasMalfunction* OWL object property. This concept contains general malfunction information, namely OWL data type properties *malfunctionName* and *malfunctionCode*, where property *malfunctionName* represents human readable name and *malfunctionCode* contains application specific malfunction reference.

Both properties are mainly used to access the information related to specific faults. OWL object property *hasCase* of *Malfunction* concept represents the one-to-many relation to potential malfunction cases represented by *MalfunctionCase* concept.

The concept *MalfunctionCase* contains two OWL data type properties *cause* and *remedy*, which contain the human readable name of particular cause and human readable remedy describing how to react to the given cause. Every device malfunction may have as many cases as needed.

In order to have a flexible model of malfunctions, the *Malfunction* concept can be further sub classed to several malfunction levels or severity, such as, error, fatal, warning and info. Possible severity levels can be further extended by the hierarchy of specific faults.

The model of basic device malfunctions is illustrated in .

Figure 24 .

Figure 24.

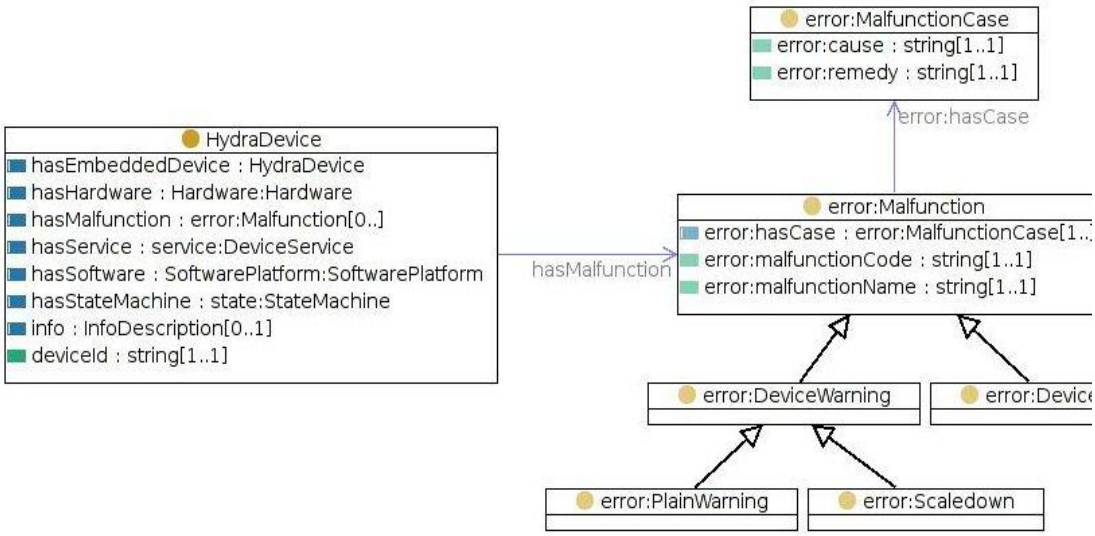


Figure 24: The malfunction part of the Hydra Device Ontology

Connecting the device taxonomy to the malfunction taxonomy creates a flexible representation of fault states, which may occur on various device types and the possibilities of their solutions. The malfunctions, using taxonomy relations, can be, according to actual context, used to retrieve the more general fault descriptions in case, when the required specific description for the concrete device (or device type) is missing. The connection of malfunction model and device state machine can be used for diagnostic purposes. The various faults related to specific ontology states can, for example, be used to predict or avoid the fatal error states of a device or to invoke the related call-back events to handle the error states that may occur in run-time.

5.2.5 Device capabilities and state machine

The device capabilities as part of the device ontology refer to the various Self-* Properties supporting models, such as models of state-machines tracking the run-time device/service state changes, models of device run-time request/response tracking and SWRL rules supporting mainly the self-monitoring and self-diagnosis processes. Details are found in WP4 Deliverable D4.8.

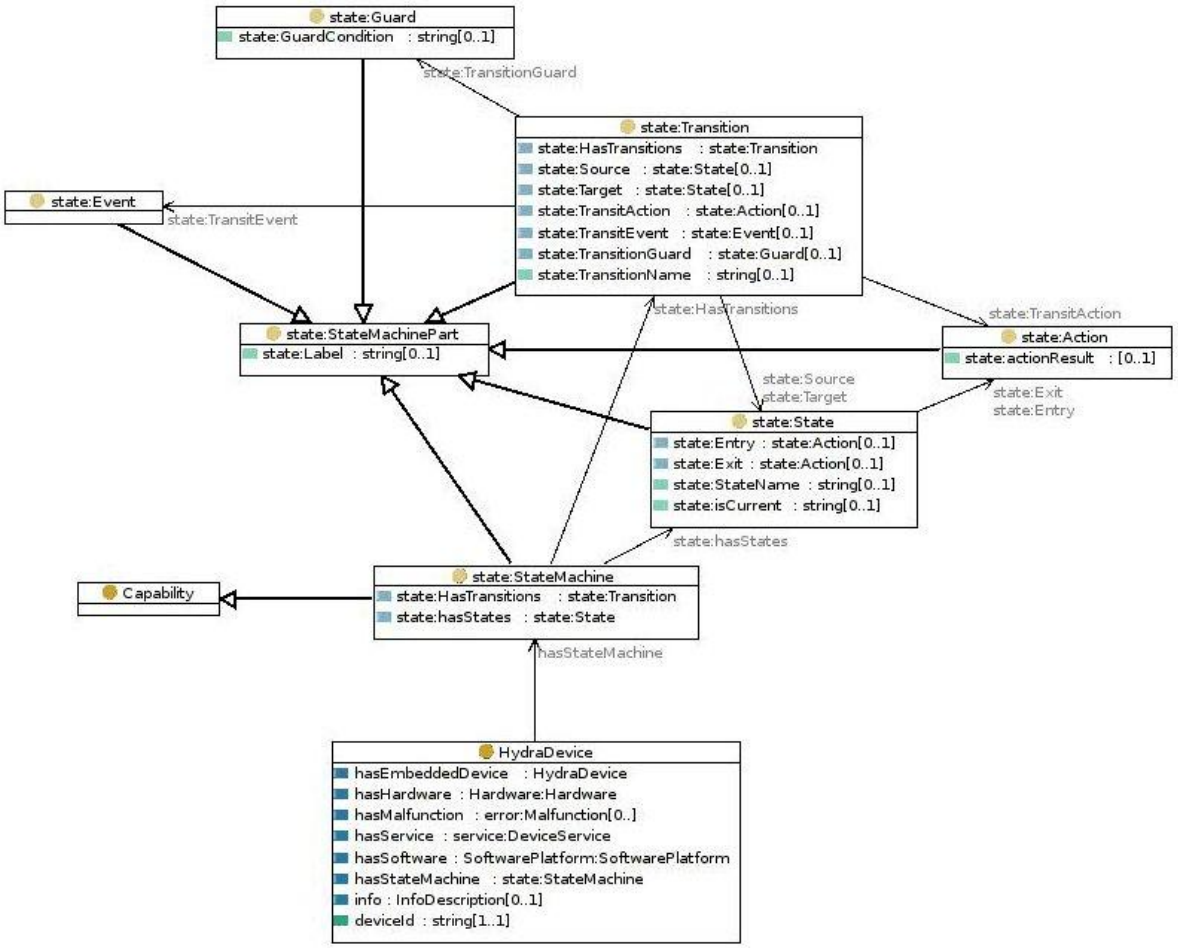


Figure 25: The State machine part of the Hydra Device Ontology

5.3 Semantic Discovery Model

When a new device enters the Hydra network, it is discovered using one of the low level discovery managers for various protocols such as Bluetooth, ZigBee or RFSwitch. In order to obtain the semantic model of this device and its services, the device entering the network has to be semantically resolved. That means, that the related device model has to be identified in the ontology. The model is used e.g. to describe several services provided by the device. The semantic resolution is performed using this model. This model contains the device discovery information for each specific device type. The general view of the model is shown in Figure 26.

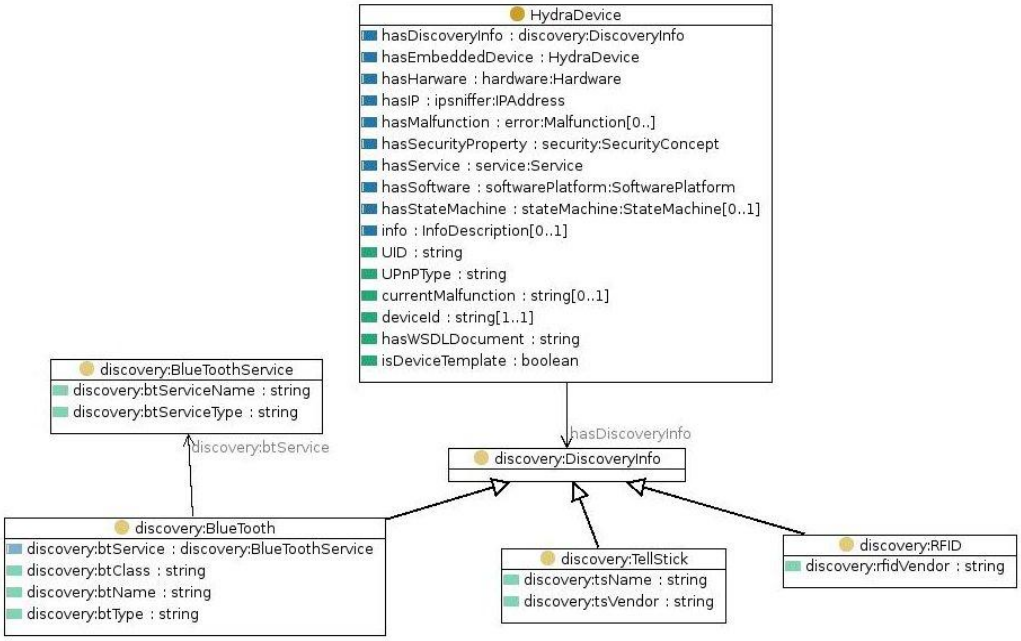


Figure 26: Semantic discovery model.

Once the device is discovered by the discovery manager, the obtained low level discovery information is transformed into a SPARQL query, which is fired against the ontology and the matching device descriptions are identified. As the low level discovery information is often too weak, in many cases, there are multiple matches of device models. Further resolution is improved by comparing the device manufacturer or model information, if possible. When there are still more possible matches, the suitable model has to be identified experimentally. Each matching model contains the description of device services, which can be executed against the physical device. If the device is able to respond to every service described in the particular model, this model is selected as the most suitable semantic representation of the device.

Semantic discovery identifies the semantic model, which is tied to physical device. This model enables the semantic support for the physical device. The semantic model is used for example to enable various kinds of semantic searches and resolutions, such as security resolution, context-related inference or retrieving the devices providing specific functionality, quality of service, etc.

The example of a semantic discovery model for the phone discovered by the Bluetooth protocol is in Figure 27.

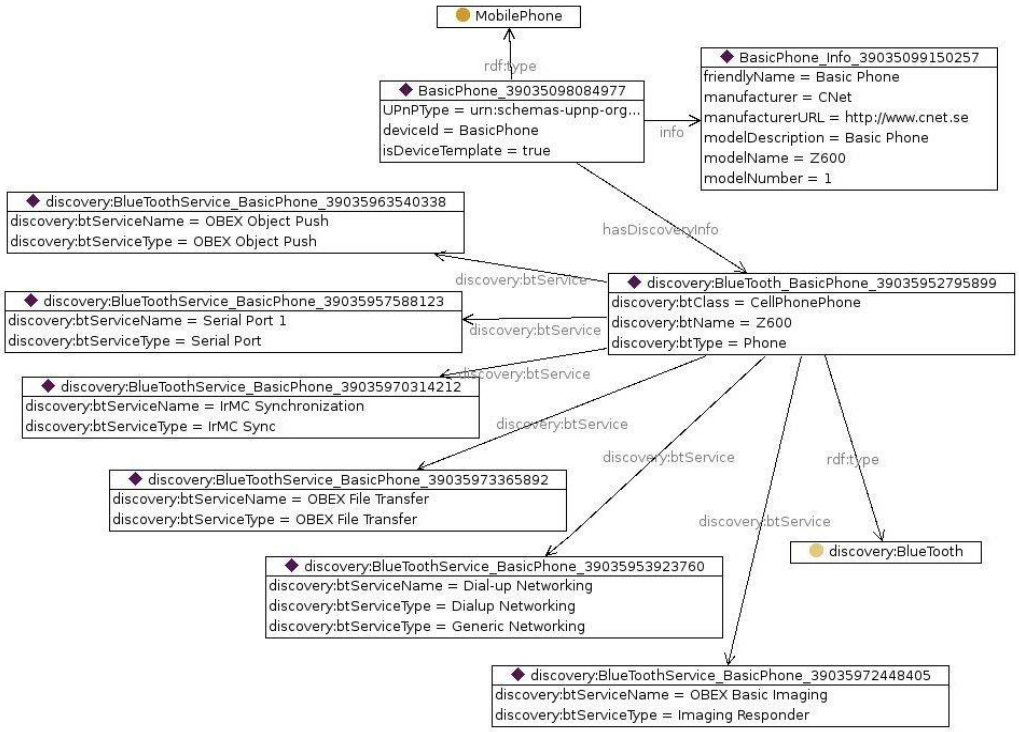


Figure 27: Bluetooth phone semantic discovery information.

5.4 Semantic Device Model

Each physical Hydra device used in an application provides a set of services, which can be directly used by the application developer. For example, the thermometer device may provide the "get temperature" or "set temperature" services. As introduced in Section 4.3, the idea behind the semantic devices is to enhance the application development by providing the application specific services, for example if there are more thermometers in the room, an application may provide "get average room temperature" or "hold the temperature on specified level" services. The concept of semantic devices brings the idea of specifying the application specific behaviour achieved as the composition of several Hydra devices services organized into complex units. Such complex units – logical aggregates of devices are called semantic devices.The semantic devcies may be implemented in two ways:

- Using the static mapping to the specific Hydra devices or other semantic devices and their services, in this case, the resulting behavior is hardcoded as the composition of specified Hydra devices services.
- Using the dynamic mapping to the devices semantically specified by various requirements, such as quality of service, context or security information (e.g. get the average temperature retrieved from any device capable to measure the temperature in degrees of Celsius in the room).

The Ontology model of semantic device is shown in Figure 28. Each specific device can be seen as a semantic device, thus basic *HydraDevice* concept extends the *SemanticDevice* concept. The same situation counts for device services, thus the basic *Service* model extends the *SemanticService* concept.

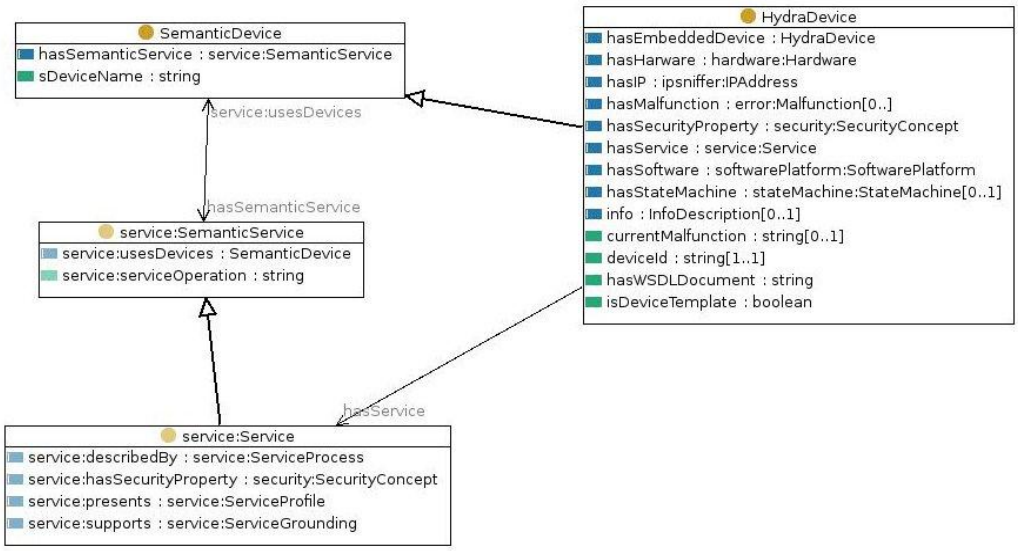


Figure 28: Model of semantic device.

The Semantic device contains only the device name and the set of provided semantic services. Each semantic service contains the specification of devices used by this service. The implementation of behaviour is provided by the developer. In this case, semantic device model can be used to support the development process, for example:

- By automatic generation of semantic device interface.
- By providing the interfaces for embedded devices.
- By providing the basic functionality for execution of various types of search queries.

The implementation of semantic services can be seen as the composition of used devices services. When semantic service implementation uses dynamic mapping to devices specified only by some kind of semantic requirements (e.g. all devices capable of temperature measuring), the devices satisfying the semantic specification are retrieved automatically. Thus, this kind of implementation uses the orchestration approach, where devices matching the requirements are specified only by defined requirements.

An example of a *TemperatureHandler* semantic device is shown in Figure 29. The semantic device has the semantic service *getAverageTemperature* using two thermometer devices in a static way. Even if the semantic service contains a set of specified devices, the implementation can be realized as the combination of statically defined devices and the orchestration behaviour. The static definition of used devices serves only as the case, when semantic service has to work exactly with some specific devices. But this specification does not entail any limitation for using also orchestrated devices. For example, the developer may decide to create a specific temperature alert device using just some selected thermometers in the room, which have to be specified (thermometers are specified as the concrete devices – static mapping). When the temperature measured by selected thermometers decreases below some level, the semantic device may perform the "low temperature alert" by blinking the light using any lamp in the room (lamp is specified only by location or by device type – dynamic mapping).

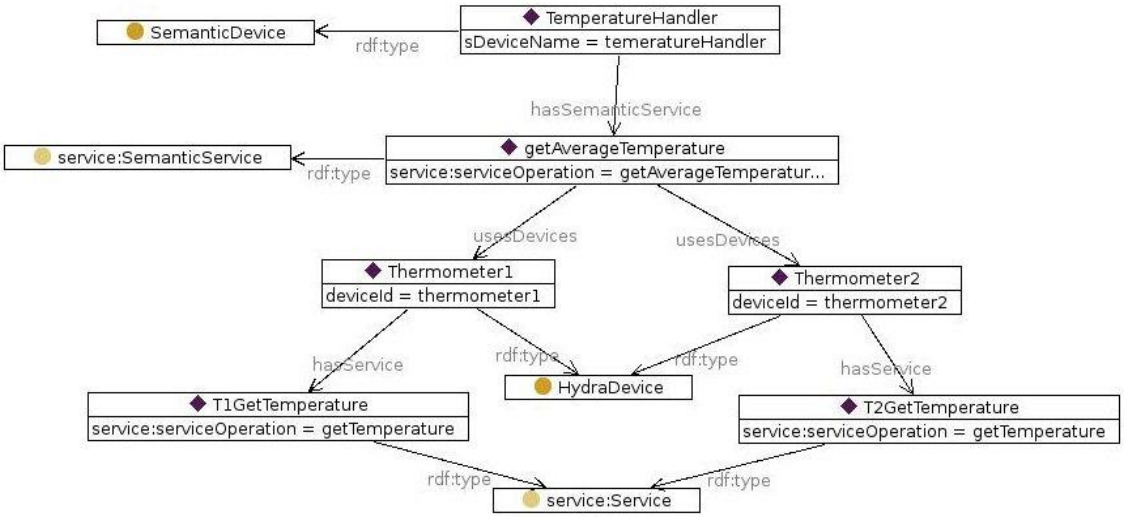


Figure 29: Example of *TemperatureHandler* semantic device.

More complex semantic devices may be also used as the decision units providing a specific functionality in terms of effectiveness by some specified criteria. For example, the application may use two semantic devices capable of controlling the light in the room. One semantic device controls the lamps, another controls the blinds. These two devices may be composed into a more complex semantic device, which would be capable for example to save the energy. Using the specific information, the device will be able to decide, how to perform the light control. In summer day-time it may use the blinds controlling semantic device to control light, in the evening or winter it can prefer to use the lamps controlling semantic device. Using more information about devices, e.g. various kinds of energy profiles, semantic devices can be used as standalone units implemented to perform the operations while satisfying the specified goals (e.g. energy saving). The application development can be radically simplified by using the existing semantic devices adjusted for the specific environment.

5.5 Application Specific Ontology

When designing an application, ontology can be used to model the application structure and the specific devices used. For example, in the case of a home automation application, it is helpful to specify, which locations (e.g. rooms) an application will have, which persons use the application, which concrete devices belongs to locations or are owned by concrete persons. An application ontology represents a simple model with two basic purposes:

- to specify simple context-related information describing the locations and persons
- to specify relation of concrete devices to locations or persons

This model should be prepared in design time, when developing the application implementation. At run-time, discovered physical devices are bound to the application model. It is not required to specify all of the devices used by the application. Application ontology should only use those devices, which should be used for specific computations in a similar way, as in the case of static/dynamic mapping for semantic devices. The concrete devices may be specified directly for the application, application locations, persons owning devices, but also by using the semantic devices. If the semantic device is used in the application, the set of concrete devices used by the semantic device (if any) can be inferred.

6. Main developments in third iteration

The main advancements (in WP6 as per the 3rd iteration) of the semantic MDA are mainly in the following areas:

- Device discovery
- Automatic generation of device web services and devices proxies
- An adapted Device Ontology
- SDK and DDK support for programming with devices
- Device profiling and annotation

6.1 Device Discovery

Several issues have been investigated and resolved for the management of the DAC and the discovery process:

- The Hydra discovery functions are able to discover other devices that use a number of different protocols; Bluetooth, UPnP, Zigbee etc. These may also be able to announce themselves to other devices using all these protocols.
- In this iteration we have moved from service composition at design time to resolving at run time when a set of devices and services that are present in the network constitute a composite device, and place this composite device in the DAC.
- We have analysed and improved semantic discovery process to be more precise.
- The ontology device descriptions have been refined with discovery information for different device and protocol types.
- A semantic discovery process (device is resolved in ontology by searching low-level discovery information) has been implemented.

6.2 Automatic generation of device web services and devices proxies

A cornerstone in the Semantic MDA is the ability to automatically generate device web services based on discovery information. The following progress was made during the third iteration.

- Automatic generation of WSDL description and web services from UPnP descriptions (SCPD and service descriptions).
- Adding Event Generation for devices, configurable based on state variables.
- Generation of SA-WSDL annotations from ontology for a number of device types.
- The discovery process currently works with an implicit software component model. This model represents the device managers and service managers that are selected for automatic proxy generation. This model is currently not available to the developer, but the device and service manager objects are, and can be specialized.

6.3 Adaptation of the Device Ontology

The semantic MDA in Hydra is driven from the device ontology. The following improvements and extensions have been done with respect to the device ontology:

- Basic design of ontology models for device energy (properties/consumption) models and middleware software components

- Extension of ontologies with models for semantic devices support and application/deployment models
- Revision of the device ontology with energy profile properties
- Revisions of the concepts and representation for semantic devices
- Ontology extensions: models for data acquisition support, QoS model, models of device events as the alternative to device services and software and hardware ontology refined
- Design of ontology support needed for Device Developers Kit (DDK)
- Integration of a QoS ontology

6.4 SDK and DDK support for programming with devices

To support application developers as well as device developers a number of tools and components have been designed and implemented:

- Definition of a set of generic energy device services
- Designing support for Semantic Devices with mapping to Hydra devices
- Revision of the device naming and identification scheme with persistent names and dynamic HID binding
- Specification and development of tool interfaces to the ontology manager for application developers and device manufacturers.

6.5 Device profiling and annotation

To further improve on how devices are described and their services modeled the following issues have been pursued:

- Definition of energy profiles and energy policies (rule-based language) for the orchestration of energy consumption parameters and services.
- Preliminary survey on device energy profile models and modelling of device energy properties/consumption/features
- Development of support for device annotation of energy features
- Implementation of SAWSDL annotations of device service input/output parameters to ontology concepts.

7. Future Work

7.1 SW components ontology

The purpose of SW components ontology is to provide a model of the middleware software [Oberle, 2006] components that comprise a Hydra configuration (Hydra-117: Hydra component ontology, Hydra-139: Knowledge model of hydra middleware). This model will support activities of composition, configuration, deployment and monitoring of the Hydra middleware (Hydra-115: Decomposable middleware, Hydra-122: Configurable and easy to install middleware).

The requirements to a component model are well met by the OSGi component model (which is also basis for the dynamic component model in Java as described in JSR-291[1]. We will use this as a basis for component ontology. The specification allows components to be declared through metadata and be assembled at runtime using a class loader delegation network. The specification also allows components to be dynamically life cycle managed (install, start, stop, update, uninstall). The JSR-291 specification is basically OSGi R4. It is suggested to model the OSGi Module Layer as ontology.

7.2 Ontology design and management

The Semantic MDA of Hydra includes certain generic ontology management functions for the Hydra IDE. The Hydra middleware as such does not impose any specific engineering or management methods with respect to ontologies, but should be open to any approach.

In Hydra we adopt the following view on the management of ontologies:

Ontology management is the whole set of methods and techniques that is necessary to efficiently use multiple variants of ontologies from possibly different sources for different tasks. Therefore, an ontology management system should be a framework for creating, modifying, versioning, querying, and storing ontologies. It should allow an application to work with ontology without worrying about how the ontology is stored and accessed, how queries are processed, etc. Ontology modification is accommodated when an ontology management system allows changes to the ontology that is in use, without considering the consistency. Ontology evolution is accommodated when an ontology management system facilitates the modification of ontology by preserving its consistency. Ontology versioning is accommodated when an ontology management system allows handling of ontology changes by creating and managing different versions of it [Hydra, 2006].

"Ontologies, to be effective, need to change as fast as the parts of the world they describe" (Davies et al.). This would hold for any model claiming to be an accurate abstraction of some part of the world, but becomes very critical in an ontology-based system like Hydra where openness and reasoning over system capabilities expressed in models are vital.

Ontology changes can emanate from user requirements on changes to structure and classification; in Hydra this would be the developer users' requirements. The changes can also be induced by changes in the underlying domain objects being modelled by the ontology, in Hydra; this would e.g. be changes in device capabilities, in security protocols, or in middleware components.

7.2.1 Ontology design process

The initial Hydra ontology design process is been manual, performed by ontology engineering experts (a Hydra partner) and domain (device) experts (developer users / focus group members).

The requirements capturing process is part of and based on the requirements work performed as part of WP2 and the Volere elicitation process. This naturally follows the iterative approach of the Hydra project's development model.

[1] http://jcp.org/aboutJava/communityprocess/final/jsr291/

7.2.2 Modifying and Evolving ontologies in Hydra

A developer must be able to define new or extend existing device ontologies (Hydra-101: Manual device ontology definition), and hence the SDK/IDE is required to provide the necessary tools, including an ontology browser and editor.

To semantically maintain the device ontologies, it is necessary to identify and find the relevant descriptive sources and to retrieve the necessary semantic descriptions. This description must then be transformed into the model structure of the actual ontology.

The manual ontology updates are complemented by support mechanisms for (semi-) automatic extension to ontologies. This support can be divided into mechanisms for:

- device descriptions mining and parsing
- device instance change discovery and capture

7.2.2.1 Automation support for classifying devices

Hydra ontology evolution can be supported by providing functions for the automatic classification of devices (Hydra-103: Automatic device ontology construction).

The construction of device ontology should be facilitated through finding and parsing product or device descriptions to annotate and produce ontology entries. By this we mean the process of retrieving device related information and the transformation of this into a device description which can be included in the device ontology as a (sub-) class. The transformation process should be able to map multiple input formats (such as MS Word, PDF, HTML, XML), to the ontology language of Hydra (OWL).

The updated ontology description is then usable in the process of dynamically binding a specific device instance to the particular device class in the ontology (Hydra-110: Device Categorisation in runtime).

7.2.2.2 Change discovery and capture

The complementary function to the above is to capture changes to existing devices and to propagate these as updates to the ontology (Hydra-126: Automatic Device ontology updates). This has been referred to as data-driven change discovery, in ontology literature.

7.2.3 Mediation, aligning and merging of ontologies

A Hydra installation must be able to interface with existing ontologies (Hydra-141: Harmonization of 3rd party device ontologies). A developer should be able to import external (device) ontology and be provided with tools for its adaptation and use in application development.

8. Appendix: Standards and Tools

8.1 Standards used

8.1.1 Modelling and query languages

8.1.1.1 Ontology Web Language (OWL)

The OWL Web Ontology Language [McGuinness, 2004] is designed for use by applications that need to process the content of information instead of just presenting information to humans. OWL facilitates greater machine interpretability of Web content than that supported by XML, RDF, and RDF Schema (RDF-S) by providing additional vocabulary along with formal semantics. OWL has three increasingly expressive sublanguages: OWL Lite, OWL DL, and OWL Full.

The basic reasons for decision to use of OWL for modelling in Hydra are:

- OWL extends all other languages like XML, RDF, and RDF-S. Actually, OWL has been developed on top of the existing XML and RDF standards, which did not appear adequate for achieving efficient semantic interoperability.
    - E.g. in XML and XML Schema same term may be used with different meaning in different contexts, and different terms may be used for items that have the same meaning.
    - E.g. RDF and RDF-S address some problem by allowing simple semantics to be associated with identifiers. With RDFS, one can define classes that may have multiple subclasses and super classes, and can define properties, which may have sub properties, domains, and ranges. However, in order to achieve interoperation between numerous, autonomously developed and managed schemas, richer semantics are needed, like disjoints and cardinality of relations.
    - OWL adds more vocabulary for describing properties and classes, relations between classes, cardinality, equality, richer typing of properties, characteristics of properties and enumerated classes, and all available in three increasingly expressive and increasingly complex sublanguages (Lite, DL, Full) designed for use by specific communities of implementers and users.
- OWL is well-known widely used open W3C recommendation with very good support and promising potential and real usage in several industry applications.
- OWL has wide support of modelling tools, platforms, and reasoners.
- Previous languages could express (in most cases) the same things, but for some of them OWL provide direct solution by a predefined type of predicates.
- There are several well-known mechanisms for expressing OWL-Lite and OWL-DL ontologies to stay on decidable level, where Description Logic (DL) could be used correctly.
- OWL language has proved its potential to use for modelling of semantic interoperability in several middleware-based applications and domains.

In Hydra the same OWL-based framework can be used for representation of context, device descriptions (capabilities), descriptions of middleware components, services, security aspects, with several specific goals such as:

- Use of semantic models of device descriptions and services for model-driven architecture design (code generation for devices and services).
- Use of semantic-based models in run-time for discovery of devices (adoption to interfaces supported by device), resolving application requests, resolving security requirements, services execution and context awareness.

- Modelling of particular elements to create necessary semantic-based models mostly based on the Semantic Web technologies.

OWL Lite and DL should be used for reasoning with DL-reasoners for organising context definitions, merging domain knowledge into these definitions, and performing recognition of contexts from sensor inputs. The ontology has many merits, of which the most notable are the excellent extensibility, and high expression power. Many systems in the "ubiquitous" and embedded environments are developed using DL-based ontologies and used with DL-based reasoning. Usually, ontologies are used for modelling context that the systems should collect and analyze. A pure DL-based approach, however, has certain limitations in a context environment. OWL and other ontology languages based on Description Logic cannot properly handle rules expressed in Horn-Logic. Hence, to ensure syntactic and semantic interoperability on device level (e.g. "low-level" ontologies), SWRL (Semantic Web Rule Language) can be used for expressing rules.

8.1.1.2 Semantic Web Rule Language (SWRL)

SWRL [SWRL, 2004] combines sublanguages of the OWL (OWL DL and Lite) with those of the Rule Mark-up Language (Unary/Binary Datalog). Actually, it is an extension of OWL which adds support for Datalog syntax-style rules over OWL DL ontologies. Instead of arbitrary predicates (as in Datalog), SWRL allows arbitrary OWL DL descriptions in both the head and the body of rules, where a unary predicate corresponds to an OWL class and a binary predicate corresponds to an OWL property. While a subset of SWRL falls inside Horn Logic, a SWRL knowledge base easily goes beyond this fragment, because of the use of classical negation and existentially quantified variables and disjunction in the head of the rule. A set of Horn Logic formulae can be reduced to standard Logic Programming rules; the Horn Logic formulae and the Logic Programming rules entail exactly the same set of ground formulae. Consequently, SWRL and standard rule languages differ in expressiveness. The advantage of common rule languages which are based on Horn Logic is the efficient reasoning support which has been developed for certain reasoning tasks like query answering. By going beyond the Horn fragment, SWRL loses this advantage.

More details about usage of modelling directly for Hydra-related purposes are presented in particular chapters in this document and/or other deliverables related to already mentioned topics like context awareness, semantic security, semantic interoperability in Hydra middleware (device discovery and usage in runtime, model-driven architecture design, etc).

8.1.1.3 SPARQL

Last topic to be mentioned in this section is querying of ontologies, this is based on the well-known (and already mentioned) SPARQL. Many semantic reasoners/engines have built-in support for this query language (e.g. Jena, RacerPro and Pellet). SPARQL is an RDF query language; its name is a recursive acronym that stands for SPARQL Protocol and RDF Query Language, and it is undergoing standardization under the W3C (currently November 2007 the status of SPARQL changed into Proposed Recommendation). The beneficial properties of a query language (like SPARQL) for the Semantic Web defined [Bailey, 2005]:

- Referentially transparent - "within the same scope, an expression always means the same",
- Strong answer closure - the result of a query can be used as the input for further querying,
- Set-oriented functional – also known as a backtracking-free logic programming,
- Incomplete queries and answers - support for data on the Web that may not have defined schemas,
- Multiple serialisation aware - able to serialise data to various formats including XML, OWL, RDF,
- Queries that support reasoning capabilities - the ability to query different data sources and infer new statements.

SPARQL is a Server-Client-based RDF query language. It has SQL syntax and is influenced by RDQL and SquishQL4. SPARQL can process more complex query than RDQL and provides optional variable binding and result size control mechanisms for real world usage. SPARQL allows for a query to

consist of triple patterns, conjunctions, disjunctions, and optional patterns. Several implementations for multiple programming languages exist. The SPARQL query processor will search for sets of triples that match particular triple patterns, binding the variables in the query to the corresponding parts of each triple. To make queries concise, SPARQL allows the definition of prefixes and base URIs.

8.1.2 Reasoners

Reasoning over designed ontologies is important part of any semantic-based application. Here we can see several important aspects for usage of particular reasoners. First, reasoning over created ontology and their instances, querying languages over meta-data. The selection among the aforementioned alternatives is basically based on the language capabilities and the availability of further querying APIs and frameworks for it (it is a fact that available frameworks or querying APIs are strongly associated and dependent on the languages).

8.1.2.1 JENA

According to the fact that OWL is used for modelling in Hydra middleware, it is natural that reasoners in our case have to support OWL-based (DL and OWL-Lite) reasoning. The main application element of Hydra middleware responsible for ontologies is Application Ontology Manager. In order to achieve unified and comprehensive solution in programmatic way, Jena Semantic Web Framework (http://jena.sourceforge.net/) has been used for implementation of the manager. Jena is specifically suited to develop Java-based Semantic Web applications. It is open source and grown out of work with the HP Labs Semantic Web Programme. The Jena Framework includes:

- A RDF API
- Reading and writing RDF in RDF/XML, N3 and N-Triples
- An OWL API
- In-memory and persistent storage
- SPARQL query engine
- Rule support – own rule engine

Jena provides a very comprehensive framework easy usable not only for reasoning, but also for other purposes of querying, persisting, updating and versioning of different types of ontologies in Hydra middleware.

The only weakness of the Jena framework is SWRL support. Jena has its own Rule engine support, which is slightly different to standard SWRL. Actually, in most cases (where SWRL is not directly used) Jena prove its potential, only in some cases where SWRL plays an important role (e.g. see chapter about use of models for context awareness) it can be problematic.

8.1.2.2 RacerPro

During the development and design of SWRL-based parts of middleware semantics another engine has been used – RacerPro (http://www.racer-systems.com/). RacerPro is a knowledge representation system that implements a highly optimized calculus for a very expressive description logic augmented with qualifying number restrictions, role hierarchies, inverse roles, and transitive roles. In addition to these basic features, RACER also provides facilities for algebraic reasoning including concrete domains for dealing with min/max restrictions over the integers, linear polynomial (in-)equations over the reals or cardinals with order relations, nonlinear multivariate polynomial (in-)equations over complex numbers, equalities and inequalities of strings. Actually, RacerPro is commercial and can be only used as trial for academic/research purposes, as it was somehow used also in our case.

8.1.2.3 Pellet

A solution for future can be using of another open-source engine for rule support. Pellet (http://pellet.owldl.com/) has an implementation of an algorithm for a DL-safe rules extension to OWL-DL. This implementation allows one to load and reason with DL-safe rules encoded in SWRL.

Pellet has also been coupled with a Datalog reasoner to support AL-log (Datalog + OWL DL). This coupling implements the traditional algorithm and a new pre-compilation technique that is incomplete but more efficient. What is important here is that there is implemented reasoner interface for Jena, so it is possible to use the rule support based on SWRL within whole framework.

Pellet reasoner was used in the ontology development process as the part of TopBraid composer (see bellow).

8.2 Platform and Tools

In the ontology development process, includes two ontology editing tools supporting all of used standards languages: TopBraid composer and Protégé-OWL editor.

8.2.1 TopBraid composer

TopBraid Composer (http://www.topbraidcomposer.com/), a component of TopBraid Suite, is a modelling tool for the creation and maintenance of semantic models (ontologies). It is a complete editor for RDF(S) and OWL models, as well as a platform for other RDF-based components and services.

TopBraid Composer enables individual users and communities to collaborate effectively in developing Semantic Web ontologies. Key features of TopBraid Composer include:

- Standards-based, syntax directed development of RDFS and OWL ontologies, SPARQL queries and SWRL rules using ontology-driven forms, which can be customized. Ontologies can be developed using form-based GUI or also the manual source code editing.
- Imports and namespace management.
- Re-use of the legacy models and data through XML, UML, spreadsheet and database schema imports.
- Visualization and diagramming using UML class like diagrams or visual RDF graphs.
- Consistency checking and debugging.
- Multi-user support.
- HTML documentation generation.

TopBraid Composer is implemented as an Eclipse plug-in. Many other Eclipse plugins for editing other languages such as UML and XML exist, and therefore users can use a single tooling environment for many different modelling tasks. Furthermore, the foundation on the Eclipse plug-in architecture means that developers can build additional services (such as custom visualization and reasoning engines) on top of TopBraid Composer.

TopBraid Composer is built on top of Jena, a Semantic Web framework from HP Labs. Jena is open-source and plug-in developers will be able to exploit arbitrary Jena-based services. TopBraid Composer is also shipped with the OWL DL Pellet reasoner from the University of Maryland MindLab. Additional inference engines can be integrated and specified in the configuration preferences.

8.2.2 Protégé-OWL editor

The Protégé-OWL (http://protege.stanford.edu/overview/protege-owl.html) editor is an extension of Protégé (http://protege.stanford.edu/) that supports the OWL. The Protégé platform supports two main ways of modelling ontologies:

- The Protégé-Frames editor enables users to build and populate the frame-based ontologies (in accordance with the Open Knowledge Based Connectivity Protocol (OKBC)). Using this modelling approach, ontology consists of a set of classes organized in a subsumption hierarchy representing a domain concepts, a set of slots describing the properties of classes and relationships, and a set of instances of defined classes.
- The Protégé-OWL editor enables users to build ontologies directly on OWL standard.

Hydra ontologies are modelled using OWL; the Protégé-OWL editor was used for development purposes. Protégé OWL provides a variety of features that makes it very useful for building ontologies in OWL, namely:

- Loaded or newly created ontologies can be maintained using form-based GUI. In various visual ways of editing the classes, properties and individuals.
- Wizards to streamline complex tasks supporting common ontology-engineering patterns, such as creating groups of classes, making a set of classes disjoint, creating a matrix of properties in order to set many property values, and creating value partitions.
- Direct access to reasoners is used for three default types of reasoning: (1) consistency checking, (2) classification (subsumption), and (3) instance classification).
- Multi-user support for synchronous knowledge entry.
- Support for multiple storage formats. Current formats include Clips, XML, RDF, N-TRIPLE, N3, TURTLE and OWL.

Protégé-OWL's flexible architecture makes it easy to configure and extend the tool. Protégé-OWL is integrated with Jena and has an open-source Java API for the development of custom-tailored user interface components or arbitrary Semantic Web services.

Protégé has also strong ontology visualisation tools implemented as Protégé plug-ins. The well known and commonly used are OWLViz and OntoViz plug-ins.

OWLViz is designed to be used with the Protege OWL plug-in. It enables the class hierarchies in an OWL Ontology to be viewed and incrementally navigated, allowing comparison of the asserted class hierarchy and the inferred class hierarchy. OWLViz integrates with the Protege-OWL plug-in, using the same colour scheme so that primitive and defined classes can be distinguished, computed changes to the class hierarchy may be clearly seen, and inconsistent concepts are highlighted in red. OWLViz has the facility to save both the asserted and inferred views of the class hierarchy to various concrete graphics formats including png, jpeg and svg.

The OntoViz Tab allows you to visualize Protege ontologies with the help of highly sophisticated graph visualization software called GraphViz (http://www.graphviz.org/) from AT&T. The types of visualizations are highly configurable and include:

- Picking a set of classes or instances to visualize part of ontology.
- Displaying slots and slot edges.
- Specifying colours for nodes and edges.
- When picking only a few classes or instances, you can apply various closure operators (e.g., subclasses, super classes) to visualize their vicinity.

9. Appendix: Components implementing the MDA

This appendix describes the middleware software components that implement the main parts of the semantic MDA, explaining their roles, functions and component structure. The corresponding software deliverables are D6.7 and D6.8. For details of the overall Hydra architecture we refer to deliverable D3.9.

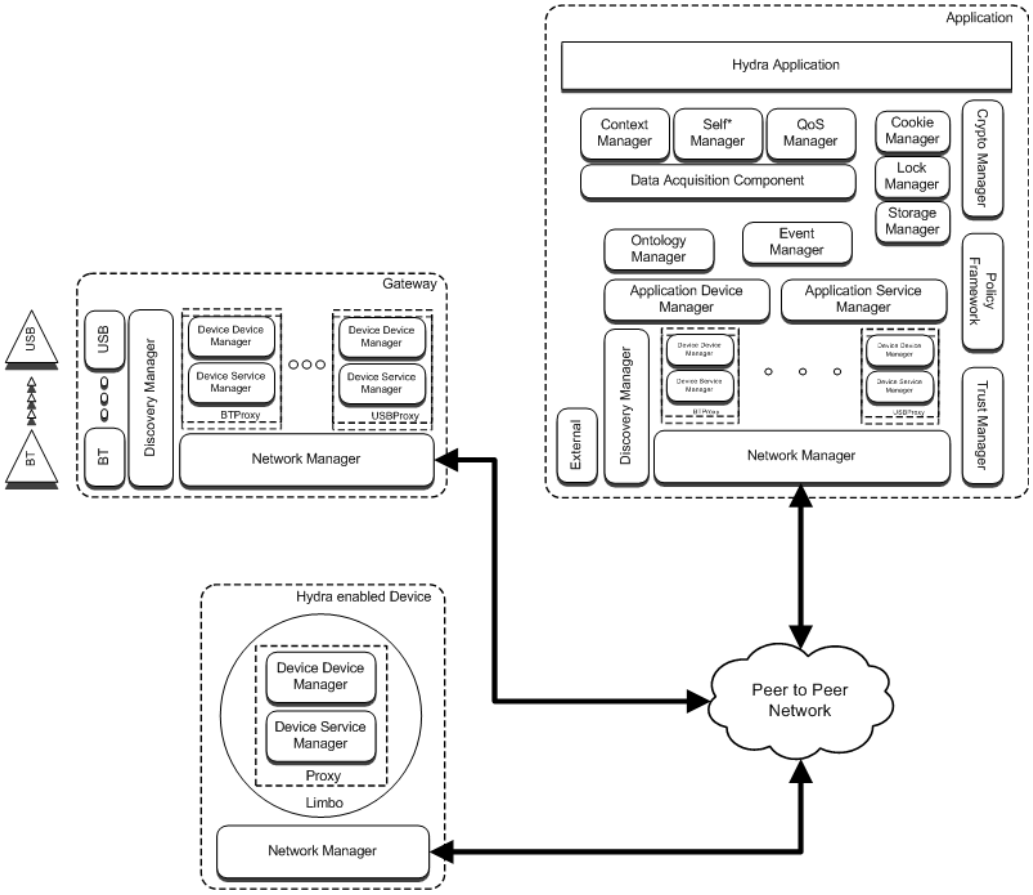


Figure 30: Hydra Software components (managers) architecture

9.1 The Application Device Manager

The Application Device Manager manages all knowledge regarding devices that have been discovered and are active in the Hydra network. It maintains the Device Application Catalogue (DAC) and makes use of a set of Discovery Managers which are running locally on different gateways to do physical discovery of devices (Bluetooth, z-wave, ZigBee, RF-switches, serial ports). Figure 31 displays the main class structure of the Application Device Manager.

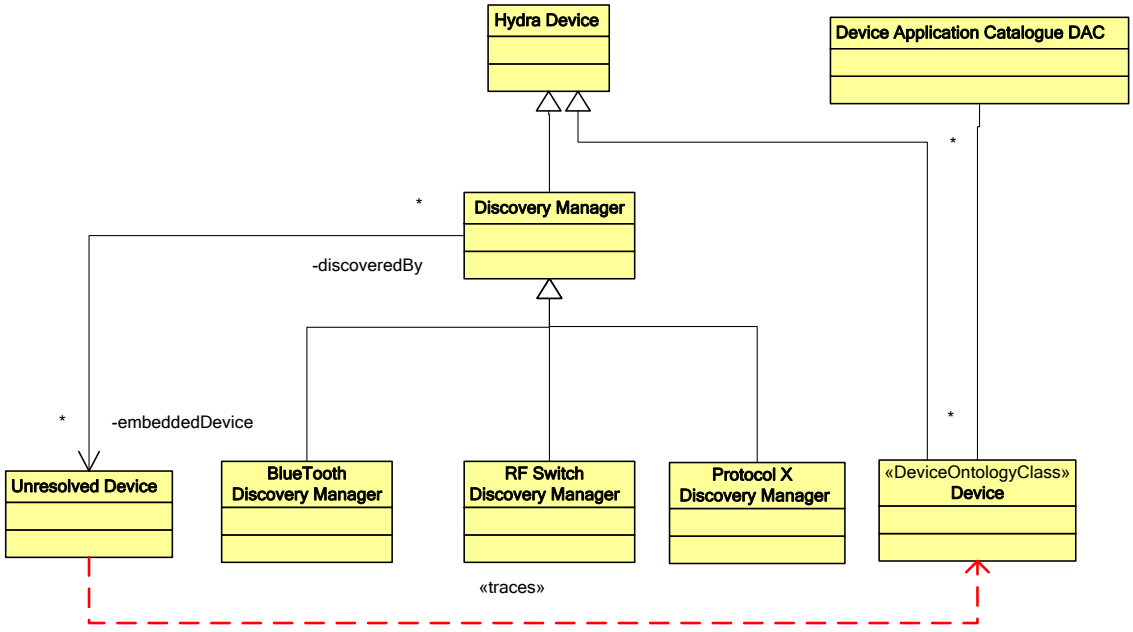


Figure 31: Application Device Manager (Discovery Manager) Main Structure

Main Functionalitie:

- Discovering devices
- Semantically resolving the device type and available services based on the Device Ontology
- Creating a service interface for the device
- Managing semantic device descriptions
- Providing semantic device aggregation
- Managing the Device Application Catalogue (DAC)

9.1.1 Related WP6 requirements

[Hydra-91] Any Hydra device should have an associated description	
Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	For management, search and discovery purposes, all Hydra enabled devices should be described (classified) according to the Hydra device ontology.
Source:	WP6 MDA scenario
Fit Criteria:	Any device associated to a Hydra application is also included in the Hydra device ontology, and its description can be retrieved.
Developer Satisfaction:	high
Developer Dissatisfaction:	high

[Hydra-108] Device discovery

Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	Middleware should be able to detect new device that enters the network
Source:	St. Agustin
Fit Criteria:	7 of 10 devices are discovered
Developer Satisfaction:	very high
Developer Dissatisfaction:	high

[Hydra-110] Device Categorisation in runtime **Created: 28/Nov/06 Updated: 09/Oct/07**	
Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	Middleware should after discovery of device be able to categorise a device based on device ontology information.
Source:	WP6 MDA Focus Group
Fit Criteria:	7 of 10 devices are correctly categorised and described.
Developer Satisfaction:	very high
Developer Dissatisfaction:	high
Dependencies:	101

[Hydra-111] Dynamic Web Service Binding	
Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	Middleware should be able to after device discovery and categorisation expose a new device as a web service that can be called without re-compilation.
Source:	WP6 SoA Focus Group
Fit Criteria:	New devices are callable and controllable in 7 out of 10 cases.
Developer Satisfaction:	very high
Developer Dissatisfaction:	very high

[Hydra-112] Dynamic Web Service Generation	
Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	Configuration tool that is able to generate the necessary interfaces to wrap the device functionality as a web service.
Source:	WP6 SoA Focus Group
Fit Criteria:	7 of 10 device functionalities are automatically represented as web services
Developer Satisfaction:	very high
Developer	high

Dissatisfaction:	
[Hydra-120] Multiple Device Virtualisations	
Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	It should be possible to have several different views/virtualisations of a device depending on context and applications.
Source:	WP6 MDA Focus Group
Fit Criteria:	At least 2 different virtualisations are provided
Developer Satisfaction:	high
Developer Dissatisfaction:	high

[Hydra-376] Security requirements must be part of the Hydra MDA	
Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	Security must be defined to be resolved semantically
Source:	WP 6 Focus group Kosice
Fit Criteria:	Security model can be defined semantically
Developer Satisfaction:	high
Developer Dissatisfaction:	high

9.1.2 Internal Components

The figure below displays the three main subcomponents of the Application Device Manager that will be described in more detail by the following subsections.

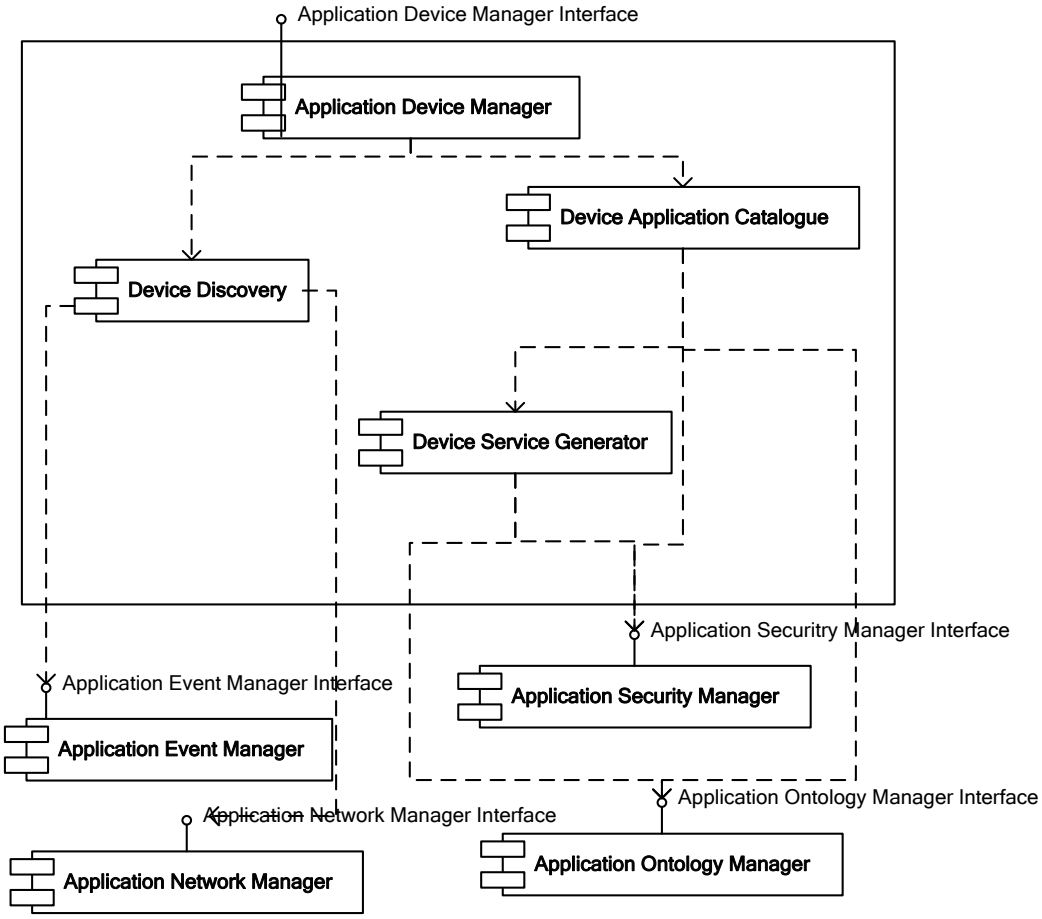


Figure 32: Application Device Manager

9.1.2.1 Device Discovery

Purpose

One of the major functions of the Application Device Manager is to discover new devices in the network. It will support user-initiated discovery as well as automatic schemes. Requirements 108 and 218 are associated with this module.

Main Functionalities

For each device protocol such as BlueTooth and ZigBee there is a dedicated discovery module that manages the protocol specifics. Discovery managers run on Hydra gateways where they look for physical devices such as Bluetooth devices.

Description

This is the base class for all discovery managers in Hydra. A discovery manager is part of the Application Device Manager. A discovery manager keeps track of the devices it has discovered. As long as the devices are unresolved they are treated as embedded devices of the Discovery Manager. A discovery manager runs locally on a gateway/PC where it looks for remote devices such as Bluetooth or RF switches devices. The discovery manager has direct access to the device objects it has created. Furthermore, the corresponding *DiscoveryManager* class is inherited by specializations such as *BluetoothDiscoveryManager*, *DeviceControllerDiscoveryManager*, *SerialPortDiscoveryManager*, *RfswitchDiscoveryManager*, and *ZigBeeDiscoveryManager*

9.1.2.2 Device Application Catalogue

Purpose

The Device Application Catalogue keeps track of and manages all devices that are currently active within one application. It is a view on the Device Ontology and a current set of (discovered) devices.

Main Functionalities

- Maintains a database of discovered devices for an application
- Maintains a mapping of logical device names to Hydra Identifiers (HIDs)
- Provides a HID query interface for other managers
- Provides a search function over current DAC member devices
- Provides a search function onto the Device ontology
- Stores Energy Profiles and Policies for Devices and Applications

Description

The Device Application Catalogue can be queried about existing devices and their status. It can also provide service interfaces for the different devices upon request. The Device Application Catalogue will also keep track of when the device entered the system, when it was last heard of and its current state. The Device Application Catalogue should also provide methods for removing devices. Requirements 91, 98, 110 and 111 are associated with this module.

9.1.2.3 Device Service Generator

Purpose

The Device Service Generator is responsible for generating a service interface for a certain device. It will create web services as well as UPnP services.

Main Functionalities

Generates five web services for a Hydra Device,

- A device type specific web service, exposing the device functions
- A Generic Hydra web service, exposing metadata and management functions common to all Hydra Devices
- An Energy web service, providing a set of functions for the monitoring and control of energy consumption of devices.
- A Memory Service which allows logging and storing of device internal data such as state variables and energy consumption data.
- A Location Service which can be used to query the device about is location and position.

Description

Provides service interfaces for a Hydra device, to allow proxy-based access. The service interface is created based on information in the device ontology.

9.2 Application Service Manager

The purpose of the Application Service Manager is to discover, create and execute semantic (web) service services on top of devices. It adds a service layer above the Application Device Manager. Services might map to several device functionalities.

Main Functions:

- Discovering service
- Creating semantic services for service orchestration and mapping to device service.
- Provide a service query interface to allow applications to locate services that match their requirements.
- Service descriptions and annotations.

9.2.1 Related WP6 requirements

[Hydra-104] Automatic Discovery of Services	
Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	It should be possible to configure the middleware to discover available services that meets defined criteria.
Source:	St. Augustin
Fit Criteria:	8 of 10 services are automatically discovered.
Developer Satisfaction:	high
Developer Dissatisfaction:	high

[Hydra-113] Composition (of services and devices)	
Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	In order to enhance or replace application level functions it will be useful to be able to compose services and devices from different providers and/or manufacturers into high level services/devices
Source:	WP6 MDA Focus Group, WP6 eHealth Focus Group
Fit Criteria:	Service composition during design-time is possible.
Developer Satisfaction:	high
Developer Dissatisfaction:	high

[Hydra-114] Semantic enabling of device web services	
Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	Middleware should be able to attach semantic descriptions to device web services based on device ontology.
Source:	WP6 SoA Focus Group
Fit Criteria:	7 of 10 devices are semantically enabled.
Developer Satisfaction:	very high
Developer Dissatisfaction:	high

[Hydra-119] Domain modelling support

Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	The middleware and IDE should be able to interface with application domain frameworks representing core concepts and functions of specific application domains. These could in the most basic form be represented by UML Profiles, or domain ontologies.
Source:	WP6 MDA focus group
Fit Criteria:	The Hydra IDE supports at min 2 defined domain modelling frameworks.
Developer Satisfaction:	high
Developer Dissatisfaction:	high
Dependencies:	117

[Hydra-120] Multiple Device Virtualisations	
Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	It should be possible to have several different views/virtualisations of a device depending on context and applications.
Source:	WP6 MDA Focus Group
Fit Criteria:	At least 2 different virtualisations are provided
Developer Satisfaction:	high
Developer Dissatisfaction:	high

[Hydra-129] Support for Semantic Web Standards for Device Communication	
Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	Middleware should support different semantic web standards, including OWL-S, WSMO, and selected parts of WS-*
Source:	WP SoA Focus Group
Fit Criteria:	Support for at least OWL-S and WSMO
Developer Satisfaction:	high
Developer Dissatisfaction:	high

[Hydra-376] Security requirements must be part of the Hydra MDA	
Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	Security must be defined to be resolved semantically
Source:	WP 6 Focus group Kosice
Fit Criteria:	Security model can be defined semantically
Developer	high

Satisfaction:	
Developer Dissatisfaction:	high

9.2.2 Internal Components

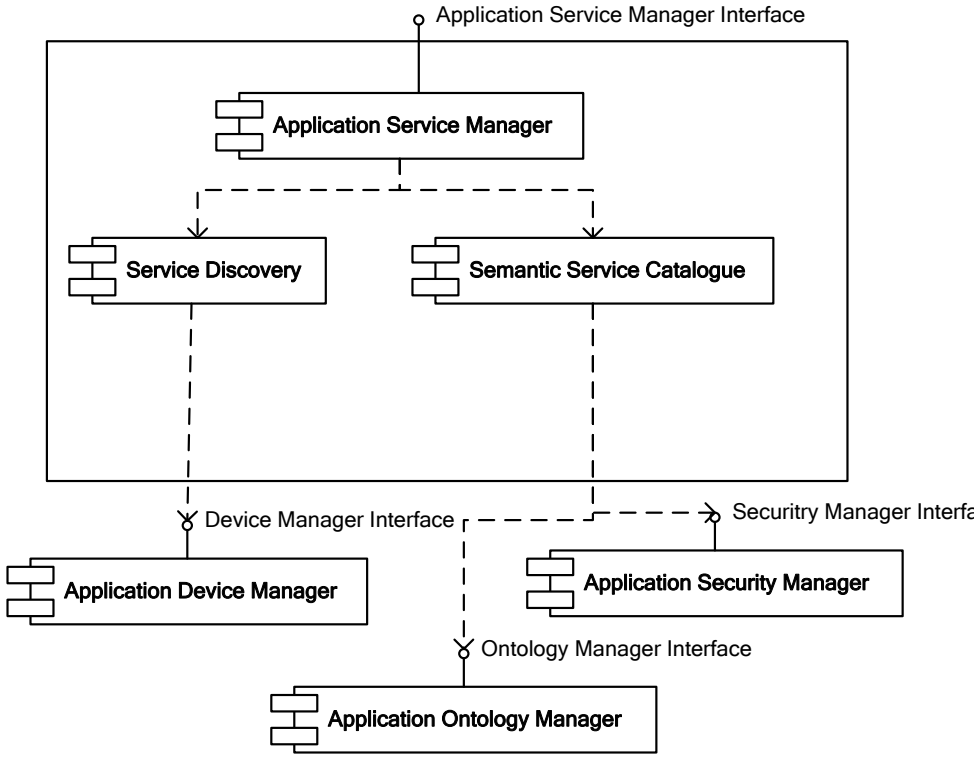


Figure 33: Application Service Manager

9.2.2.1 Service Discovery Module

Purpose

The Service Discovery Module discovers new services in the Hydra network.

Main Functionalities

- Discovering services
- Managing semantic service descriptions and annotations.

Description

The Service Discovery Module discovers new services in the Hydra network; it provides a developer with a service perspective on their device applications. The service discovery process is less complicated than the device discovery process. Services can only be discovered if a device that offers the service has been discovered.

- Application Service Manager asks DAC for the device it has discovered.
- For each device the Application Service Manager asks for its service description.

- Application Service Manager checks the SA-WSDL annotations of the service descriptions and queries the Ontology manager about more information regarding the service.
- It updates the service catalogue with the newly discovered service.

9.2.2.2 Semantic Service Catalogue

Purpose

The Service Application Catalogue keeps track of all services that are currently associated with an active device within one application. It is a service view on the Device Ontology and a current set of (discovered) services.

Main Functionalities

- Maintains a database of discovered services for an application
- Provides a HID query interface for other managers
- Provides a search function over current DAC services

Description

The Semantic Service Catalogue keeps track of and manages all of the services offered within an application. It can be queried for existing services and provides service interfaces for invocation. The catalogue is based on service descriptions in the device ontology.

9.3 Application Orchestration Manager

The Application Orchestration Manager provides support for composite services and workflows. It is an execution engine for the Hydra Device Orchestration Language ("DOLL"). The main purpose for Application Orchestration Manager in this iteration is to focus on energy efficiency aspects. Therefore DOLL has been specialised into an energy policy language.

Main Functions:

- Executing call sequences consisting of invocations of Device services
- Providing interpretation, execution and monitoring of energy policies.

9.3.1 Related WP6 requirements

[Hydra-113] Composition (of services and devices)	
Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	In order to enhance or replace application level functions it will be useful to be able to compose services and devices from different providers and/or manufacturers into high level services/devices
Source:	WP6 MDA Focus Group, WP6 eHealth Focus Group
Fit Criteria:	Service composition during design-time is possible.
Developer Satisfaction:	high
Developer Dissatisfaction:	high

[Hydra-392] Rules for selection of alternative devices

Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	The developer user should be able to specify how devices can replace or complement each other. This is relevant in situations when a device fails and another device exists which can provide a replacement service, or, when different levels of quality of service are available.
Source:	WP6 eHealth focus group
Fit Criteria:	In the SDK, contructs are available that allow the developer to specify rules for when and how devices and services can be interchanged and combined.
Developer Satisfaction:	neutral
Developer Dissatisfaction:	neutral

9.3.2 Internal Components

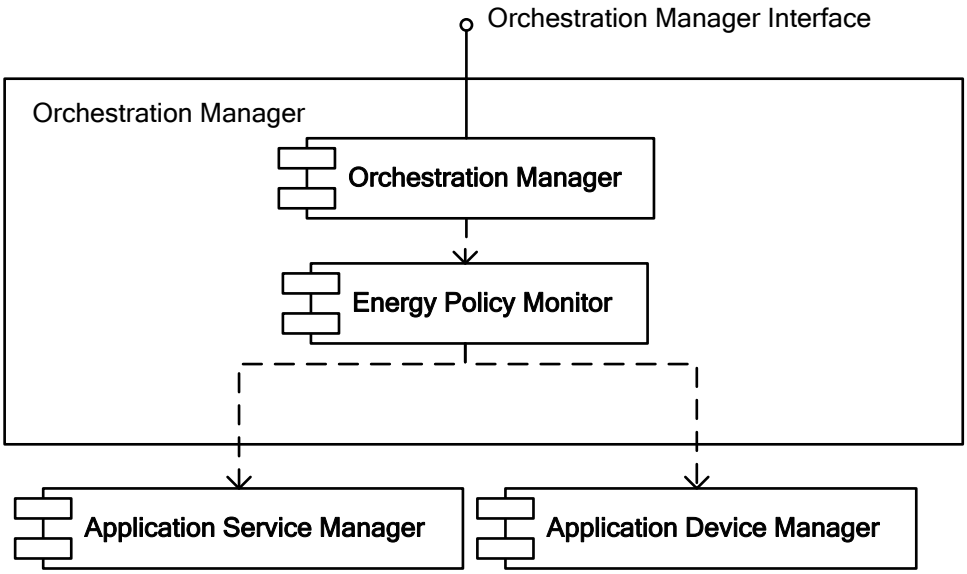


Figure 34: Application Orchestration Manager

9.3.2.1 Energy Policy Monitor

Purpose

In order to monitor and coordinate a collection of devices from en energy perspective, a developer can also specify energy policies on a system-level. These policies typically operate on a set of Hydra devices, which are related in different policy rules.

Main Functionalities

- Interpret energy policy
- Check energy constraints and execution actions
- Monitor and prevent service execution if energy policy is violated

Description

Hydra middleware provides both device developers and solution developers with support to control the energy consumption of their devices, thus paving the way for new energy efficient applications. This support consists of

- *Energy Profiles*, which describe the energy consumption characteristics of individual devices.
- *Energy Policies*, divided into,
    - *Device Energy policies*, specifying operational constraints in order to control the runtime aspects of energy consumption for a device.
    - *System-level energy policies*, which specify run-time constraints for energy consumption over sets of devices.

The system-level policies are managed and executed by the Application Orchestration Manager.

The energy policy monitor interprets energy policies and executes a set of services depending on the policy. Devices can be selected by explicit reference to name/id or by selection criteria expressed over their Energy Profiles and other device descriptions in the device in the Device Ontology.

Different categories of rules can be specified in the policy including device dependency constraints and various run-time consumption restrictions. Examples of Device dependencies are rules specifying replacement devices, in case of device failure or other unavailability, or, rules for mutual exclusion of usage, preventing two sets of devices to be used simultaneously. Consumption rules include the specification of thresholds for overall consumption or for subsets of devices, and the actions taken such as disabling devices.

9.4 Application Ontology Manager

One of the key components in the Hydra middleware is the Device Ontology, where all meta-information and knowledge about devices and device types are stored. The purpose of the Application Ontology Manager is to provide an interface for using the Device Ontology. This manager could possibly also maintain other models in addition to devices.

Main Functions:

- Device description & annotation
- Parsing & annotation of device description
- Parsing & annotation of device service descriptions
- Device search/query function
- Device services search/query function
- Run-time ontology update
- Reasoner module

This manager also maintains the run-time instances of hydra devices.

9.4.1 Related WP6 requirements

[Hydra-91] Any Hydra device should have an associated description	
Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	For management, search and discovery purposes, all Hydra enabled devices should be described (classified) according to the Hydra device ontology.
Source:	WP6 MDA scenario
Fit Criteria:	Any device associated to a Hydra application is also included in the Hydra device ontology, and its description can be retrieved.
Developer Satisfaction:	high
Developer Dissatisfaction:	high

[Hydra-101] Manual device ontology definition	
Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	The developer should be able to define and extend device ontologies. The IDE is required to provide descriptors for devices and device classes
Source:	WP6 MDA Scenario Focus Group
Fit Criteria:	The Hydra IDE supports the manual editing of devices in the framework of device ontology.

Developer Satisfaction:	low
Developer Dissatisfaction:	high

[Hydra-103] Automatic device ontology construction	
Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	The construction of device ontology should be facilitated through finding and parsing product or device descriptions to annotate and produce ontology entries. The component should handle different input formats like Word, PDF, HTML, databases.
Source:	St. Augustin Workshop
Fit Criteria:	5 of 10 device descriptions can be successfully processed
Developer Satisfaction:	very high
Developer Dissatisfaction:	high

[Hydra-108] Device discovery	
Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	Middleware should be able to detect new device that enters the network
Source:	St. Agustin
Fit Criteria:	7 of 10 devices are discovered
Developer Satisfaction:	very high
Developer Dissatisfaction:	high

[Hydra-110] Device Categorisation in runtime	
Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	Middleware should after discovery of device be able to categorise a device based on device ontology information.
Source:	WP6 MDA Focus Group
Fit Criteria:	7 of 10 devices are correctly categorised and described.
Developer Satisfaction:	very high
Developer Dissatisfaction:	high
Dependencies:	101

[Hydra-117] Hydra component ontology	
Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	In order to support and ease the management of the Hydra middleware, the Hydra middleware

	components should be described and mapped to a corresponding Hydra middleware software component ontology.
Source:	WP6 MDA focus group
Fit Criteria:	All Hydra components can be identified through a software component ontology
Developer Satisfaction:	high
Developer Dissatisfaction:	high

[Hydra-119] Domain modelling support	
Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	The middleware and IDE should be able to interface with application domain frameworks representing core concepts and functions of specific application domains. These could in the most basic form be represented by UML Profiles, or domain ontologies.
Source:	WP6 MDA focus group
Fit Criteria:	The Hydra IDE supports at min 2 defined domain modelling frameworks.
Developer Satisfaction:	high
Developer Dissatisfaction:	high
Dependencies:	117

[Hydra-126] Automatic Device ontology updates	
Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	The device ontology should automatically update its device descriptions.
Source:	WP6 MDA Focus Group
Fit Criteria:	The device ontology can detect device updates and handle that in 7 of 10 cases.
Developer Satisfaction:	high
Developer Dissatisfaction:	high

Developer Satisfaction:	very high
Developer Dissatisfaction:	very high

[Hydra-376] Security requirements must be part of the Hydra MDA	
Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	Security must be defined to be resolved semantically

Source:	WP 6 Focus group Kosice
Fit Criteria:	Security model can be defined semantically
Developer Satisfaction:	high
Developer Dissatisfaction:	high

9.4.2 Internal Components

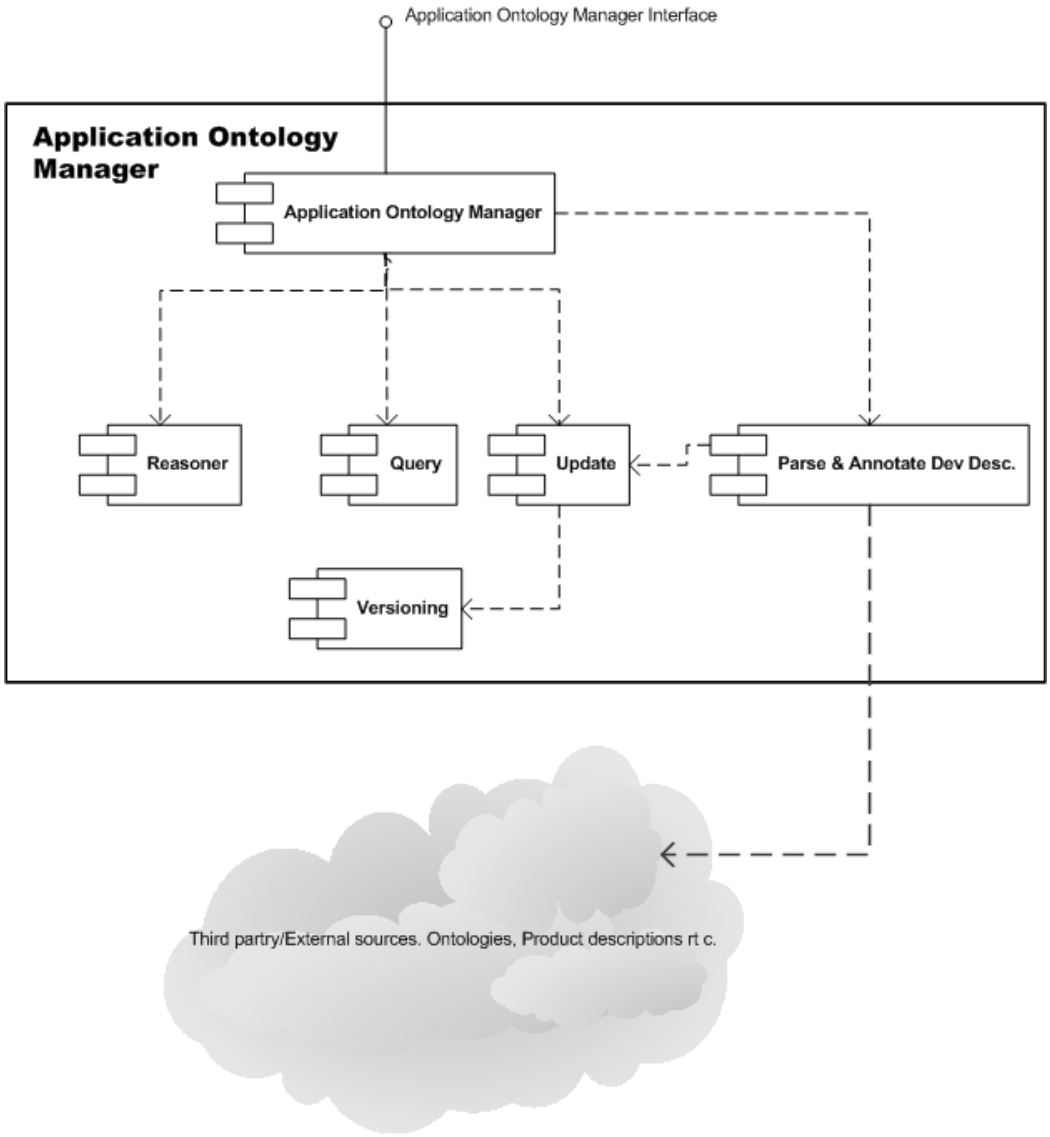


Figure 35: Application Ontology Manager

9.4.2.1 Reasoner

The reasoner module is responsible for reasoning about devices and their status and provides inference mechanisms for instance to conclude what type of device has entered the network.

9.4.2.2 Query module

The query module allows for retrieving information regarding devices and their capabilities.

9.4.2.3 Update module

The update module allows entering of new information, deletion and changes to the ontology at both design time and run time.

9.4.2.4 Versioning

The versioning module is responsible for managing different version of the ontology. This includes different versions of devices and services.

9.4.2.5 Parse & Annotate

The parse & annotate modules is responsible for automatically update the ontology with new device types. It does so by analyzing and annotates existing device and product descriptions which are fed into the ontology.

9.5 Application Diagnostics Manager

The purpose of the Application Diagnostics Manager (aka Self-* Manager) is to monitor the system conditions and state. It will be responsible for error detection and logging of device events. The Diagnostics Manager will be an important component in providing the self-* properties of Hydra (i.e., self-configuration, self-adaptation, self-diagnosis, and self-protection). Completely reliable failure detection is impossible in a distributed system with the characteristics of Hydra, so the Diagnostics Manager will need to work with imperfect failure detectors.

Main Functions:

- Systems diagnostics (e.g., a device is dead/ doesn't respond)
  - dead/live lock detection
  - software failure
  - hardware failures
  - network failures
- Device Diagnostics (device responds but...)
  - service failure
  - device status reports
- Application diagnostics / Monitoring
  - global resource consumption
  - overall property use (e.g., room is too warm)
- Logging
- Self-adaption
  - QoS based adaptation
  - Switching of communication protocols
  - Energy awareness for adaptation
- Self-configuration
  - QoS based configuration
  - Energy awareness for configuration
- Self-management planning
  - service selection based on multiple QoS requirements
  - Multiple planning algorithm support

9.5.1 Related WP6 requirements

[Hydra-91] Any Hydra device should have an associated description	
Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6

Rationale:	For management, search and discovery purposes, all Hydra enabled devices should be described (classified) according to the Hydra device ontology.
Source:	WP6 MDA scenario
Fit Criteria:	Any device associated to a Hydra application is also included in the Hydra device ontology, and its description can be retrieved.
Developer Satisfaction:	high
Developer Dissatisfaction:	high

[Hydra-98] Detection of device failures	
Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	The system should be able to detect malfunctioning devices in order to be robust.
Source:	WP6 MDA focus group
Fit Criteria:	Malfunctioning devices are detected in 8 out of 10 cases.
Developer Satisfaction:	neutral
Developer Dissatisfaction:	neutral

9.5.2 Internal Components

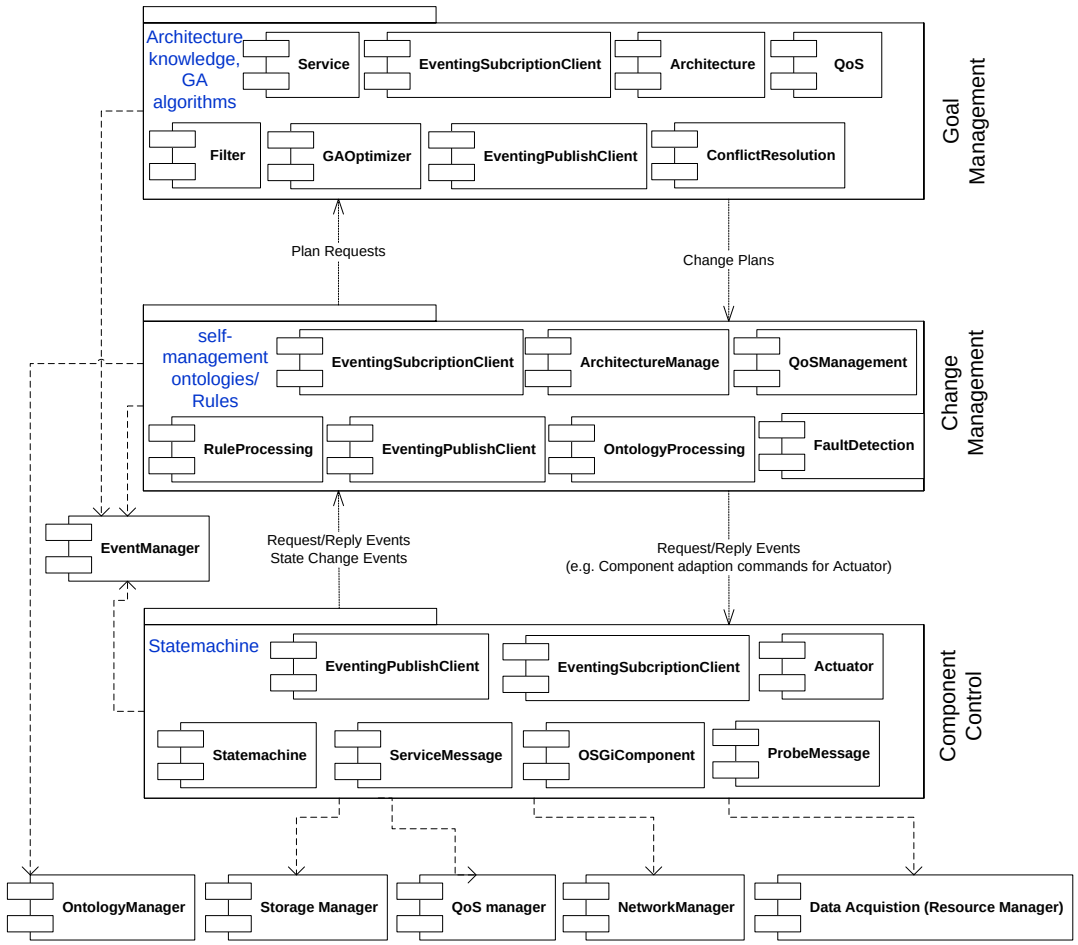


Figure 36: Application Diagnostics Manager

9.5.2.1 Device Status

The Device Status module is responsible for finding out the status of a device and if there are any malfunctions detected. This component should be coordinated with the device state machine running on Resource Manager (or Data Acquisition component) component in order to get all the interested information. This is modeled in StateMachine component in the Component control layer.

9.5.2.2 Log Facility

The Log Facility is used to log all events and interactions between devices. This is used by several other modules to implement their functionality. The log can also be used to detect different erroneous states. The whole log facility will be coordinated with the Storage manager, and currently we log 3 historical results of in the State Machine ontology.

9.5.2.3 Fault Detection

This component will execute rules or rule sets to discover if there is any malfunctioning or strange behavior in the system. Recovery actions can also be published or taken in order to achieve self-managing. The fault detection is realized when the diagnosis rules (detailed in D4.3 and D4.8.) executed.

9.5.2.4 Device Monitoring and management

This feature is used to for instance by monitoring the resource usage of certain devices, especially the battery, memory consumption to achieve power awareness. This reporting of critical resource changes is realized through the Data acquisition component to be implemented. The management

layer is then conduct related management to check whether to switch transportation protocols, and other rules execution. Then the Change management layer is used to process rules or rule sets to monitor devices in order to be preemptive to avoid errors and malfunctions,

9.5.2.5 Communication Monitoring and management

This component is used to conduct packet sniffing on the host running the Web Services and then can be used to make decisions on the working status of the device. This is realized with IPSniffer (Flamencoprobe) ontology and the corresponding monitoring rules as detailed in D4.3 and D4.8.

9.5.2.6 QoS Monitoring and planning

This feature needs to be implemented within the QoS management features. It is used to conduct QoS based self-management. This component is to be implemented by the QoS manager. If QoS is changed, which can then trigger the planning layer (currently implemented using genetic algorithms).

9.5.2.7 Architecture Monitoring and its change management

This component is realized through the OSGiComponent component and its associated OSGiComponent ontology. This will monitor the component changes (introducing services changes), the application architecture are then monitored.

9.6 Device Device Manager

The Device Device Manager handles service requests and manages the responses. The Device Device Manager class is a generic class which is used as the base class for all Hydra Device Managers.

Main Functions:

- Mapping of requests to the services offered by Device Service Manager
- Generation of response
- Advertising Hydra device descriptions including services
- Monitor device energy policies
- Provides memory services for event and state logging
- Provides location services (location data related to device).

9.6.1 Related WP6 requirements

[Hydra-91] Any Hydra device should have an associated description	
Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	For management, search and discovery purposes, all Hydra enabled devices should be described (classified) according to the Hydra device ontology.
Source:	WP6 MDA scenario
Fit Criteria:	Any device associated to a Hydra application is also included in the Hydra device ontology, and its description can be retrieved.
Developer Satisfaction:	high
Developer Dissatisfaction:	high

[Hydra-92] Rule-based configuration of devices	
Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	The possibility for the developer to specify device behaviour using rules. It should be possible to derive and re-use rules from pre-existing or generic rule sets for application domains. Possibility to hide device specific details.
Source:	WP6 MDA Focus Group and WP6 eHealth focus group
Fit Criteria:	The functionality (services) of a device is accessible (by user or application) thru a rule-based interface.
Developer Satisfaction:	high
Developer Dissatisfaction:	high

[Hydra-108] Device discovery	
Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	Middleware should be able to detect new device that enters the network
Source:	St. Agustin
Fit Criteria:	7 of 10 devices are discovered
Developer Satisfaction:	very high
Developer Dissatisfaction:	high

[Hydra-109] Device Virtualization	
Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	The complexity of devices may be hidden, or simplified, by means of virtual device interfaces; these would correspond to "views" on device descriptions as provided by the Hydra device models (ontologies).
Source:	WP6 MDA scenario focus group
Fit Criteria:	An existing virtualization can be used to find exactly one proper Hydra device.
Developer Satisfaction:	neutral
Developer Dissatisfaction:	neutral

[Hydra-111] Dynamic Web Service Binding	
Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	Middleware should be able to after device discovery and categorisation expose a new device as a web service that can be called without re-compilation.
Source:	WP6 SoA Focus Group
Fit Criteria:	New devices are callable and controllable in 7 out of 10 cases.
Developer Satisfaction:	very high

Developer Dissatisfaction:	very high

[Hydra-114] Semantic enabling of device web services	
Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	Middleware should be able to attach semantic descriptions to device web services based on device ontology.
Source:	WP6 SoA Focus Group
Fit Criteria:	7 of 10 devices are semantically enabled.
Developer Satisfaction:	very high
Developer Dissatisfaction:	high

[Hydra-120] Multiple Device Virtualisations	
Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	It should be possible to have several different views/virtualisations of a device depending on context and applications.
Source:	WP6 MDA Focus Group
Fit Criteria:	At least 2 different virtualisations are provided
Developer Satisfaction:	high
Developer Dissatisfaction:	high

[Hydra-376] Security requirements must be part of the Hydra MDA	
Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	Security must be defined to be resolved semantically
Source:	WP 6 Focus group Kosice
Fit Criteria:	Security model can be defined semantically
Developer Satisfaction:	high
Developer Dissatisfaction:	high

9.6.2 Internal Components

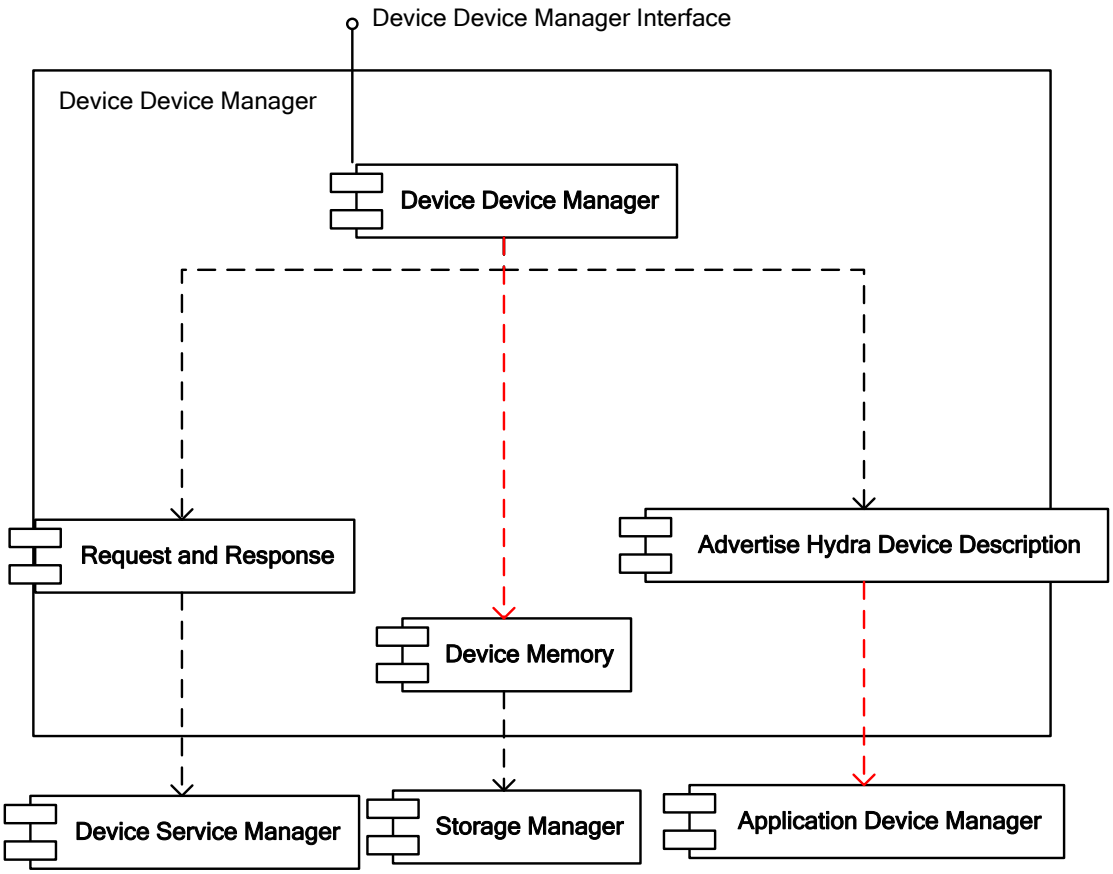


Figure 37: Device Device Manager

9.6.2.1 Request and Response Mapping

Purpose

Provides the interface with the Device Service Manager

Main Functionalities

- Map request to Device Service
- Translate Device Service Manager response to response to the external request.

Description

This module maps a request from an outside caller to an internal service in the device.

9.6.2.2 Advertise Hydra Device Description

Purpose

This module is responsible for broadcasting the existence of the device to the outside world. It will support advertising thru several protocols, at least UPnP (Universal Plug and Play).

Main Functionalities

- Create UPnP broadcast advertise message
- Create Hydra device model description

Description

This module can advertise and provide the service description of the device.

9.6.2.3 Device Memory

Purpose

Provides a virtual memory for the device.

Main Functionalities

- Storing of event log
- Storing of state variables and overall state.

Description

Provides a virtual memory for the device, using the Hydra Storage Manager

9.7 Device Service Manager

The Device Service Manager implements a service interface for physical devices. It should normally not be used directly by any other manger than the Device managers.

Main Functions:

- Maps services to physical device operations
- Maps (physical) device events to Hydra enabled events

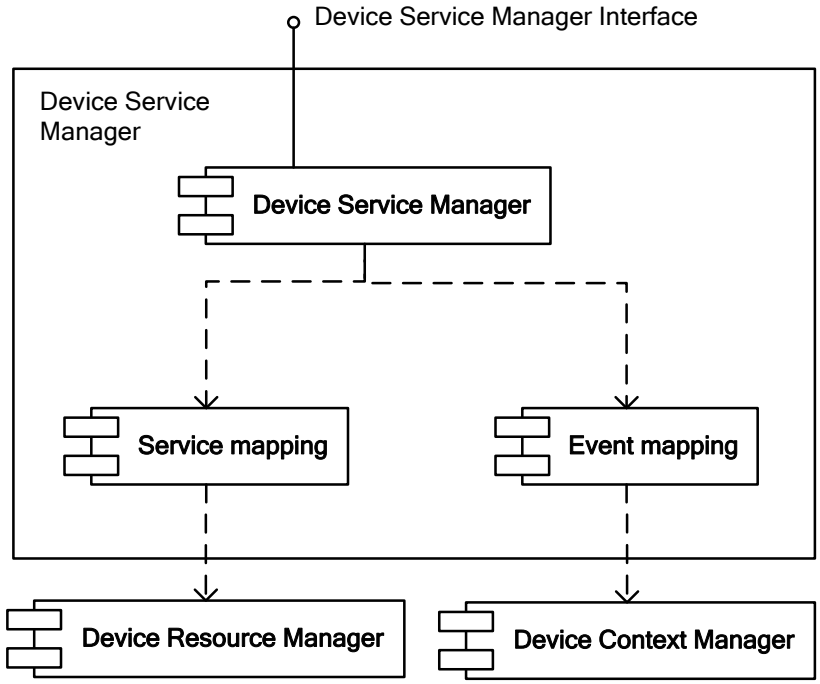
9.7.1 Related WP6 requirements

[Hydra-120] Multiple Device Virtualisations	
Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	It should be possible to have several different views/virtualisations of a device depending on context and applications.
Source:	WP6 MDA Focus Group
Fit Criteria:	At least 2 different virtualisations are provided
Developer Satisfaction:	high
Developer Dissatisfaction:	high

[Hydra-376] Security requirements must be part of the Hydra MDA	
Status:	Part of specification
Requirement Type:	Functional
Work package:	WP6
Rationale:	Security must be defined to be resolved semantically
Source:	WP 6 Focus group Kosice
Fit Criteria:	Security model can be defined semantically
Developer	high

Satisfaction:	
Developer Dissatisfaction:	high

9.7.2 Internal Components



9.7.2.1 Service Mapping

This module maps device service request to internal device operations. One device can have several service mappings.

9.7.2.2 Event Mapping

This module handles physical device events and maps them into Hydra-events.

10. References

[AMIGO, 2006] IST Amigo Project (2006). Amigo middleware core: Prototype implementation and documentation, deliverable 3.2. Technical report, IST-2004-004182.

[Bailey, 2005] J. Bailey et al., Web and Semantic Web Query Languages: A Survey, LNCS 3564, Norbert Eisinger, Jan Maluszynski (editor(s)), 2005

[Chandrakasan, 2001] Amit Sinha and Anantha Chandrakasan, Dynamic Power Management in Wireless Sensor Networks, IEEE Design & Test of Computers, Vol. 18, No. 2, March-April 2001

[Chen, 2005] H. Chen, T. Finin, and A. Joshi. The SOUPA Ontology for Pervasive Computing. Ontologies for Agents: Theory and Experiences, 2005.

[DCMI, 2007] DCMI. (2007). "The Dublin Core Metadata Initiative: http://dublincore.org/." fromhttp://dublincore.org/.

[FIPA 2002] FIPA Device Ontology Specification, Foundation for intelligent physical agents, 2002.

[Flury, 2004] T. Flury, G. Privat, and F. Ramparany. OWL-based location ontology for context-aware services. Proc. Artificial Intelligence in Mobile Systems, Nottingham (UK), pages 52–58, 2004.

[Hydra, 2006] Hydra (2006). D2.2 Initial Technology Watch Report. Hydra Project Deliverable, IST project 2005-034891.

[Hydra, 2007] Hydra (2007). D6.1 Quality Attribute Scenarios. Hydra Project Deliverable, IST project 2005-034891.

[Hydra, 2007b] Hydra (2007). D4.2 Embedded Service SDK Prototype and Report. Hydra Project Deliverable, IST project 2005-034891.

[Hydra, 2007c] Hydra (2007). D7.2 Draft of Virtualisation Ddesign Specification. Hydra Project Deliverable, IST project 2005-034891.

[Matheus, 2005] C. Matheus. Using ontology-based rules for situation awareness and information fusion. W3C Work. on Rule Languages for Interoperability, 2005.

[McGuinness, 2004] D.L. McGuinness, F. van Harmelen, OWL Web Ontology Language Overview, W3C Recommendation, 2004

[Oberle, 2006] Oberle, D. (2006). Semantic Management of Middleware, Springer.

[Oconnor, 2007] M. J. O'Connor, S. W. Tu, A. K. Das, and M. A. Musen. Querying the semantic web with swrl. In The International RuleML Symposium on Rule Interchange and Applications (RuleML-2007), Orlando, FL, Oct. 2007. LNCS, Springer-Verlag.

[OWL-S, 2004] D. Martin et al., OWL-S: Semantic Mark-up for Web Services, http://www.daml.org/services/owl-s/1.1/overview/, 2004

[Plas, 2006] D.-J. Plas, M. Verheijen, H. Zwaal, and M. Hutschemaekers. Manipulating context information with swrl. I/RS/2005/117, Freeband/A-MUSE/D3.12, 2006.

[RDF, 2007] RDF. (2007). "The Resource Description Framework (RDF): http://www.w3.org/RDF/." from http://www.w3.org/RDF/.

[SAWSDL, 2007] SAWSDL (2007). Semantic Annotations for WSDL and XML Schema. W3C Recommendation. J. Farrell and H. Lausen, W3C.

[Schmidt, 2002] Schmidt, D. C. (2002). "Middleware for real-time and embedded systems." Communications of the ACM **45**(6): 43-48.

[SPARQL, 2007] E. Prud'hommeaux, A. Seaborne, SPARQL Query Language for RDF, W3C Proposed Recommendation, 2007

[SWRL, 2004] SWRL, 2004 I. Horrocks, et al., SWRL: A Semantic Web Rule Language Combining OWL and RuleML, W3C Member Submission, 2004

[Zhang, 2007] Weishan Zhang, Klaus Marius Hansen, Kristian Ellebæk Kjær. Exploring OWL/SWRL based Diagnosis in aWeb Service-based Middleware for Embedded and Networked Systems. Submitted to ICECCS2008

[Zheng, 2004] Jianliang Zheng and Myung J. Lee, Will IEEE 802.15.4 Make Ubiquitous Networking a Reality?: A Discussion on a Potential Low Power, Low Bit Rate Standard, Communications Magazine, IEEE, Vol. 42, No. 6. (2004), pp. 140-146.