Contract No. IST 2005-034891

Hydra

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

D6.2 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 **2007-12-21- version 1.0**
Coordinating Partner: C International Ltd.

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

Dissemination Level: Confidential

Document file: D6 2 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	2007-08-28	TOC based on Aarhus Ontology WS
0.14		2007-10-19	Structure revised
0.15	Mathias Axling, Matts Ahlsen	2007-10-26	Sec 4 + 7
0.16	Paolo Sperandio	2007-10-29	Sec 5, Wireless
0.17	Matts Ahlsén, Mathias Axling, Peter Rosengren, Peeter Kool, Paolo Sperandio	2007-11-02	1st draft: sec 1,2,3,4,7
0.18	Klaus Marius Hansen, Weishan Zhang	2007-11-12	Update of 4.1: code generation + context modeling, 5.1 and 5.4
0.19	Peter Kostelnik, Matts Ahlsen	2007-11-20	Update of Device Ontology
0.20	Peter Rosengren. Mathias Axling, Matts Ahlsén	2007-11-25	Services and Device application catalogue
0.21	Peter Kostelnik, Peter Butka, Matts Ahlsén	2007-11-27	Standards & tools
0.22	Stephan Engberg, Morten Harning	2007-11-30	Semantic resolution added
0.23	Peter Rosengren. Mathias Axling, Matts Ahlsén, Peeter Kool	2007-12-03	Update ch. 4-5-6
0.24	Peter Rosengren, Matts Ahlsén, Peeter Kool, Paolo Sperandio	2007-12-10	Update ch 5
0.35	Peter Rosengren, Matts Ahlsén, Peeter Kool	2007-12-12	Update ch 4
0.40	Peter Rosengren, Matts Ahlsén, Peeter Kool	2007-12-13	Update ch 6 with new manager descriptions
0.43	Peter Rosengren, Matts Ahlsén, Peeter Kool	2007-12-13	Update ch 4-6, formatting, revision of figures
0.50	Peter Rosengren, Matts Ahlsén, Peeter Kool	2007-12-14	Revised Managers descriptions for Application Elements
0.60	Peter Rosengren, Matts Ahlsén, Peeter Kool	2007-12-15	Revised Managers descriptions for Device Elements. Added Application Orchestration Manager
0.70	Peter Rosengren, Matts Ahlsén, Peeter Kool, Weishan Zhang	2007-12-17	Revised manager descriptions, updated Diagnostics Manager
0.80	Peter Rosengren, Matts Ahlsén, Peeter Kool	2007-12-17	Spell-checked, updated Future Work.
0.85	Peter Rosengren, Matts	2007-12-20	Update with respect to review

	Ahlsén, Peeter Kool		comments
0.90	Peter Rosengren, Matts Ahlsén, Peeter Kool, Mathias Axling	2007-12-21	2nd revision after internal review
1.0	Peter Rosengren, Matts Ahlsén, Peeter Kool, Mathias Axling	2007-12-21	Final version submitted to the European Commission

Internal review history:

Reviewed by	Date	Comments
Daniel Thiemert (University of Reading)	2007-12-19	Approved with comments
Pablo Rafael Antolin (TID)	2007-12-20	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, to which the underlying communication layer is transparent. 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

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 middleware layer should provide adaptations to whatever interface the devices offer. To achieve this, Hydra aims to be able to describe the capabilities of the devices (ontologies) in such way that an automatic agent can understand these capabilities and use them. Once the semantics describing the model of the other device has been found, then the device capabilities could be accessed.

This document (D6.2) describes the Semantic Model Driven Architecture of HYDRA the objective of which is to facilitate application development and to promote semantic interoperability for services and devices. The semantic MDA of HYDRA includes a set of models, i.e., ontologies, and describes how these can be used both in design-time and in run-time.

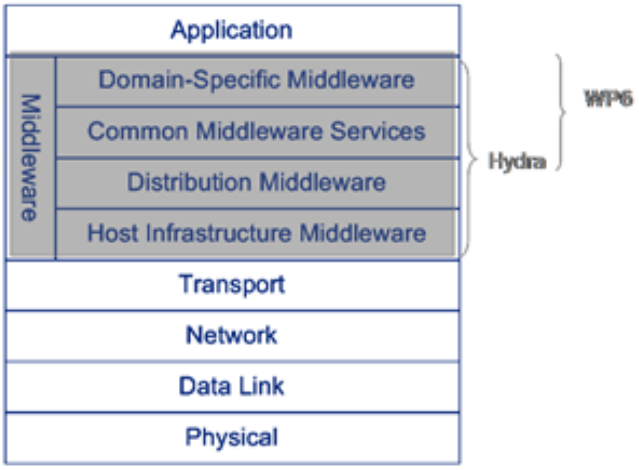


Figure 1: The middleware stack and the role of WP 6.

Figure 1 shows HYDRA and the Semantic MDA (WP6) in relation to generic middleware reference model [Schmidt, 2002].

1.3 Hydra Innovations and Contributions

Hydra's technological innovations in semantic MDA will be achieved 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 "low-level 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" summaries 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:	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

	• Web services stack • Fast evaluation of WS • Semantic stack		
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

1.3.1 Semantic Web

Web technologies are shifting from rendering information in a format for human interpretation, towards providing an automated environment for delivering a wide variety of e-commerce and business-to-business services and applications such as the ones envisioned in Hydra.

Such services and applications will communicate and interoperate in a world composed of Web-accessible programs and databases, and interface wirelessly with many smart devices and sensors. These shifts have the potential to change significantly the way we communicate, co-operate, and organise our commercial and personal relationships.

The Semantic Web is fundamental to enabling these types of services and applications by providing a universally accessible platform that allows data to be shared and processed by automated tools, and by providing the machine-understandable semantics of data and information that will enable automatic information processing and exchange.

The Semantic Web principles are realized by layers of related Web technologies and standards, commonly depicted as the Semantic Web Layers introduced by the W3C (Figure 2) (aka the semantic web cake).

The Unicode and URI layers make sure that we use international characters sets and provide means for identifying the objects in Semantic Web. The XML layer with namespace and schema definitions make sure we can integrate the Semantic Web definitions with the other XML based standards. With RDF and RDFSchema it is possible to make statements about objects with URI's and define vocabularies that can be referred to by URI's. This is the layer where we can give types to resources and links.

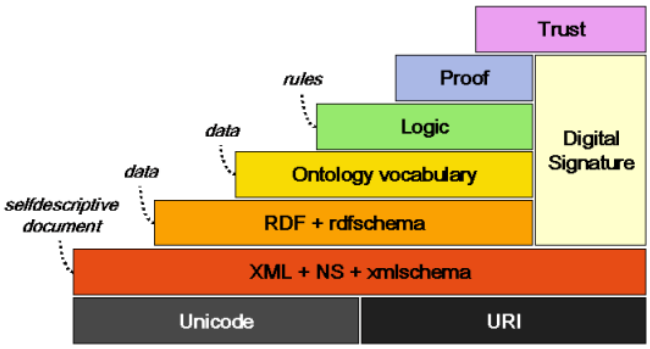


Figure 2: Semantic Web Layers (W3C)

The Ontology layer supports the evolution of vocabularies as it can define relations between the different concepts. With the Digital Signature layer for detecting alterations to documents, these are the layers that are currently being standardized in W3C working groups.

The top layers: Logic, Proof and Trust, are currently being researched and simple application demonstrations are being constructed. The Logic layer enables the writing of rules while the Proof layer executes the rules and evaluates, together with the trust layer mechanism for applications, whether to trust the given proof or not.

OWL is a W3C recommendation [McGuinness, 2004]. The OWL Web Ontology Language 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.

OWL has been designed to meet this need for a Web Ontology Language. OWL is part of the growing stack of W3C recommendations related to the Semantic Web.

- XML provides a surface syntax for structured documents, but imposes no semantic constraints on the meaning of these documents.
- XML Schema is a language for restricting the structure of XML documents and also extends XML with data types.
- RDF, the Resource Description Framework [RDF, 2007], is a family of specifications for a metadata model that is often implemented as an application of XML. The RDF family of specifications is maintained by the World Wide Web Consortium (W3C).
- RDF is a data model for objects ("resources") and relations between them. It provides a simple semantics for this data model, and these data models can be represented in XML syntax.
- RDF Schema is a vocabulary for describing properties and classes of RDF resources, with semantics for generalization-hierarchies of such properties and classes.
- OWL adds more vocabulary for describing properties and classes: among others, relations between classes (e.g. disjointedness), cardinality (e.g. "exactly one"), equality, richer typing of properties, characteristics of properties (e.g. symmetry) and enumerated classes.

The RDF metadata model is based upon the idea of making statements about resources in the form of a subject-predicate-object expression, called a triple in RDF terminology. The subject is the resource being described. The predicate is a trait or aspect about that resource, and often expresses

a relationship between the subject and the object. The object is the object of the relationship or value of that trait.

This mechanism for describing resources is a major component in what is proposed by the W3C's Semantic Web activity: an evolutionary stage of the World Wide Web in which automated software can store, exchange, and utilise metadata about the vast resources of the Web, in turn enabling users to deal with those resources with greater efficiency and certainty. RDF's simple data model and ability to model disparate, abstract concepts has also led to its increasing use in knowledge management applications unrelated to Semantic Web activity.

The Dublin Core Metadata Initiative [DCMI, 2007] is an organization dedicated to promoting the widespread adoption of interoperable metadata standards and developing specialized metadata vocabularies for describing resources that enable more intelligent information discovery systems.

The Universal Description, Discovery and Integration (UDDI) protocol provides a catalogue for web services. UDDI creates a standard interoperable platform that enables companies and applications to quickly, easily, and dynamically find and use Web services over the Internet. UDDI also allows operational registries to be maintained for different purposes in different contexts. UDDI is a cross-industry effort driven by major platform and software providers, as well as marketplace operators and e-business leaders within the OASIS standards consortium. The main contribution of Hydra to the Semantic Web is to bring 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 applications. We will explore and support the use of standards such as WSMO, OWL-S and SAWSDL [SAWSDL, 2007]. This will be further researched in Task 6.5 SoA and Semantic Web Services for Devices, and presented in deliverable D6.3 "Semantic Web Service Design Document".

1.3.2 OMG Model-Driven Architecture

As an OMG process, the MDA represents a major evolutionary step in the way the OMG 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 be in the use of ontologies both for the application developer and the device developer. For the latter we will support an OMG MDA process at design time through the use of ontologies semi-automatic code generation for devices. Ontologies will also be an integral part of the run-time environment, i.e. program execution will be based on the models and descriptions in the ontologies, providing an easy to configure and dynamic extensible middleware.

1.3.3 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 will opt 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 aims to be able to describe the capabilities of the devices (using ontologies) in such way that an automatic agent 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.

1.3.4 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 to obtain 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. This involves the merging of meta-data from multiple sources and may need for complex algorithms being defined during the project. However, reasoners must rely on querying languages over meta-data and there are several initiatives and languages that allow for queries over RDF annotated data: RQL, RDQL and SPARQL. The selection among the aforementioned alternatives will be guided both by 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 these languages).

2. Executive Summary

This workpackage 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 which 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, in Hydra we also aim at providing interoperability at a *semantic level*. Thus, the Hydra middleware must also model services offered by different devices from an applications point of view.

A main contribution of this workpackage to is to bring 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 will be done in close cooperation with WP4 which are investigating 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 will opt for choice b), so it will try to be 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 some automatic way.

In order to achieve this, Hydra aims to be able to describe the capabilities of the devices (ontologies) in such way that an automatic agent can understand these capabilities and use them. Once the semantics describing the model of a device 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 "low-level 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 to obtain complex functionality from atomic operations of heterogeneous end-points (services, interfaces provided by any entity: user agents, servers, devices, etc.).

But 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. This involves the merging of meta-data from multiple sources and may require complex algorithms to be defined during the project.

3. Requirements for the HYDRA Semantic Model-driven Architecture

3.1 User requirements

Below we present the current set of user requirements as produced by the Volere method in workpackage 2.

Table 2: WP6 requirements summary list

Key	Summary	Source	Rationale	Fit Criteria
HYDRA-91	Any HYDRA device should have an associated description	WP6 MDA Scenario Focus Group	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-101	Manual device ontology definition	WP6 MDA Scenario Focus Group	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 a device ontology.
HYDRA-102	Device Ontology with user interface	St. Augustin	Tool that allows browsing, searching, navigating device classes and their capabilities.	Tool for browsing device ontology exists
HYDRA-103	Automatic device ontology construction	St. Augustin Workshop	The IDE should facilitate 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.	7 of 10 device descriptions can be successfully processed
HYDRA-109	Device Virtualization	WP6 MDA scenario focus group	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).	An existing virtualization can be used to find exactly one proper HYDRA device.
HYDRA-110	Device Categorisation	WP6 MDA Focus Group	Middleware should after discovery of device be able to categories a device based on device ontology	7 of 10 devices are correctly categorised and described.

			information.	
HYDRA-112	Dynamic Web Service Generation	WP6 SoA Focus Group	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-114	Semantic enabling of device web services	WP6 SoA Focus Group	Middleware should be able to attach semantic descriptions to device web services based on device ontology.	7 of 10 device are semantically enabled.
HYDRA-117	HYDRA component ontology	WP6 MDA focus group	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.	All HYDRA components can be identified through a software component ontology
HYDRA-119	Domain modelling support	WP6 MDA focus group	The middleware and IDE should be able to host or 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	WP6 MDA Focus Group	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-121	Compiled device ontology	WP6 MDA Focus Group	The device ontology should be compiled to be deployed and used in device discovery process	Possible to compile device ontology
HYDRA-126	Automatic Device ontology updates	WP6 MDA Focus Group	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-139	Knowledge model of hydra middleware	State of the Art	Knowledge model of the whole middleware providing developers with knowledge on all middleware components offers a guidance how ho compose a hydra-based application.	Support for knowledge model based rapid development is available
HYDRA-141	Download and harmonisation of third party device	Hydra D2.2 Initial Technology Watch	Device ontological models describing devices, which will be provided by manufacturers or third	Ontologies from different manufacturers can be used if they are in RDF, OWL or

	ontologies	Report	parties, should be automatically downloaded (updated) and harmonised to ensure the same ontological view. Formal definition of ontologies should be realised using the world wide accepted formats, recommended by W3C, such as RDF, OWL, OWL-S.	OWL-S
HYDRA-143	Model-based reasoning about itself	Hydra D2.2 Initial Technology Watch Report	Rich self knowledge model of the middleware provides the middleware with self-awareness. It enables to reason on middleware status (self-diagnostics, current configuration, optimality of using available resources, estimation response, etc.).	Middleware is able to reason on itself. It is able to detect its status in 9 of 10 cases.
HYDRA-210	Middleware should support different architectural styles	WP6 SoA Focus Group	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-212	Support for a declarative application development paradigm	WP6 SoA Focus Group	A declarative approach can hide complexity of underlying structure and can increase productivity of embedded software development.	More than 50% of the module functionality should be programmable using a declarative approach.
HYDRA-248	Definition of Virtual Devices	WP6 Focus group in Kosice	In order to ensure flexibility, protecting weak devices and manage differentiated access to device and information, the developer or advanced users should be able to define virtual devices that replace/represent physical devices.	Separation of physical and logical device definition. A virtual device can fully replace a physical device
HYDRA-316	Service descriptions should include service semantics	UAAR focus group	To support dynamic (and reflective) systems, it is important to know more than just the syntax of an interface to a discovered service	Semantics approach defined. Service description language uses this approach. Semantic service descriptions are published
HYDRA-359	Device ontology versioning	WP6 MDA Focus Group	The device ontology should be able to handle different versions of a device.	The device ontology can maintain at minimum 2 versions of any single device
HYDRA-	Ability to self-	Hydra D2.2	Rich knowledge model	Middleware is able to adapt its

365	adaptation	Initial Technology Watch Report	enables the middleware to contain a representation of itself and manipulate its state during its execution. This feature should serve as the basis for self-adaptation of the middleware (e.g. reconfiguration of resource usage, triggering the component-based services).	configuration in 60% of identified cases requiring reconfiguration.
HYDRA-376	Security requirements must be part of the device ontology	WP 6 Focus group Kosice	Security must be defined and resolved semantically	Security model part of device ontology
HYDRA-378	Application model must provide the security requirements	WP3 Meeting Kosice - Roundtable discussion - S. Engberg	Application must provide the security requirements on a semantic level in order to resolve if devices are allowed to interact with the application or to allow the middleware to resolve the security in the process	If the application model contains security requirements all requests will be resolved correctly
HYDRA-389	Service browsing in device ontology	WP6 eHealth focus group	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-390	Different views on the device ontology	WP6 eHealth focus group	It should be possible to present a developer user with different perspectives on the device ontology, depending on that users functional needs (e.g., a services perspective, device category perspective. etc.)	At least two different views are available in the ontology browser
HYDRA-392	Rules for selection of alternative devices	WP6 eHealth focus group	The developer user should be able to specify how devices can replace or complement each other. This is relevant in situations when a device fail 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.

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 are reported in deliverable D6.1 [Hydra, 2007].

4. HYDRA approach to Semantic MDA

4.1 Introduction

The semantic model-driven architecture of HYDRA (SeMDA) is based on the application of ontologies and semantic web technologies to support the design of device-oriented networked applications and is also intended as a run-time resource in the execution of device services.

The basic idea behind the HYDRA Semantic MDA is to differentiate between the physical devices and the application´s view of the device. We introduce the concept of *Semantic Devices*. The physical 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 3 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 instances "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.

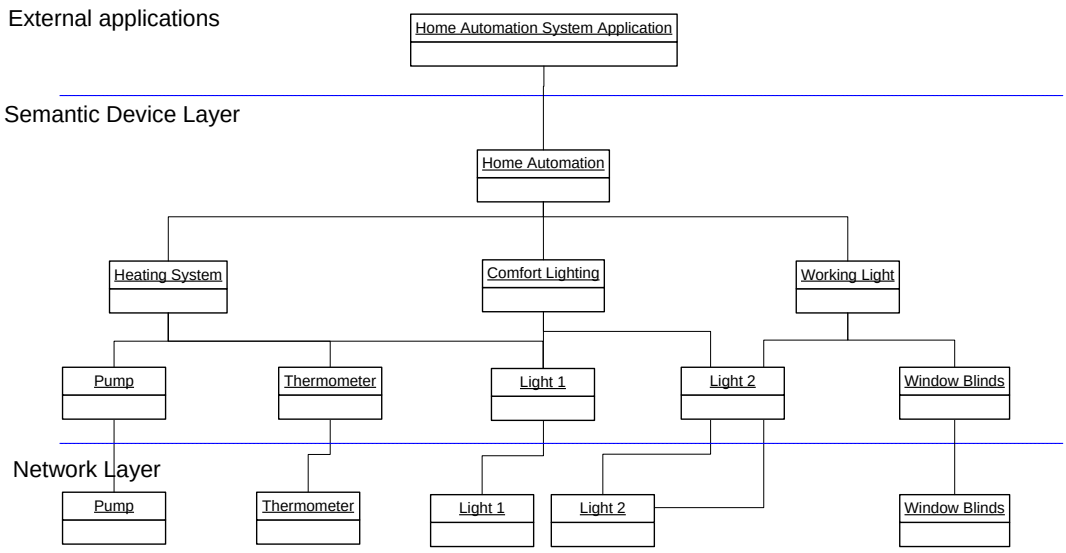


Figure 3: Semantic Devices provide a high-level programming interface.

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.

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. The device ontology is divided into four interconnected modules:

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

The content and structure of the Device Ontology as well as the others ontologies are described in more detail in chapter 5.

To summarise, there are two uses of the semantic MDA in Hydra. 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 Semantic MDA at design-time

4.2.1 Model-driven code generation for physical 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).

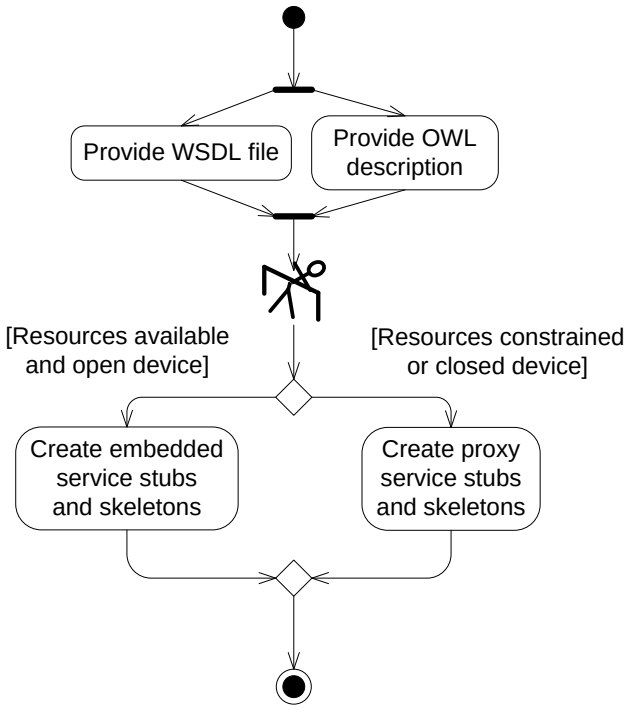


Figure 4: Automatic generation of web service code for devices.

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 "Embedded Service SDK Prototype and Report" for more detail.

4.2.2 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.

We are currently aiming at making the HYDRA SDK available in an object-oriented language environment. 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.

These device objects could be used when creating a semantic device or HYDRA application from the selected devices and services. The services could also be used by a service orchestration engine (however, considering that some applications will be standalone and have a fairly small footprint, this may not be suitable for all HYDRA applications).

An example of this is a heating control system, where service proxies to call the heating system services, 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 personal computers, 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.

How the application uses the Device Ontology should be configurable, so that the middleware supports both standalone applications that only use the Device Ontology at design time as well as applications that always query the Device Ontology for new types of services that match the descriptions.

4.3 Semantic MDA at Run-time

4.3.1 Models for Discovery and the Hydra Device Application Catalogue

A fundamental part in every Hydra-based application is the *Device Application Catalogue* (DAC). This is a runtime component that keeps track of and manages all devices that are currently active within an application. The Hydra Device Application Catalogue serves all Hydra middleware managers with the information and meta data they need regarding devices, their services, and their status.

The Hydra DAC 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.

To illustrate the functionality of the Device Application Catalogue we can view the figure below that shows a graphical browser tool, a *Hydra DAC Browser* that allows browsing of the different devices that have currently been discovered by the Hydra DAC.

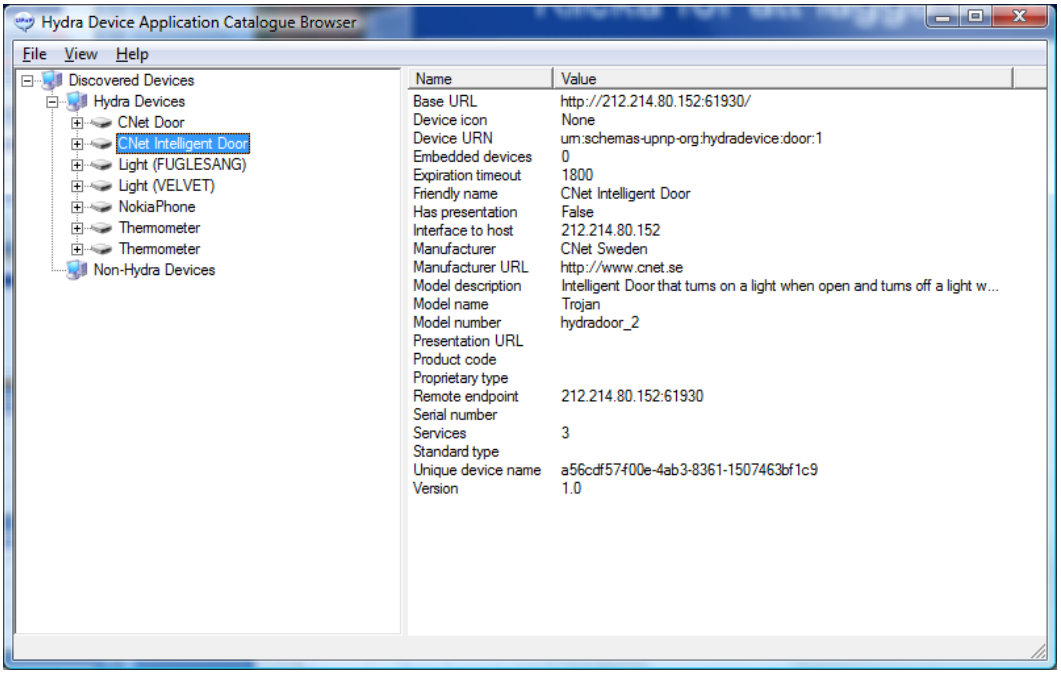


Figure 5: The Hydra DAC Browser

The browser tool uses the Device Application Catalogue to retrieve and display the services offered by a certain device. An example is shown below, where the device "CNet Intelligent Door" provides the services "GetErrorCode", "GetErrorMessage", "GetHydraID" and "GetStatus".

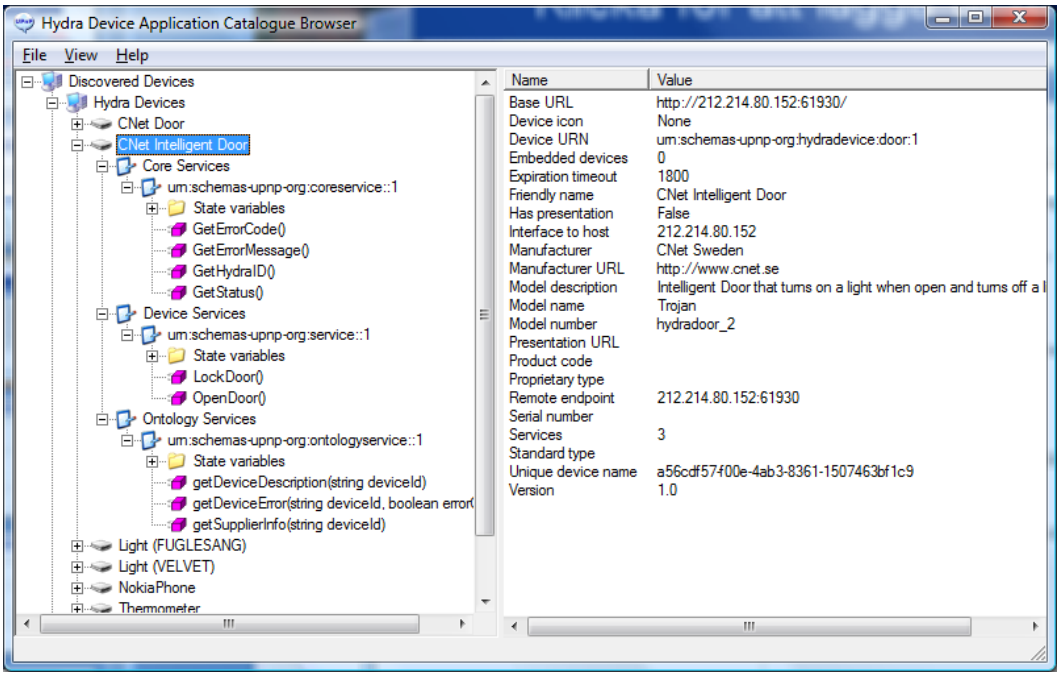


Figure 6: Expansion of services provided by a device.

By retrieving a service interface for the device from the Device Application Catalogue the browser tool can execute a service on the device. In the figure below we see how an SMS service is invoked on the device "NokiaPhone".

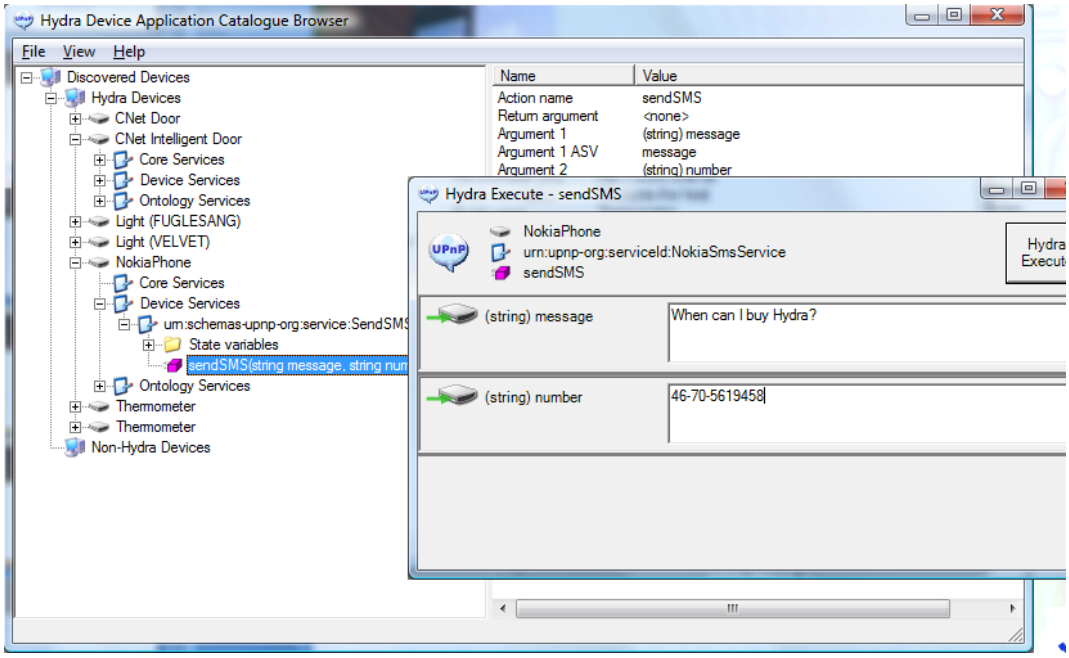


Figure 7: Retrieving and executing services on devices.

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.

When a device is discovered, the device type is looked up in the Device Ontology and if it can be mapped to a Hydra Device (perhaps it will always be mapped to the most generic type of device or "Hydra Unknown Device") it will be 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.

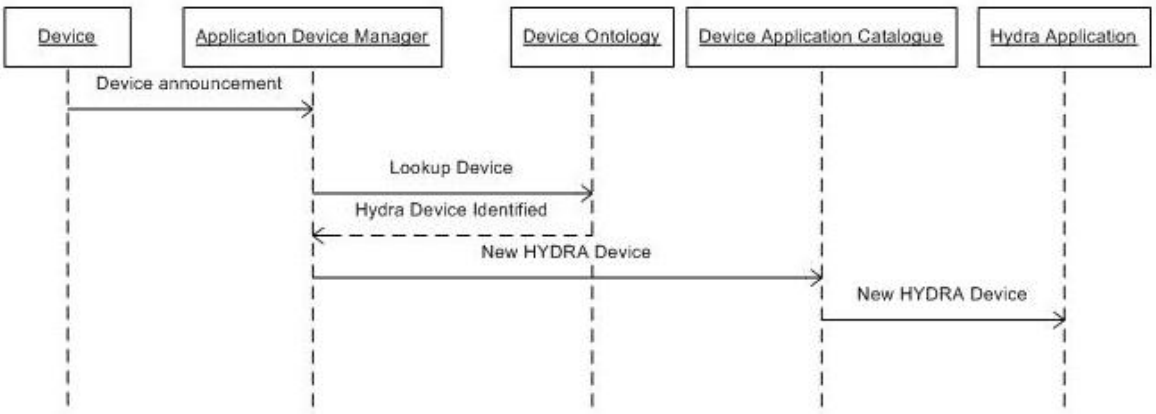


Figure 8: As a device is discovered in the network, its type is resolved against the device ontology, and then entered into the DAC notifying the HYDRA application.

A HYDRA Application may present an external interface so that it can be integrated with other applications and devices. It will do this by announcing itself as a *HYDRA device* with a set of services. This is transparent to other devices, which means that some devices or services used in the application will be composite ones: based on other HYDRA applications that have exposed external interfaces. When such an application is discovered, the applications interested in that type of device and its services will be notified as described above.

The DDK (Device Development Kit) is used to HYDRA-enable limited devices, while the SDK (Software Development Kit) is used to build more advanced HYDRA applications (devices) using other HYDRA devices (new "HYDRA heads" grow out continuously).

4.3.2 Use of models for resolving security requirements

This section is built on a walkthrough of the security model from deliverable D7.2 [Hydra, 2007c] which will be further detailed as part of deliverable D7.3.

The dynamic and networked execution environment of HYDRA requires strong yet adaptable security mechanisms to be in place. In order to establish the ability to securely connect any application/device to any application/device, HYDRA also uses the semantic MDA to define and enforce security policies.

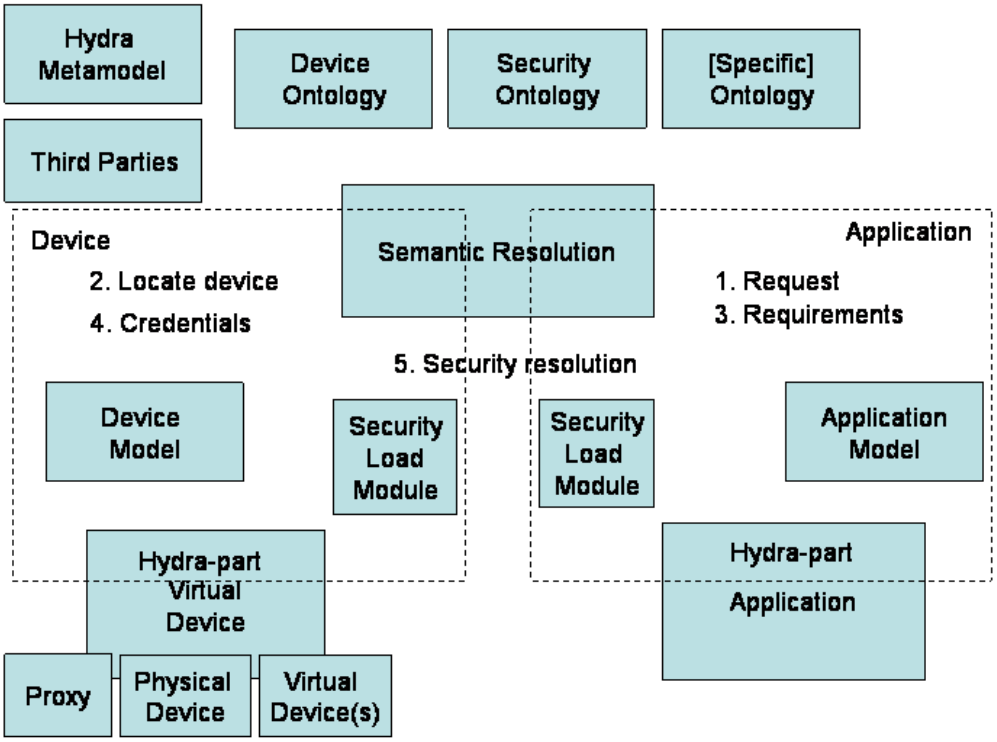


Figure 9: HYDRA Security Metamodel [Hydra, 2007c]

A basic design objective for the HYDRA security model is to provide a secure information flow with a minimum of pre-determined assumptions, while being able to dynamically resolve security requirements. This approach has been referred to as the Semantic Resolution of Security (c.f D7.2).

The security policies of HYDRA can thus be defined and enforced based upon knowledge in the Device Ontology as well as on knowledge of the context of devices, and also makes use of Virtual Devices.

4.3.2.1 Local -out

Virtual Devices and Device Proxies can be developed by any developer given access to the necessary device features.

The concept of *Local-out* means that a more local developed entity will take precedence over a more globally developed entity, i.e. that the closest to the run-time environment take precedence over developed modules further away from the Run-rime environment. For example, a Device manufacturer of a Non-Hydra enabled device makes a Proxy implementation and deploys this at his website to allow his device to be interoperable with Hydra. However, the local Domain Owner prefers an alternative Proxy either because he developed it himself or because some other entity made the Proxy available. When the Domain Owner downloads the version he prefers, it will take precedence over the Device Manufacturer version.

4.3.2.2 Virtual Device Network Connection

When a device comes into contact with a HYDRA Gateway, it will respond to a challenge by the HYDRA gateway by providing its device model type.

In the case a non-HYDRA device is discovered, HYDRA will from a model determination execute a search Local-out (first local implementation, then manufacturer and then global) to determine if a device model to describe the device can be located. If so the device model will be collected.

The device model can then contain linkage to a Proxy which is a specific implementation that HYDRA-enable this specific device type to HYDRA. If so the Proxy can be fetched, loaded and invoked in order to proceed with the device initialisation.

There must be specific security checks against both domain/network level security requirements and device-specific security requirements related to both fetching, loading and invoking the Proxy to protect against malicious code getting introduced this way. The most likely mechanisms will be related to PKI-signatures to validate the module and the source location. If a Proxy is loaded, there will likely be a Device model (in the Device Ontology) describing the Proxy-device (a non-hydra device model is optionally hidden by specification in the Proxy Device description).

We know have a HYDRA-enabled device, either the physical device it self or a Proxy-implementation that provide a HYDRA-interface to a non-Hydra device

In the case when a HYDRA device is discovered, HYDRA makes a search Local-out based on the Device type for Virtual Device modules. If located, HYDRA loads the Virtual Device Model which may include software modules adding new capabilities. The Virtual Device also introduces a Rules resolution layer that can implement access controls to the device such as a biometric sensor device adding end-user access controls.

In the communication with the newly discovered device, many communication protocols will also leak device identifiers - if so the Proxy or the Virtual Device strips this and relay to the semantic layer resolution while replacing it with a HYDRA Identifier (HID) for the network layer.

If the device is not able to manage its own HID interfaces, the Proxy or Virtual Device will act as an intermediate shielding the physical device from direct communication unless communication occur at the network layer outside HYDRA. .The HID is NOT representing the Device beyond network communication. All core security aspects such as accountability, authorisation, credentials negotiation is handled at the semantic layer

4.3.2.3 Semantic Resolver

The main element in the Semantic Resolution of Security is the Request. A request could be to locate either special devices or special functionality. The requesting device / application will also contain a dynamic set of Security requirements rules that govern the requirements that the requested Device must full fill in order for the process to succeed.

A Request is caught by the Semantic Resolver that orchestrates the process of locating the right devices and the subsequent resolution of security.

Based on the Device models incorporating the capabilities of the available Devices in the domain, the resolver can have different logical mechanisms to choose how resolution occurs. The process can be a multi-step process so that the Request can be functionally oriented and thereafter the security resolution takes place. A specific device may fail to comply with specific Security Requirements, so the process needs to be able to traverse through all possible candidates and somehow prioritise based on most likely candidates.

The Resolver does NOT act as a Trusted Party, i.e. it will determine which exchanges of security credentials that can and should take place and then support the end-to-end exchanges.

The Semantic Requirements is both transaction specific and can be dynamic depending on a number of changing aspects such as,

- an alert mechanism,
- a regulatory change,
- a change down the security evaluation of a certain capability,
- or for instance, simply a change in security policy

4.3.2.4 Capability upgrade

To ensure interoperability, it is vital that capabilities providing the same logical operation both are characterised as such in the Security Ontology.

When the Resolver detects a mismatch in security capabilities, it may initiate a process of Virtual Device Upgrade through loading of new capabilities, if available. Often this is a simple matter where one of the entities needs to upgrade a capability. The resolver will orchestrate the process. An upgrade will result in a new Virtual Device and an updated device description that covers the combined device.

Examples of such upgrades could be a newer symmetric encryption algorithm where the decrypt/re-encrypt module acts as a part of the former device where the transformation happens as close to the device as possible. Another example of a capability upgrade may be a sensor which was initially deployed without user authentication, but where errors have lead to an upgrade of an Application Security requirement that the End-user needs to authenticate as part of the device. This may be resolved by a load module that communicates for instance with an end-user PDA and subsequently the combination of a sensor device and PDA/End-User appearing as a virtual device with a richer security profile. A third example is an application in need to upgrade capabilities and security requirements. There the application may collaborate with online services

4.3.3 Use of models for context awareness

To support ambient intelligence applications we are also using models to provide context awareness functionality. This work is mainly pursued in workpackage 7 as well as in workpackages 3 and 4. In this section we give an example of how context can be handled by applying the ontology languages OWL and SWRL (see section 4.4 for details on these technologies). We refer to the forthcoming deliverable D3.8 and D7.3 for the overall approach to context management in HYDRA.

Below we can see how person and location are being modelled. The Location ontology imports different type of location description ontologies, which is based on [Flury, 2004]. The Person ontology is based on the SOUPA ontologies [Chen, 2005] and incorporated user-centered concepts for example user habits and user preferences.

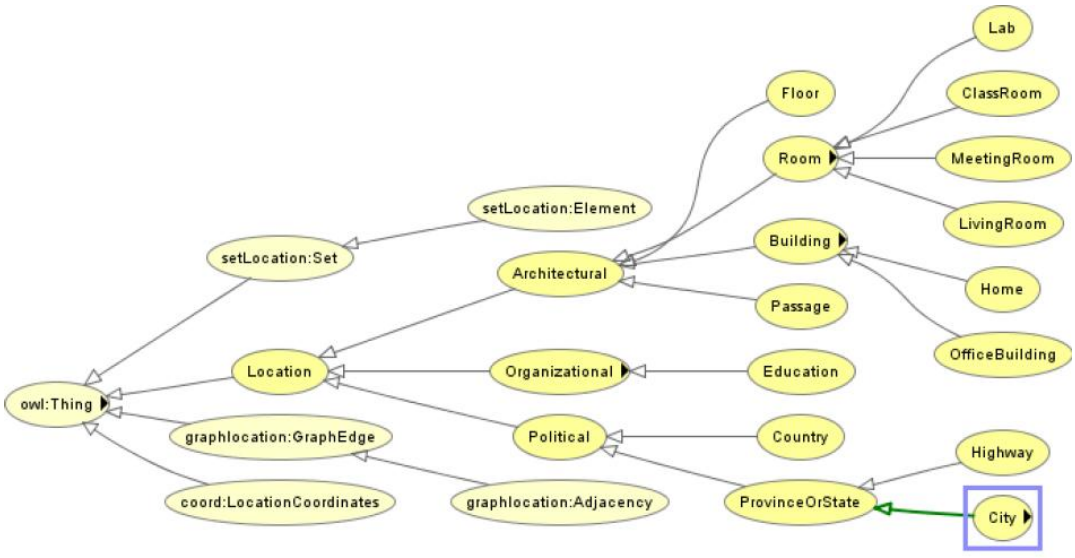


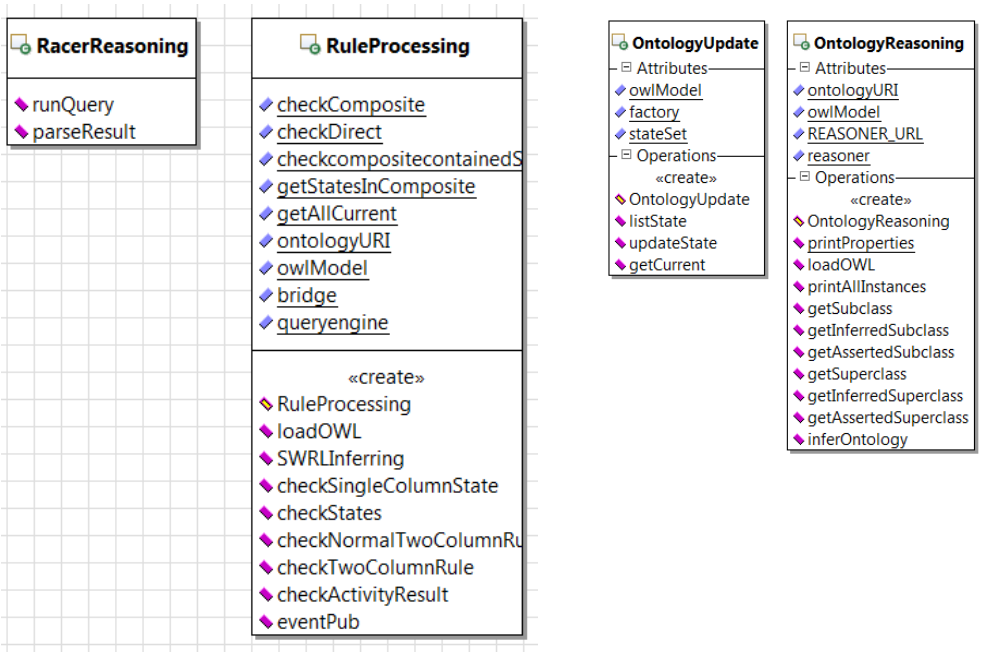
Figure 10: Example of a location concept modelled to support context awareness

Figure 11: Example of a person concept modelled to support context awareness

We will show an example with a Location based context awareness with SWRL to specify a complex context, which is not easily achievable with OWL itself. SWRL shares OWL's open world assumption, and is adding more expressive power to the underlying ontology. SWRL generalized OWL conditionals in two ways: arbitrary patterns of variables, free mixing of expressions (e.g., property and class expressions).

A working scenario can be like the following: A GPS publishes its data through the event manager, the Context manager gets raw GPS data, and processes it to get the coordinates, then the context manager will update StateMachine ontology to include new location data into the GPS state machine (update the activityResult datatype property associated with the objectProperty has Activity for the Simple (sub class of State concept) state). As the GPS is supposed to be with a person (say Klaus) to tell where he is, therefore the GPS data should also update the hasLocation property of the Person ontology at the same time. Then the context manager may process a rule stating that if the distance from home is more than 5 miles, then rule execution will create farAwayFromHome property instance for Klaus, and at the same time, the context of farAwayFromHome (Klaus, "True") will be published via the event manager, other parties can then respond to the new context, and take actions. For example home surveillance can use the highest security level policy.

The responsibilities of the managers can be changed when the design, performance, security issues are all clarified. And in simple cases of context awareness, there can be no SWRL rules involved. A prototypical implementation to prove the above concepts is implemented as shown in the paper [Zhang, 2007], with the case of Diagnostics manager using the SWRL/OWL. The concept of utilization of ontology and rules are the same in both Diagnostics manager and Context manager. The class diagram is shown in the following figure.



From the class diagram we can see that the rule processing can be generalized, but the rule result interpretation may be very different, from device to device, and from manager to manager. There are also common reasoning functions that are universal to all the managers that need ontology support, for example the class OntologyReasoning implemented with Protégé-OWL APIs.

SWRL is currently a W3C submission and attracts more and more attention from research for its usage for context awareness and situation-awareness. However, the development of tools and APIs are just starting or under planning. At present, we are using the SWRL tool from Protege [Oconnor, 2007].

4.4 Standards used

One of middleware is the ability to use of semantic web languages. In the development process, three standards were used:

- 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
- Semantic Web Rule Language (SWRL) allows definition of rules, which were used to extend the models of device state-machines.
- 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.

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

4.4.1 Modelling and query languages

4.4.1.1 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 superclasses, and can define properties, which may have subproperties, 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 standard 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-reasoner for organizing 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.

4.4.1.2 Semantic Web Rule Language (SWRL)

SWRL [SWRL, 2004] combines sublanguages of the OWL (OWL DL and Lite) with those of the Rule Markup 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.

The intended meaning of SWRL rules can be read as: whenever the conditions specified in the antecedent hold, then the conditions specified in the consequent must also hold. This is important fact which predetermines this framework to be used in context awareness, security (policy), in other words, if there is important to express facts about devices and their actual contexts, SWRL could provide very promising framework for modelling of naturally spread knowledge on several devices, application managers or context-aware elements in semantic middleware.

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).

4.4.1.3 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.

4.4.2 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).

4.4.2.1 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.

4.4.2.2 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.

4.4.2.3 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).

4.5 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.

4.5.1 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.

4.5.2 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, an 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 a 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 an 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, superclasses) to visualize their vicinity.

5. HYDRA ontologies

5.1 HYDRA ontology architecture

In Hydra there are three major ontologies used: The Device Ontology, a Security Ontology and a Software Components Ontology.

5.2 Device ontology

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.

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 the development phase, 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 subclassed by more specialized concepts and the new properties can be added. Specific device models are created as the instances of device ontology concepts are filled with real data.

The ontology diagrams presented in this chapter have been exported using the TopBraid editor. Note, that presented diagrams describe only high-level ontology structure. In some cases, the concept properties are hidden in order to reduce the complexity of figures.

The ontology architecture was designed to support the maintainability and future extensions of used concepts. The *HydraDevice* concept presents the main ontology class. *HydraDevice* class has one OWL DataType property *deviceId*, which is used in run-time as the unique URI assigned to the real device instance connected to HYDRA. Using this URI, it is possible to retrieve and update the relevant information related to the general description of a device and its actual run-time properties. The complete structure of semantic device descriptions, represented by the full device ontology, is divided into four interconnected modules:

- basic device information and taxonomy
- device malfunctions
- device capabilities and state machine
- device services

References to several ontology modules are realised as OWL ObjectProperties. The principal structure and usage of each module will be described in more details.

5.2.1 Basic device information

Basic device information represents only general and ordinary device information.

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 subclassing the *HydraDevice* 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 several device information into smaller units. Each device type should refer only to relevant part 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.

Further, the OWL object property *hasEmbeddedDevice* 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.

The semantic model of the basic device description is illustrated in Figure 12. The initial device taxonomy was taken from AMIGO project vocabularies for device descriptions [AMIGO, 2006].

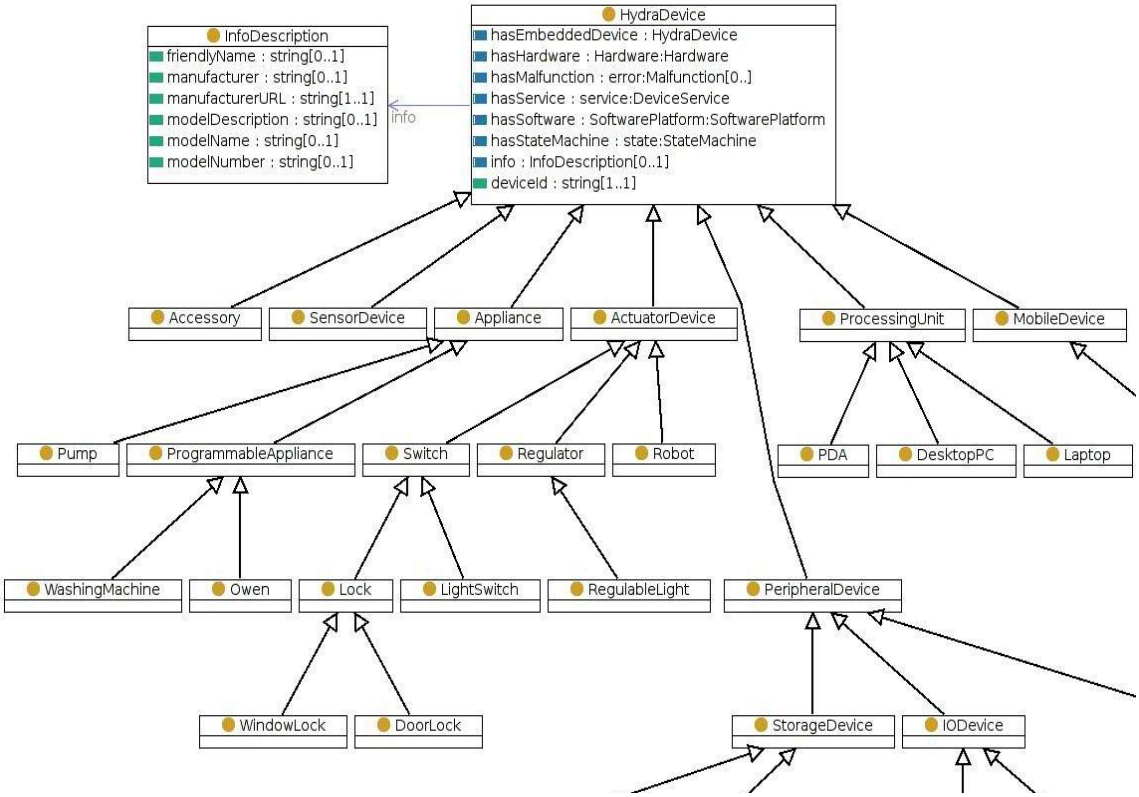


Figure 12: The basic Hydra Device taxonomy

5.2.2 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 *malfunctionCod*e, 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 subclassed 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.

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 device or to invoke the related callback events to handle the error states that may occur the run-time.

The model of basic device malfunctions is illustrated in figure Figure 13.

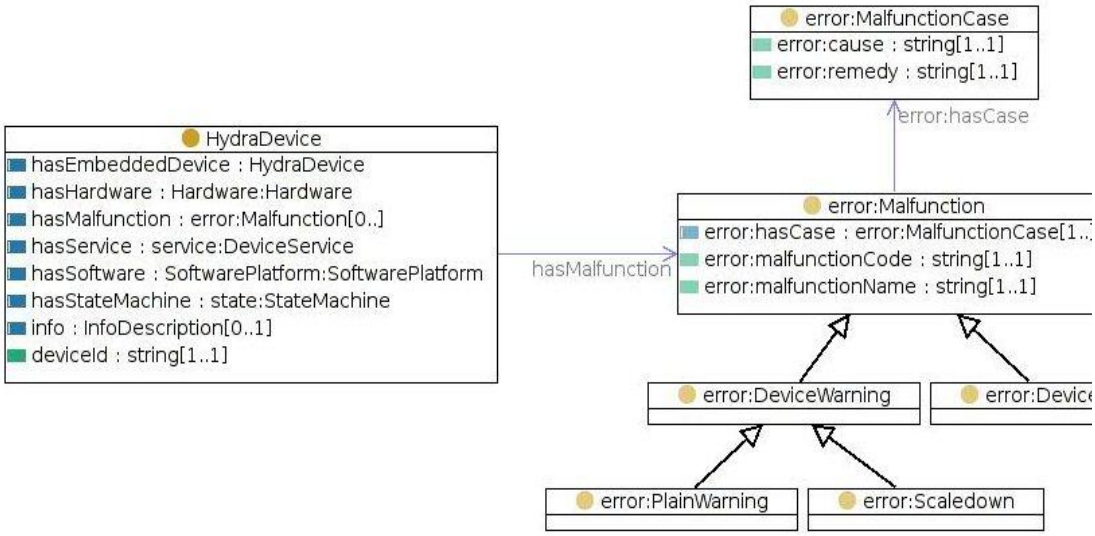


Figure 13: The malfunction part of the Hydra Device Ontology

5.2.3 Device capabilities and state machine

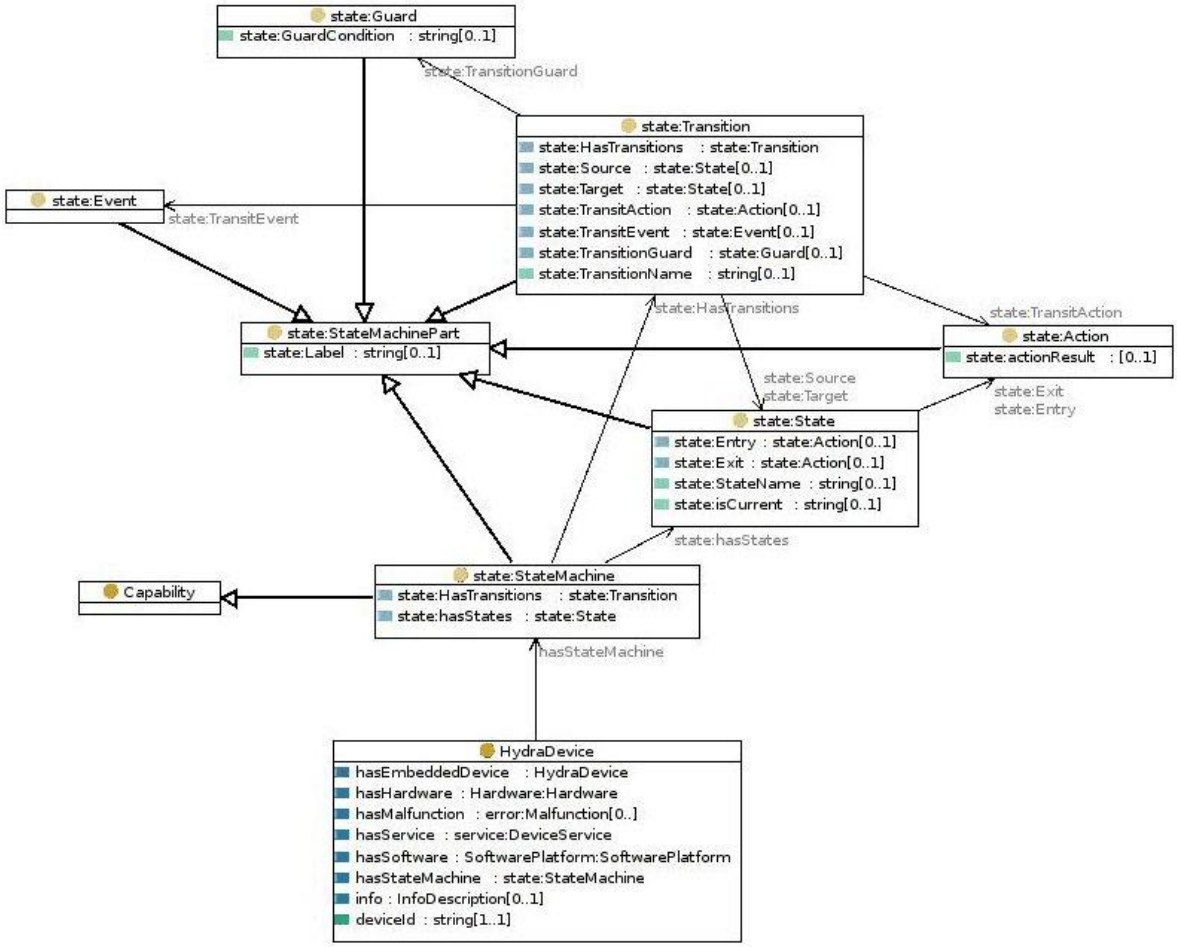


Figure 14: The State machine part of the Hydra Device Ontology

5.2.4 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 markup for services following web service architecture (the overview of related standards for semantic web service markup is presented in D6.3 deliverable). The OWL-S approach was taken as the starting point for HYDRA service model.

The *DeviceService* concept is referred from *HydraDevice* using OWL object property *hasService* and is composed of four main parts.

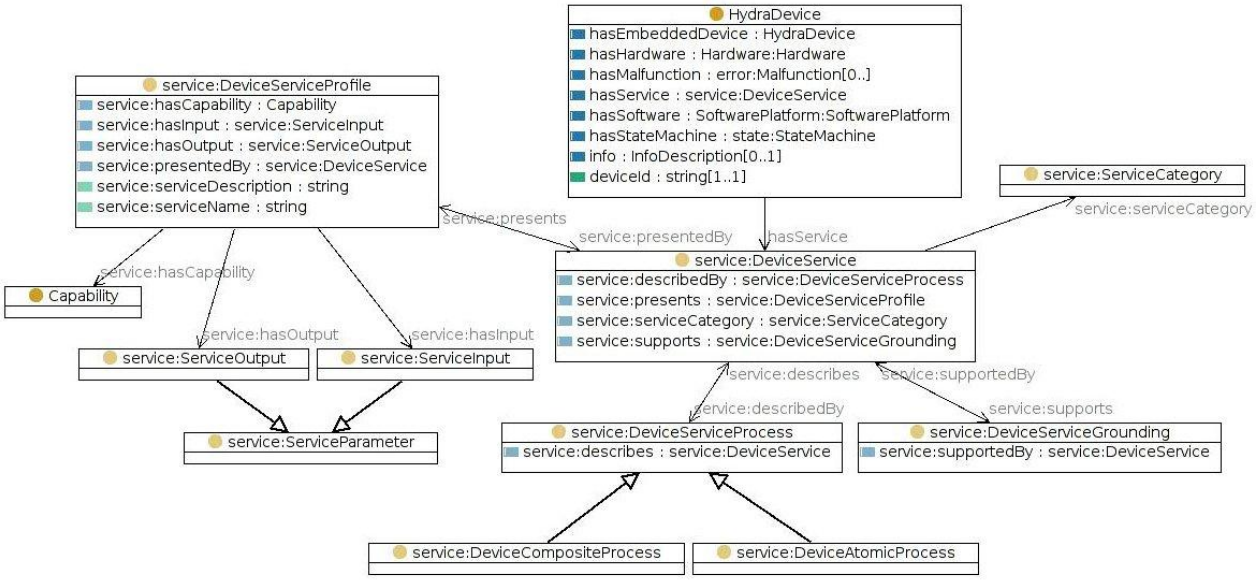


Figure 15: Modelling of services in the Hydra Device Ontology.

The concept *ServiceCategory* represents the taxonomy based on specific classification 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.

The *DeviceServiceProfile* concept presents the basic service description used mainly for service discovery process. The Service profile describes the general information, such as human readable service name and description, service capabilities and service inputs and outputs. The *Capability* concept is used to describe the specific service capabilities related to service functional properties, such as ability to handle various media formats or to handle required device states. *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. 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.

DeviceServiceProcess concept describes the service process model, which defines if the service represents the immediately invocable atomic process or work-flow of composite processes.

The *DeviceServiceGrounding* concept specifies the details, how to access the service and physically realise the service invocation. Service grounding represents the mapping from abstract to concrete specification of service elements used for interaction, namely the inputs and outputs of atomic processes. The atomic processes are mapped into WSDL files provided by the specific devices.

Proposed HYDRA device services model presents only the first draft of the service modelling approach and requires further investigation and research related to possible existing semantic service markup standards (such as WSMO) and the system architecture requirements.

5.2.5 Modelling Wireless and Resource Consumption Aspects

The three HYDRA domains are realizable deploying Wireless Sensor Networks (WSN), for this reason we have focused our attention on the technologies developed to deploy such networks. WSN is a wireless network consisting of spatially distributed autonomous devices using sensors to cooperatively monitor physical or environmental conditions, such as temperature, sound, vibration, pressure, motion or pollutants, at different locations. Although WSN was born for military

applications they are now used in many civilian application areas, including environment and habitat monitoring, healthcare applications, home automation, and traffic control.

WSNs are networks formed by small devices powered by batteries, in this way the main constraints on sensor nodes result in corresponding constraints on resources such as energy, memory, computational speed and bandwidth. The power consumption for the sensing, communication and data processing is lower than the energy required for data communication.

The data transmission is the central point to deploy WSNs, in particular we must consider the amount of data transmitted in WSNs. Using devices with a high transmission capacity will be a great advantage during video streaming or downloading documents, but inside a WSN the communications are formed by small packets, the dimensions of transmitted packets are approximately of 10 or 100 kbit.

All of these aspects need to be described and captured in the Hydra Device Ontology to allow different software managers to access the properties for different devices and make intelligent decisions regarding how to most effectively use the scarce resources available in the device.

Starting from this assumptions we focused our attention on wireless technology providing less power consumption and low data rate:

- Bluetooth
- ZigBee/XBee
- Wibree
- RFiD
- WiFi
- GSM

Obviously WiFi or GSM devices are not properly the less expensive wireless technologies from the power consumptions hand side, but in the latest years and in particular for the WiFi, were developed several core and power saving procedure that could reduce the power spent during data transmissions or during the idle state of the devices.

5.2.5.1 Wireless protocols

Bluetooth

Born as cable replacement technology, Bluetooth device was also used to implement the first kind of WSN, but cause to their power characteristic, the Bluetooth technology was considered inadequate [Zheng, 2004] Table 3.

The high power consumption and the high data rate don't respect the WSN assumptions; in this way Bluetooth technology could be used to realize body area networks (BAN) of one day life[1], but it is not used to implement WSN for huge scenarios, like agriculture, where nodes must gather the information in a range of several months. To realize such scenarios other technologies are developed and an example of them is the ZigBee.

[1] http://homepage.uab.edu/cdiamond/index.htm

Protocol	Op.Freq.	Data Rate	Range(I/O)	Maximum Power
Bluetooth class I	2.4 GHz	1 Mbit/s ver (1.2) 3 Mbit/s ver (2.0)	100 m	100 mW (20dBm)
Bluetooth class II	2.4 GHz	1 Mbit/s ver (1.2) 3 Mbit/s ver (2.0)	10 m	25 mW (4dBm)
Bluetooth class III	2.4 GHz	1 Mbit/s ver (1.2) 3 Mbit/s ver (2.0)	1 m	1 mW (0dBm)

Table 3 Bluetooth Characteristics

ZigBee

ZigBee standard was born to provide communication where the Bluetooth technology doesn't assure the best performance.

ZigBee is focused on control and automation, it uses low data rate, low power consumption, and works with small packet devices. ZigBee networks can support a larger number of devices and a longer range between devices, Table 4.

Protocol	Op.Freq.	Data Rate	Range(I/O)	Maximum Power
ZigBee	2.4 GHz	250 Kbit/s	10-75 m	1 mW (0 dBm)
ZigBee	915 MHz	40 Kbit/s	10-75 m	1 mW (0 dBm)
ZigBee	868 MHz	20 Kbit/s	10-75 m	1 mW (0 dBm)

Table 4 ZigBee Characteristics

Wibree

Wibree is a new interoperable radio technology for small devices. It can be built into products such as watches, wireless keyboards, gaming and sports sensors, which can connect to host devices such as mobile phones and personal computers.

Protocol	Op.Freq.	Data Rate	Range(I/O)	Maximum Power
WiBree	2.4 GHz	1 Mbit/s	5-10 m	- mW (- dBm)

Table 5 WiBree Characteristics

At the moment there aren't any available devices using Wibree standard.

RFiD

RFiD technology was realized to provide a support for operation like: automotive, product tracking, transportation payments, etc. On the market exists three kind of RFiD devices, everyone working on different frequency range 125/134 kHz, 13,56 MHz, 868/915 MHz, >2,4 GHz:

1. Passive RFID tags have no internal power supply. Passive tags have practical read distances ranging from about 10 cm up to a few meters, depending on the chosen radio frequency and antenna design
2. Active RFID tags have their own internal power source, which is used to power the integrated circuits and broadcast the signal to the reader. Many active tags today have practical ranges of hundreds of meters, and a battery life of up to 10 years in turn, they are generally bigger and more expensive than the passive RFiD
3. Semi-active RFiD tags, are similar to active tags in that they have their own power source, but the battery only powers the microchip and does not broadcast a signal. The RF energy is reflected back to the reader like a passive tag

In this way if we want to deploy a WSN using RFiD devices we must use active and semi-active tags, equipped with a microcontroller and with a remarkable memory.

WiFi

WiFi products are today the most available solutions for high data rate wireless communications, in the years several different standards was developed to satisfy different needs, like interferences and data rate. Table 6 shows some parameters identifying the several WiFi protocol realized until now, the only aspect that are not indicated is the most important for our scope, the power consumptions.

Protocol	Op.Freq.	Data Rate	Range(I/O)
802.11a	5 GHz	54 Mbit/s	35/120 m
802.11b	2.4 GHz	11 Mbit/s	38/140 m
802.11g	2.4 GHz	54 Mbit/s	38/140 m
802.11n	2.4 GHz 5 GHz	248 Mbit/s	70/250 m
802.11y	3.7 GHz	54 Mbit/s	50/5000 m

Table 6 WiFI Characteristics

The power consumption of the WiFi technology is a constant aspect for these products, up to 20 dBm equal to 100 mW during communication. This value guarantees great performances in terms of connection availability giving the possibility to reach internet also in no line of sight connection. But if this aspect is one of the WiFi communication strong points, this represent a drawback for a lot of handled devices that need to be awake for a great amount of time.

GSM

Global System for Mobile communication (GSM) is a globally accepted standard for digital cellular communication. GSM uses the circa 900 Mhz band characterized by the 890-915MHz frequencies in uplink and the 935-960MHz in downlink as indicated in the Table 7.

Protocol	Op.Freq.	Data Rate	Maximum Power
GSM	890-915MHz UL 935-960MHz DL 1710-1785 MHz UL 1805-1880 MHz DL	270 kb/s	1 W (30dBm)

Table 7 GSM Characteristics

5.2.5.2 Power Consumption

All devices involved in communication spend a lot of their energy in data transmission. As could be seen in Table 8, these values span from 0 dBm, for ZigBee, to 30 dBm, for GSM devices.

All the devices implementing these wireless technologies are handled devices and are powered by batteries, exception done for those ones that could act like gateways or bridges, which could be powered in many way:

1. Batteries 6V DC (Q52, Q53)
2. Power over Ethernet (Q52, Q53, Libellium Multigateway)
3. Power supplied by 220V (Libellium Multigateway)

In this way the problem arising with power consumption could be translated in lifetime problems.

Protocol	Op.Freq.	Power Consumption
802.11 x	2.4 GHZ or 3.7 GHz or 5 GHz	20 dBm
Bluetooth	2.4 GHz	from 0 dBm up to 20 dBm
ZigBee	2.4 GHz	0 dBm
WiBree	2.4 GHz	- dBm
GSM	890-915 MHz UL 935-960 MHz DL 1710-1785 MHz UL 1805-1880 MHz DL	30 dBm

Table 8 Power Consumption

To resolve such kind of problems several device manufacturers implement, inside their devices, energy saving procedures. All these procedures, also called connection states, represent a reduction in term of power consumption reducing the awake time of the devices forcing them to listen for connections during scheduled times, or reducing the duty cycle of the involved device.

5.2.5.3 Wireless properties in the Device Ontology

All the showed technologies represent a solution in wireless communications for several scenarios, in which we could:

- use ZigBee/XBee technology to develop WSNs for agriculture scenario (several market solutions are proposed)
- use Bluetooth technology to develop WSNs for healthcare scenario (the development of the new Medical Device Profile gives the opportunities to reduce cost size and power consumptions)
- use ZigBee/XBee technology to develop WSNs for smart-home scenario (in particular a new device coming from Zensys enterprise, Z Wave, is one of the suitable product able to realize these applications as explained in D5.4)

As could be evident in the several tables showed in this paragraph, there are a lot of devices that works in the ISM band at 2.4 GHZ. As explained in deliverable D5.4 this couldn't represent a problem in terms of Packet Loss in an heterogeneous network formed by a small number of devices, but other analysis could be performed to evaluate other parameters that could jeopardize the network efficiency.

To decrease the collision characteristic, due to the same medium sharing, could be suitable to develop inside the device ontology same aspects that could be relevant, in particular, for the transmission:

1. Transmission frequencies already occupied by some devices
2. Number of devices working at a particular frequency
3. Medium access criteria

Knowing a priori these and other information could improve the network performances in terms of reduced interference.

A such kind of medium access criteria is already implemented in devices like Bluetooth core 5 that have replaced the FH (Frequency Hopping) with the AFH (Adaptive Frequency Hopping).

To allow all devices which are part of Hydra to implement such mechanism could be a good solution to add in the device ontology some issues where these aspects are signalled, in this way a new device that wants to communicate inside the Hydra heterogeneous network could use frequencies where the collision probability is reduced.

6. Middleware managers

This chapter describes the middleware manager elements (in red) that constitute the main parts of the semantic MDA, explaining their roles, functions and component structure. Furthermore there is a section covering the common XML-Schema that is used for representing common objects which is related to the different ontologies.

The Functional Structure model is divided into two parts: Application Elements and Device Elements. Both elements differ in the following aspects:

- Resources available on the of the machine on which they are supposed to run (e.g., embedded platform vs. server platform)
- Intended purpose of the components (e.g., to develop domain-specific applications or to develop application-independent device

The following diagrams explain how the application and device elements are logically grouped.

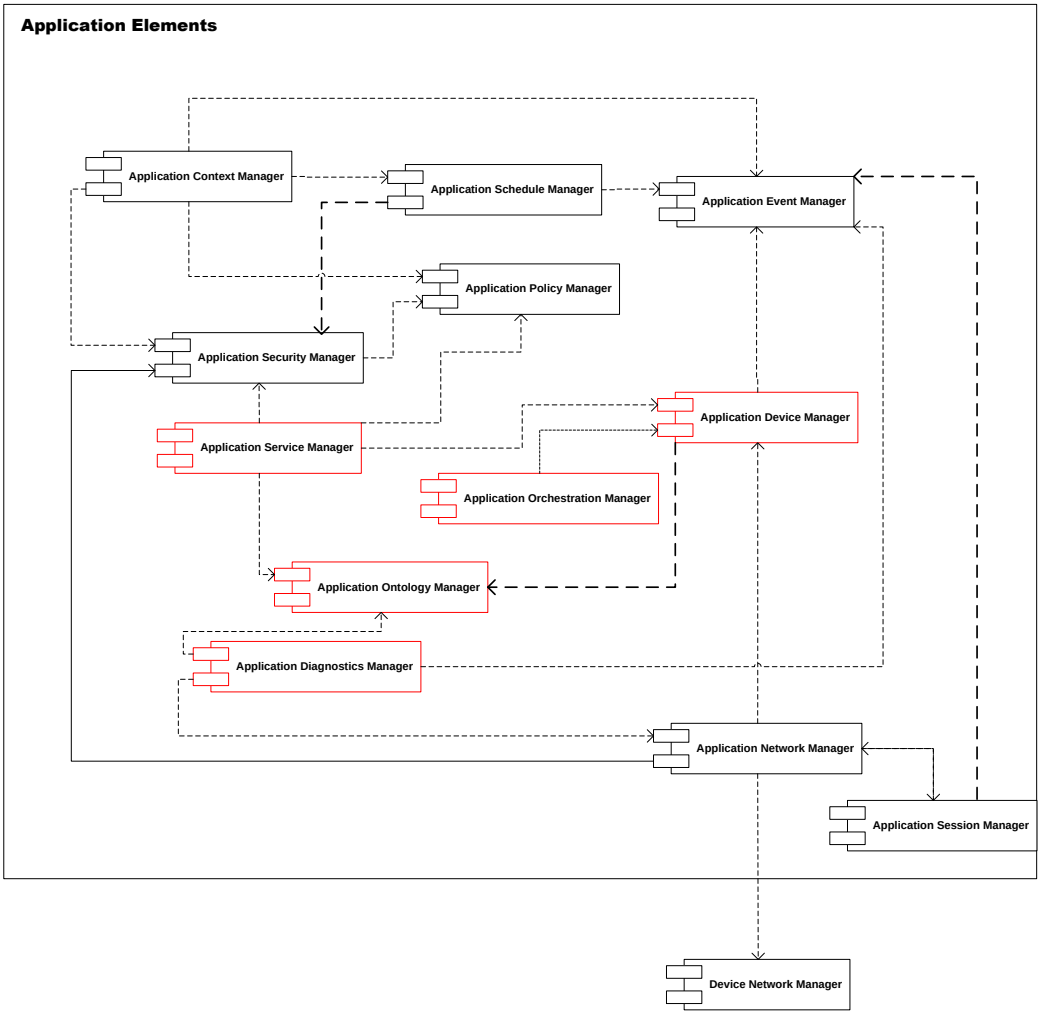


Figure 16: Overview of Application Elements

6.1 Application Device Manager

6.1.1 Purpose

The Application Device Manager manages all knowledge, metadata and information regarding devices that have been discovered and are active in the Hydra network. The Application Device Manager knows about devices from a network perspective but does not handle the locations or context of the devices.

Main Functions:

- Discover new (existing) semantic devices
- Assigns a device type to the device based on Device Ontology.
- Returns service interface for the device
- Handles device virtualization (semantic devices)
- Handles semantic device aggregation
- Manages a Device Application Catalogue

6.1.2 Related WP6 requirements

[HYDRA-91] Any HYDRA device should have an associated description	
Status:	Part of specification
Requirement Type:	Functional
Workpackage:	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
Workpackage:	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
Workpackage:	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-110] Device Categorisation in runtime **Created: 28/Nov/06 Updated: 09/Oct/07**	
Status:	Part of specification
Requirement Type:	Functional
Workpackage:	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
Workpackage:	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
Workpackage:	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 Dissatisfaction:	high
[HYDRA-120] Multiple Device Virtualisations	
Status:	Part of specification
Requirement Type:	Functional

Workpackage:	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-218] Support interaction devices	
Status:	Part of specification
Requirement Type:	Functional
Workpackage:	WP6
Rationale:	Interaction devices provide users with different forms of output (display) capabilities. This could include simple displays, tablets or more advanced units.
Source:	WP6 MDA scenario
Fit Criteria:	Interaction devices (displays) are included in the HYDRA device ontology and can be mapped to the end-user interface of an application.
Developer Satisfaction:	high
Developer Dissatisfaction:	neutral

[HYDRA-325] Support aggregation and separation of devices and services	
Status:	Part of specification
Project:	HYDRA
Requirement Type:	Functional
Workpackage:	WP6
Rationale:	Devices and services may exist in a separate application where they should not be influenced by nearby (wireless) devices such as in the case of an apartment. Thus it should be possible to view a set of services/devices as an aggregate that is separated and isolated from other sets of services/devices
Source:	UAAR focus group
Fit Criteria:	Check support for aggregation and separation of devices/services
Developer Satisfaction:	neutral
Developer Dissatisfaction:	neutral

[HYDRA-376] Security requirements must be part of the Hydra MDA	
Status:	Part of specification
Requirement Type:	Functional
Workpackage:	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	high

Dissatisfaction:	

6.1.3 Components

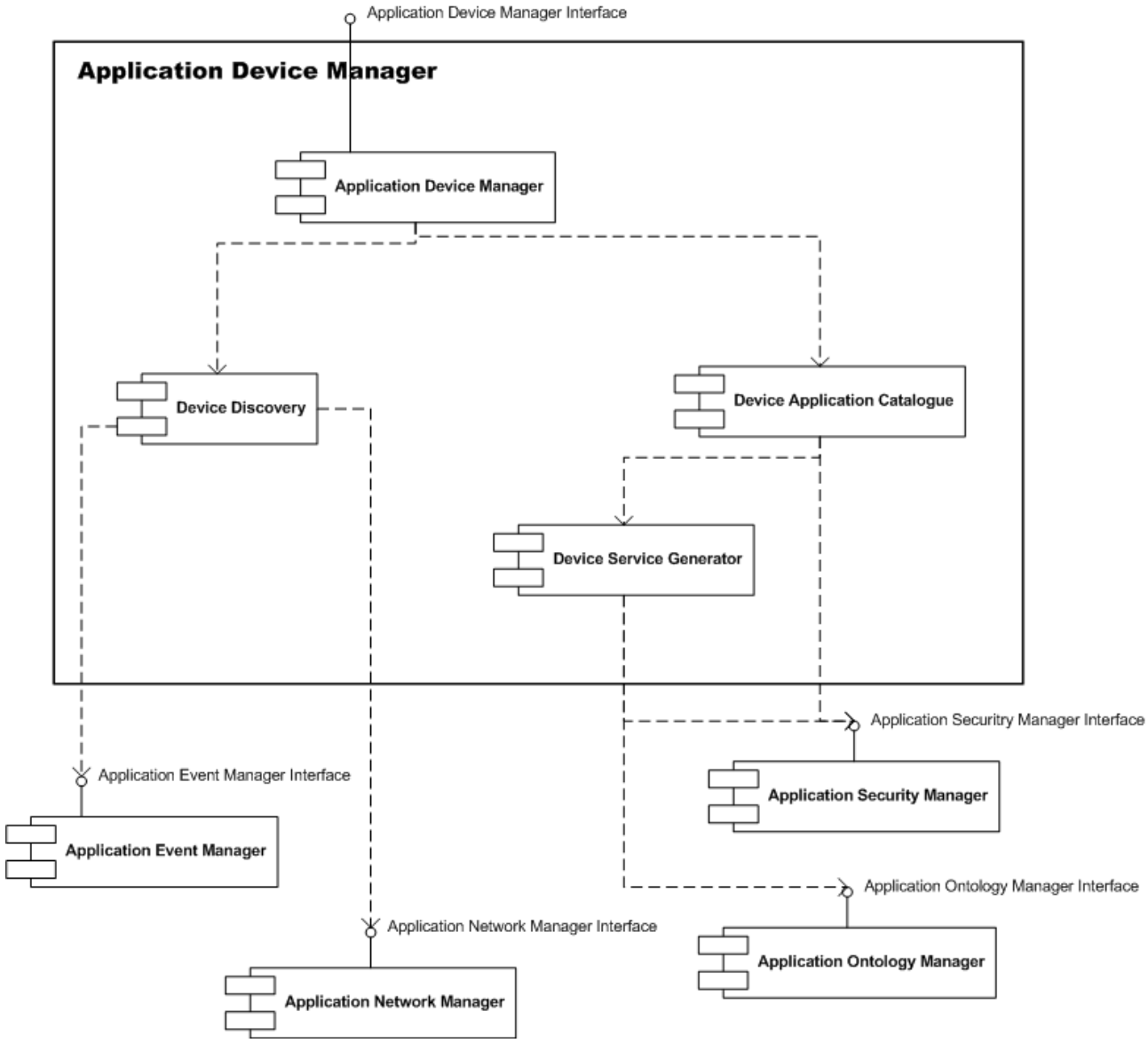


Figure 17: Application Device Manager

There are three main subcomponents of the Device Manager.

6.1.3.1 Device Discovery

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.

6.1.3.2 Device Application Catalogue

The Device Application Catalogue keeps track of and manages all devices that are currently active within one application. It 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 "error state". The "error state" will reflect if the Device Application Catalogue believes that the device is working. This state should be maintained by the Application Diagnostic Manager. The Device Application Catalogue should also provide methods for removing devices, i.e. that devices that are removed can unregister themselves from the catalogue. Requirements 91, 98, 110 and 111 are associated with this module.

6.1.3.3 Device Service Generator

The Device Service Generator is responsible for generating a service interface for a certain device. It will create a software wrapper around the device which other modules can use to communicate with and control the device. Requirements 91, 111, 120 and 325 are associated with this module.

6.1.4 Dependencies

Application Ontology Manager, Application Event Manager, Application Network Manager and Application Security Manager

6.1.5 Interface

string ApplicationDeviceManager::ProcessErrorMessage(string deviceid, XmlNode *theMessage*)

Processes an error message.

Parameters:

theMessage The error message as an XML Node.

Returns:

A description of the error.

string ApplicationDeviceManager::ProcessErrorMessageString(string deviceid, string *theMessage*)

Parameters:

theMessage The error message as a string.

Returns:

A description of the error.

string ApplicationDeviceManager::SetDeviceStatus(string *deviceid*, string *statusmessage*)

Sets the device status.

Parameters:

deviceid The ID of the device.

statusmessage A message describing the changes to be made to the device status.

Returns:

The updated device status.

string ApplicationDeviceManager::GetDeviceInfo (string *deviceid*)

Returns status and other info of the device.

Parameters:

deviceid The ID of the device.

Returns:

The current device status and info.

XmlNode ApplicationDeviceManager::GetDeviceStatus (string *deviceid*)

Returns the device status as an XML Node.

Parameters:

deviceid The ID of the device.

Returns:

The current device status.

XmlNode ApplicationDeviceManager::GetDeviceXML (string *deviceid*)

Returns the XML description of the device.

Parameters:

deviceid The ID of the device.

Returns:

An XML Node containing the description of the device.

string ApplicationDeviceManager::GetDevices (string *type*)

Returns a list of the devices currently available in the network.

Parameters:

type A device type.

Returns:

Alist of the currently available devices.

XmlDocument ApplicationDeviceManager::GetDevicesAsXML (string *type*)

Returns a list of the devices currently available in the network.

Parameters:

type A device type.

Returns:

An Xml Document containing a list of the currently available devices.

XmlDocument ApplicationDeviceManager::GetDeviceOntologyDescriptionAsXML (string *deviceontology_id*)

Returns the ontology description of the device as an OWL Document.

Parameters:

deviceontology_id The id of the device.

Returns:

The ontology description of the device as an OWL Document.

string ApplicationDeviceManager::GetDeviceOntologyDescription (string *deviceontology_id*)

Returns the ontology description of the device as a string.

Parameters:

deviceontology_id The id of the device.

Returns:

The ontology description of the device as a stringt.

string ApplicationDeviceManager::GetProperty (string *deviceid*, string *property*)

Returns the named property of the device.

Parameters:

deviceid The id of the device.

property The name of the property.

Returns:

The value of the property.

bool ApplicationDeviceManager::HasProperty (string *deviceid*, string *property*)

Indicates if the device has a property with the specified name.

Parameters:

deviceid The id of the device.
property The name of the property.

Returns:

True if the property exists, false otherwise.

string ApplicationDeviceManager::SetProperty (string *deviceid*, string *property*, string *value*)

Sets the named property of the device.

Parameters:

deviceid The id of the device.
property The name of the property.
value The new value of the property.

Returns:

The new value of the property.

string ApplicationDeviceManager::Invoke(XmlNode invokeMessage)

Generic method to invoke any method in a service on a device.

Parameters:

invokeMessage The invoking message containing deviceid,serviced, methodname, parameters, values

Returns:

The result of invoking the method.

string ApplicationDeviceManager::AddDevice(XmlNode devicedescription)

Allows manual adding of devices to the network that cannot be discovered using the default discovery protocol

Parameters:

devicedescription The device to be added to the catalogue.

Returns:

The result of adding the device.

string ApplicationDeviceManager::DeleteDevice(string deviceid)

Deletes the device from the Device Application Catalogue

Parameters:

deviceid The deviceid of the device to be deleted from the network.

Returns:

The result of deleting the device.

bool ApplicationDeviceManager::IsRegistered(string HID)

Checks if a certain device with a Hydra ID is part of this catalogue.

Parameters:

HID The HID of the device.

Returns:

True if the device is registered, *false* otherwise.

6.2 Application Service Manager

6.2.1 Purpose

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

Main Functions:

- Service discovery
- Semantic service creation (service orchestration/clustering and mapping to device service)

6.2.2 Related WP6 requirements

[HYDRA-104] Automatic Discovery of Services	
Status:	Part of specification
Requirement Type:	Functional
Workpackage:	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
Workpackage:	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
Workpackage:	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
Workpackage:	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
Workpackage:	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
Workpackage:	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-325] Support aggregation and separation of devices and services	
Status:	Part of specification
Project:	HYDRA
Requirement Type:	Functional
Workpackage:	WP6
Rationale:	Devices and services may exist in a separate application where they should not be influenced by nearby (wireless) devices such as in the case of an apartment. Thus it should be possible to

	view a set of services/devices as an aggregate that is separated and isolated from other sets of services/devices
Source:	UAAR focus group
Fit Criteria:	Check support for aggregation and separation of devices/services
Developer Satisfaction:	neutral
Developer Dissatisfaction:	neutral

[HYDRA-372] Interfacing with external systems	
Status:	Part of specification
Requirement Type:	Functional
Workpackage:	WP6
Rationale:	Searching and using external services in decision support and application intelligence must be supported
Source:	WP 6 Focus Group, WP2 Input
Fit Criteria:	Access to external systems using web service protocols (WS-I Basic Profile) is supported
Developer Satisfaction:	neutral
Developer Dissatisfaction:	neutral

[HYDRA-376] Security requirements must be part of the Hydra MDA	
Status:	Part of specification
Requirement Type:	Functional
Workpackage:	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

6.2.3 Components

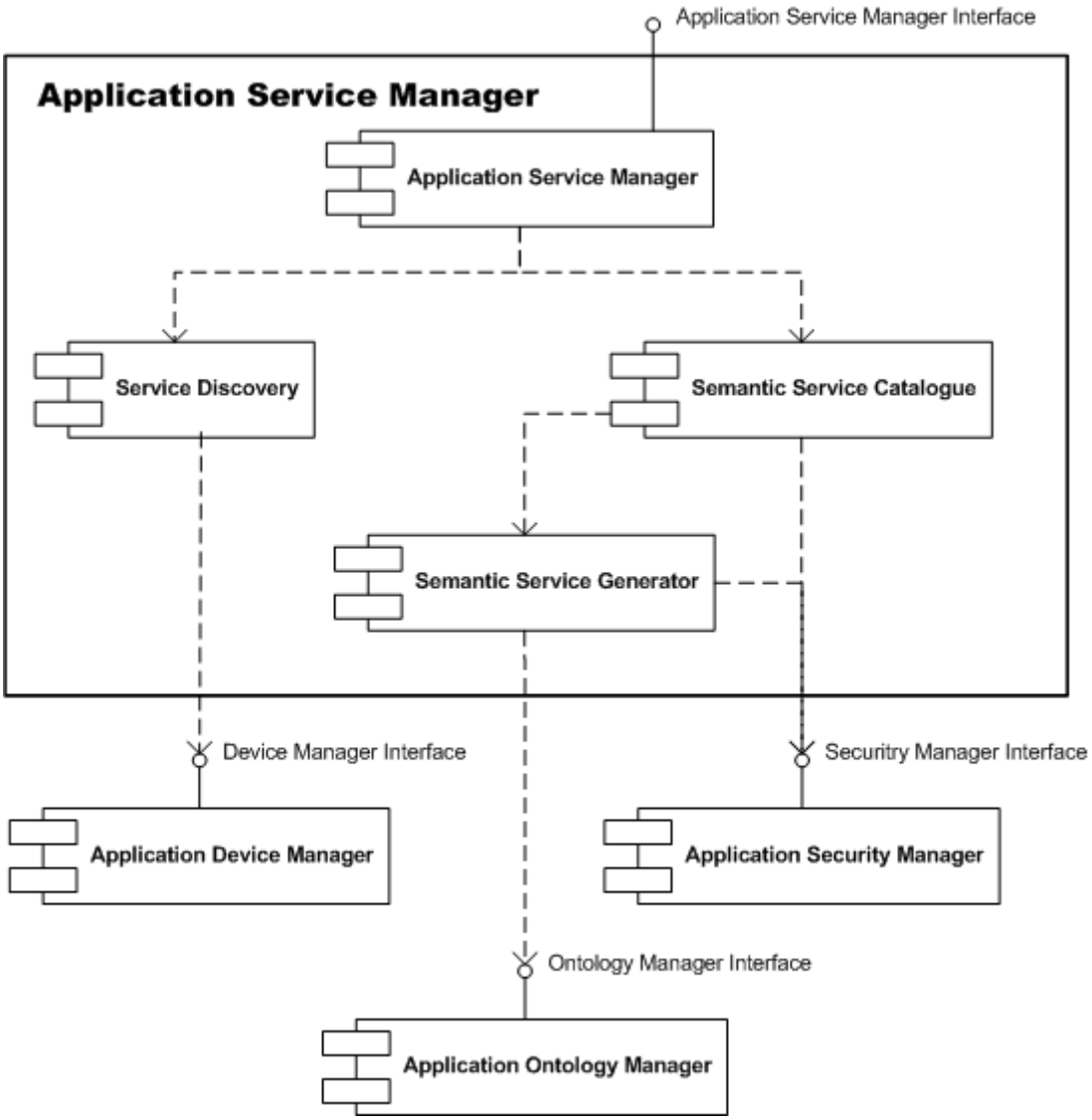


Figure 18: Application Service Manager

6.2.3.1 Service Discovery Module

One of the major functions of the Service Manager is to discover new services in the network. This is taken care of by the Service Discovery Module. It will use the Device Manager to find out about services offered by different devices.

6.2.3.2 Semantic Service Catalogue:

The Semantic Service Catalogue keeps track of and manages all service offered within one application. It can be queried about existing services. It can also provide semantic service interfaces for the different services upon request.

6.2.3.3 Semantic Service Generator

The Semantic Service Generator is responsible for generating a semantic service interface for services offered by devices. It will create a software wrapper around the device services which other modules can use. The generated software will support a semantic-based service interface. It will support several semantic web standards, at least OWL-S and WSMO.

6.2.4 Dependencies

Application Service Manager, Application Ontology Manager and Application Security Manager

6.2.5 Interface

string ApplicationServiceManager::ProcessErrorMessage (XmlNode *theMessage*)

Processes an error message.

Parameters:

theMessage The error message as an XML Node.

Returns:

A description of the error.

string ApplicationServiceManager::ProcessErrorMessageString (string *theMessage*)

Processes an error message.

Parameters:

theMessage The error message as a string.

Returns:

A description of the error.

bool ApplicationServiceManager::HasService (string deviceid, string *serviceid*)

Checks if a service is available.

Parameters:

service serviceid The service name.
deviceid The device to be queried

Returns:

True if service is available otherwise false.

string ApplicationServiceManager::GetServiceDescription (string *devicetype*, string *serviceid*)

Retrieves a device description.

Parameters:

devicetype The device type as a string.
serviceid The service id as a string.

Returns:

A string containing a service description in XML format.

XmlNode ApplicationServiceManager::GetServiceDescriptionAsXML (string *devicetype*, string *serviceid*)

Retrieves a device description.

Parameters:

devicetype The device type as a string.
serviceid The service id as a string.

Returns:

An XmlNode containing a service description.

string ApplicationServiceManager::GetServices(string *type*)

Retrieves a list of available services.

Parameters:

type The device service type as a string.

Returns:

A string containing the list of available devices services in XML format.

XmlNode ApplicationServiceManager::GetServicesAsXML (string *type*)

Retrieves a list of available services.

Parameters:

type The device service type as a string.

Returns:

An XmlNode containing the list of available service.

string ApplicationServiceManager::Invoke(XmlNode invokeMessage)

Generic method to invoke any method in a service on a device.

Parameters:

invokeMessage The invoking message containing serviced, methodname, parameters, values

Returns:

The result of invoking the method.

6.3 Application Orchestration Manager

6.3.1 Purpose

The Application Orchestration Manager provides support for composite services and workflows. It is an execution engine for the Hydra Device Orchestration Language ("DOLL").

Main Functions:

- Execute call sequences consisting of invocations of Device services
- Provide scheduling of notifications and service calls for Hydra applications

6.3.2 Related WP6 requirements

[HYDRA-113] Composition (of services and devices)	
Status:	Part of specification
Requirement Type:	Functional
Workpackage:	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
Workpackage:	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 sevices can be interchanged and combined.
Developer Satisfaction:	neutral
Developer Dissatisfaction:	neutral

[HYDRA-376] Security requirements must be part of the Hydra MDA	
Status:	Part of specification
Requirement Type:	Functional
Workpackage:	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

6.3.3 Components

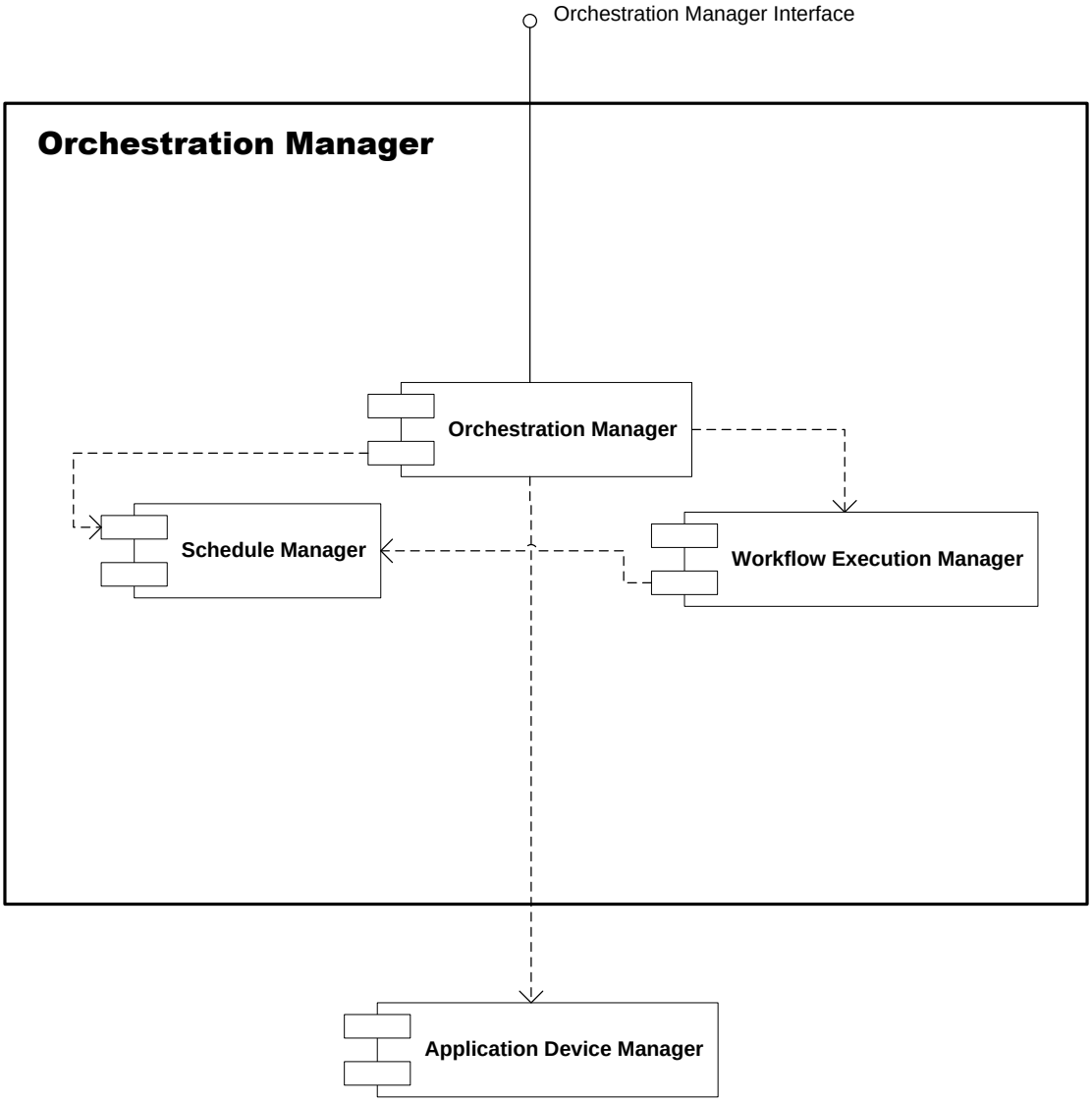


Figure 19: Application Orchestration Manager

Schedule Manager: The scheduler is responsible for running tasks or notifying applications when a specific criteria is met. Such a criteria can be a e.g. specific (possibly recurring) time, system startup, system shutdown.

Workflow Execution Manager: The workflow execution module interprets process descriptions and executes a set of services. These processes may represent a complex service composed of other services or part of a HYDRA application.

Dependencies: Application Device Manager

6.3.4 Interface

XmlNode OrchestrationManager::LoadProcessDescription (XmlNode processDescription) Loads a process description into the Orchestration Manager.

Parameters:

processDescription The ontology deviceId.

Returns:
A XML node containing the result of the operation and invocation data or the description of any errors that occurred during method invocation.

XmlNode OrchestrationManager::ListProcessDescriptions () Lists process descriptions previously loaded into the Orchestration Manager.

Parameters:

Returns:
A XML node containing all process descriptions loaded into the Orchestration Manager.

XmlNode OrchestrationManager::InvokeProcessDescription (XmlNode invocationData) Invokes a process description previously loaded into the Orchestration Manager.

Parameters:
invocationData An XML node with data identifying the process and invocation data for invocation.

Returns:
A XML node containing the result of and data returned from the invocation or the description of any errors that occurred during invocation.

6.4 Application Ontology Manager

6.4.1 Purpose

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
- Search/Query function
- Update
- Ontology versioning
- Reasoner module

6.4.2 Related WP6 requirements

[HYDRA-91] Any HYDRA device should have an associated description	
Status:	Part of specification
Requirement Type:	Functional
Workpackage:	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
Workpackage:	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

[HYDRA-101] Manual device ontology definition	
Status:	Part of specification
Requirement Type:	Functional
Workpackage:	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 a device ontology.
Developer Satisfaction:	low
Developer Dissatisfaction:	high

[HYDRA-103] Automatic device ontology construction	
Status:	Part of specification
Requirement Type:	Functional
Workpackage:	WP6
Rationale:	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.
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
Workpackage:	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
Workpackage:	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
Workpackage:	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
Workpackage:	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-123] Support updates at run-time	
Status:	Part of specification
Requirement Type:	Non-Functional - usability
Workpackage:	WP6
Rationale:	The middleware should be dynamically updatable at run-time due to critical systems updates (security updates, component upgrades, etc.).
Source:	WP6 MDA focus group
Fit Criteria:	Deployed middleware should execute 70% of the dynamic updates without failure and restart

Developer Satisfaction:	high
Developer Dissatisfaction:	very low

[HYDRA-125] Transactional updates	
Status:	Part of specification
Requirement Type:	Non-Functional - usability
Workpackage:	WP6
Rationale:	It should be possible to rollback and recover from an unsuccessful update.
Source:	WP6 MDA Focus Group
Fit Criteria:	Rollback works in 7 out of 10 scenarios.
Developer Satisfaction:	high
Developer Dissatisfaction:	high

[HYDRA-126] Automatic Device ontology updates	
Status:	Part of specification
Requirement Type:	Functional
Workpackage:	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

[HYDRA-139] Knowledge model of hydra middleware	
Status:	Part of specification
Requirement Type:	Functional
Workpackage:	WP6
Rationale:	Knowledge model of the whole middleware providing developers with knowledge on all middleware components offers a guidance how ho compose a hydra-based application.
Source:	State of the Art
Fit Criteria:	Support for knowledge model based rapid development is available
Developer Satisfaction:	very high
Developer Dissatisfaction:	neutral

[HYDRA-141] Download and harmonisation of third party device ontologies	
Status:	Part of specification

Requirement Type:	Functional
Workpackage:	WP6
Rationale:	Device ontological models describing devices, which will be provided by manufacturers or third parties, should be automatically downloaded (updated) and harmonised to ensure the same ontological view. Formal definition of ontologies should be realised using the world wide

	accepted formats, recommended by W3C, such as RDF, OWL, and OWL-S.
Source:	Hydra D2.2 Initial Technology Watch Report
Fit Criteria:	Ontologies from different manufacturers can be used if they are in RDF, OWL or OWL-S
Developer Satisfaction:	very high
Developer Dissatisfaction:	very high

[HYDRA-359] Handling of different device versions in device ontology	
Status:	Part of specification
Requirement Type:	Functional
Workpackage:	WP6
Rationale:	The device ontology should be able to handle different versions of a device.
Source:	WP6 MDA Focus Group
Fit Criteria:	The device ontology can maintain at minimum 2 versions of any single device
Developer Satisfaction:	high
Developer Dissatisfaction:	neutral

[HYDRA-365] Ability to self-adaptation	
Status:	Part of specification
Requirement Type:	Functional
Workpackage:	WP6
Rationale:	A knowledge model enables the middleware to contain a representation of itself and manipulate its state during its execution. This feature should serve as the basis for self-adaptation of the middleware (e.g. reconfiguration of resource usage, triggering the component-based services).
Source:	Hydra D2.2 Initial Technology Watch Report
Fit Criteria:	Middleware is able to adapt its configuration in 60% of identified cases requiring reconfiguration.
Developer Satisfaction:	very high
Developer Dissatisfaction:	neutral

[HYDRA-376] Security requirements must be part of the Hydra MDA	
Status:	Part of specification
Requirement Type:	Functional
Workpackage:	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

[HYDRA-392] Rules for selection of alternative devices	
Status:	Part of specification
Requirement Type:	Functional

Workpackage:	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 sevices can be interchanged and combined.
Developer Satisfaction:	neutral
Developer Dissatisfaction:	neutral

6.4.3 Components

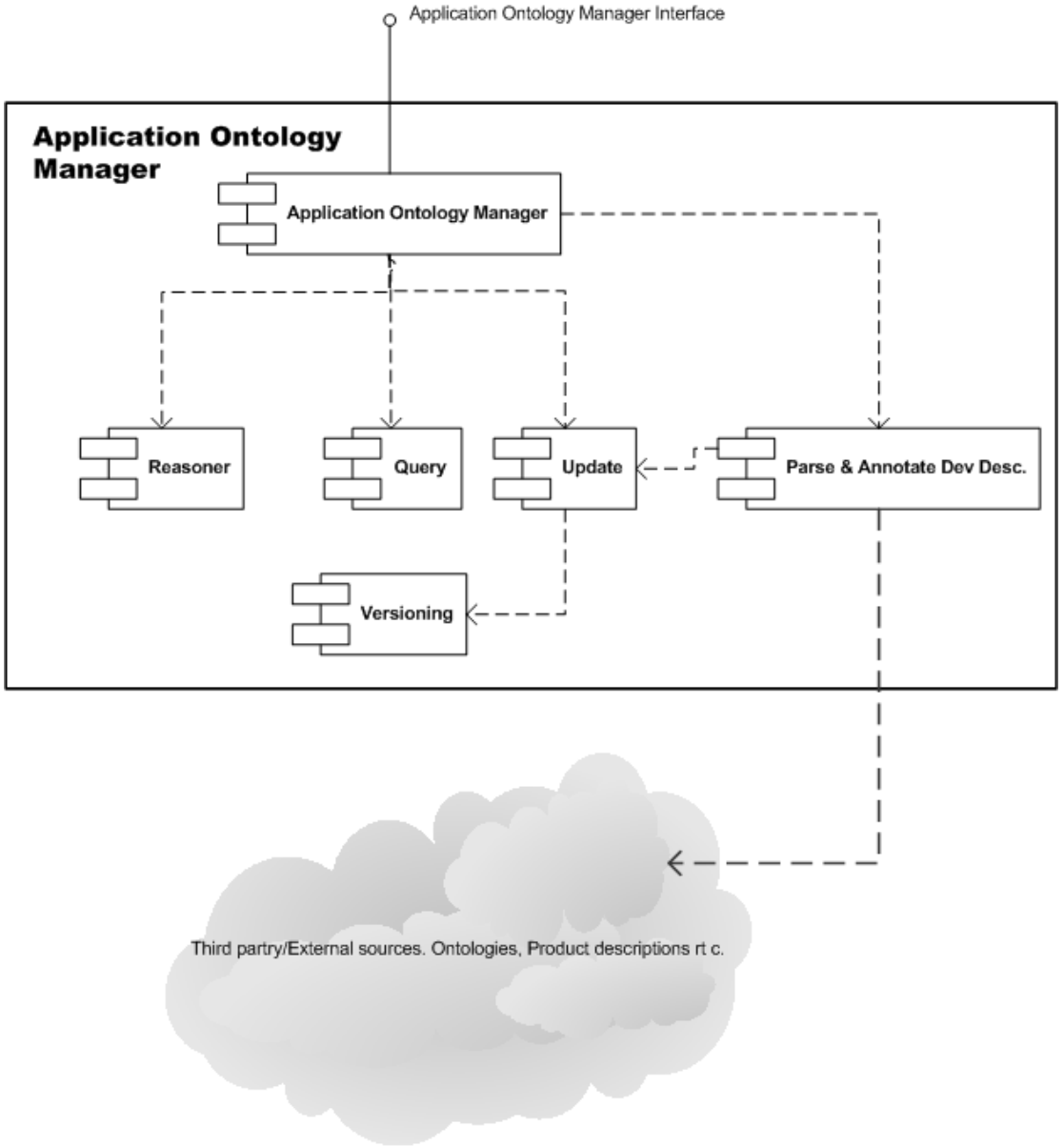


Figure 20: Application Ontology Manager

6.4.3.1 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.

6.4.3.2 Query module

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

6.4.3.3 Update module

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

6.4.3.4 Versioning

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

6.4.3.5 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.

6.4.4 Dependencies

External ontologies and product description databases

6.4.5 Interface

String ApplicationOntologyManager::getDeviceDescription (String *deviceId*)

Retrieves the device description from the ontology.

Parameters:

deviceId The ontology deviceId.

Returns:

A XML string containing the device description or the description of error that occurred during method invocation.

String ApplicationOntologyManager::getDeviceDescriptions ()

Retrieves descriptions of all devices presented in the ontology.

Returns:

A XML string containing the descriptions of all devices contained in the ontology or the description of error that occurred during method invocation.

String ApplicationOntologyManager::createNewDevice (String *deviceType*, String *deviceId*)

Creates the new device ontology instance.

Parameters:

deviceType The device type specified as the name of the ontology concept in the device type hierarchy.
deviceId The requested ontology deviceId.

Returns:

A XML string containing the description of unique and requested ontology id of newly created device instance or the description of error that occurred during method invocation.

String ApplicationOntologyManager::setDeviceDescription (String *deviceId*, String *description*)

Sets or updates the basic device information in the ontology.

Parameters:

deviceId The ontology deviceId.
description The XML string specifying the new basic information of defined device.

Returns:

A XML string containing the update operation success or the description of error that occurred during method invocation.

String ApplicationOntologyManager::getSupplierInfo (String *deviceId*)

Retrieves the device's supplier information from the ontology.

Parameters:

deviceId The ontology deviceId.

Returns:

A XML string containing the information of device supplier (manufacturer name and URL) or the description of error, that occurred during method invocation.

String ApplicationOntologyManager::findDeviceDescription (String *device*)

Retrieves the device's description from the ontology.

Parameters:

device The name of the device.

Returns:

A XML string containing the information of the device.

String ApplicationOntologyManager::parseDeviceDescription (String *description*)

Parses a free-text, or semi-structured, device description and updates the ontology.

Parameters:

description A free-text, or semi-structured, device description of the device.

Returns:

The new deviceId create in the ontology.

String ApplicationOntologyManager::getDeviceClass (String *deviceId*)

Returns the class of a device.

Parameters:

deviceId The deviceId of the device.

Returns:

A string containing the device class.

String ApplicationOntologyManager::resolveError (String *deviceType*, String *error*)

Retrieves the errors for all devices of specified type.

Parameters:

deviceType The device type specified as the name of the ontology concept in the device type hierarchy.
error The human readable error description.

Returns:

A XML string containing the list of all devices, which contain the specified error or the description of error that occurred during method invocation. For each device, the XML string contains the list of matching errors and for each error the related cause-remedy pairs.

String ApplicationOntologyManager::resolveErrorByCode (String *deviceType*, String *error*)

Retrieves the errors for all devices of specified type.

Parameters:

deviceType The device type specified as the name of the ontology concept in the device type hierarchy.

error The error specified by ontology error code.

Returns:

A XML string containing the list of all devices, which contain the specified error or the description of error that occurred during method invocation. For each device, the XML string contains the list of matching errors and for each error the related cause-remedy pairs.

String ApplicationOntologyManager::getDeviceError (String *deviceId*, String *error*)

Retrieves the errors for specified device.

Parameters:

deviceId The ontology deviceId.
error The human readable error description.

Returns:

A XML string containing the list of all device errors matching the query or the description of error that occurred during method invocation. For each error, the XML string contains the list of related cause-remedy pairs.

String ApplicationOntologyManager::getDeviceErrorByCode (String *deviceId*, String *error*)

Retrieves the errors for specified device.

Parameters:

deviceId The ontology deviceId.
error The error specified by ontology error code.

Returns:

A XML string containing the list of all device errors matching the query or the description of error that occurred during method invocation. For each error, the XML string contains the list of related cause-remedy pairs.

String ApplicationOntologyManager::setDeviceErrors (String *deviceId*, String *errors*)

Sets or updates the device errors in the ontology.

Parameters:

deviceId The ontology deviceId.
description The XML string specifying the device errors and cause-remedy pairs.

Returns:

A XML string containing the update operation success or the description of error that occurred during method invocation.

String ApplicationOntologyManager::removeDeviceErrors (String *deviceId*)

Removes the device errors from ontology.

Parameters:

deviceId The ontology deviceId.

Returns:

A XML string containing the remove operation success or the description of error that occurred during method invocation.

String ApplicationOntologyManager::answer (String *query*)

Utility method: retrieves the result of SPARQL query.

Parameters:

query The SPARQL query.

Returns:

A XML string containing the result of executed SPARQL query or the error code. XML string and error codes are automatically generated by Jena API.

String ApplicationOntologyManager::getDeviceTypes ()

Utility method: retrieves the names of device type hierarchy concepts.

Returns:

A string containing comma-separated list of names of device type hierarchy concepts.

6.5 Application Diagnostics Manager

6.5.1 Purpose

The purpose of the Application Diagnostics 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. 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
- Fault detection rule engine
  - manages rules/dependencies over devices

6.5.2 Related WP6 requirements

[HYDRA-91] Any HYDRA device should have an associated description	
Status:	Part of specification
Requirement Type:	Functional
Workpackage:	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-94] Simulation environment	
Status:	Part of specification
Requirement Type:	Functional
Workpackage:	WP6

Rationale:	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.
Source:	WP6 MDA Focus group
Fit Criteria:	Simulation environment is available
Developer Satisfaction:	high
Developer Dissatisfaction:	very high

[HYDRA-96] Detect deadlocks	
Status:	Part of specification
Requirement Type:	Functional
Workpackage:	WP6
Rationale:	The middleware must have functionalities for detecting deadlocks between devices, for instance two devices that are waiting for each other to take an action.
Source:	WP6 MDA Focus Group
Fit Criteria:	Detects deadlocks in 7 out 10 cases
Developer Satisfaction:	very high
Developer Dissatisfaction:	very high

[HYDRA-97] Detect livelocks	
Status:	Part of specification
Requirement Type:	Functional
Workpackage:	WP6
Rationale:	The middleware must be able to detect livelocks between two or more devices, i.e. devices that are constantly changing each others state back and forth.
Source:	WP6 MDA Focus Group
Fit Criteria:	Detects livelocks in 7 out of 10 cases
Developer Satisfaction:	very high
Developer Dissatisfaction:	very high

[HYDRA-98] Detection of device failures	
Status:	Part of specification
Requirement Type:	Functional
Workpackage:	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

[HYDRA-122] Configurable and easy to install middleware	
Status:	Part of specification

Requirement Type:	Non-Functional - usability
Workpackage:	WP6
Rationale:	The middleware should be configurable and easy to install/deploy.
Source:	WP6 MDA Focus Group
Fit Criteria:	The average installation time is less than 1 hour.
Developer Satisfaction:	very high
Developer Dissatisfaction:	high

[HYDRA-125] Transactional updates	
Status:	Part of specification
Requirement Type:	Non-Functional - usability
Workpackage:	WP6
Rationale:	It should be possible to rollback and recover from an unsuccessful update.
Source:	WP6 MDA Focus Group
Fit Criteria:	Rollback works in 7 out of 10 scenarios.
Developer Satisfaction:	high
Developer Dissatisfaction:	high

6.5.3 Components

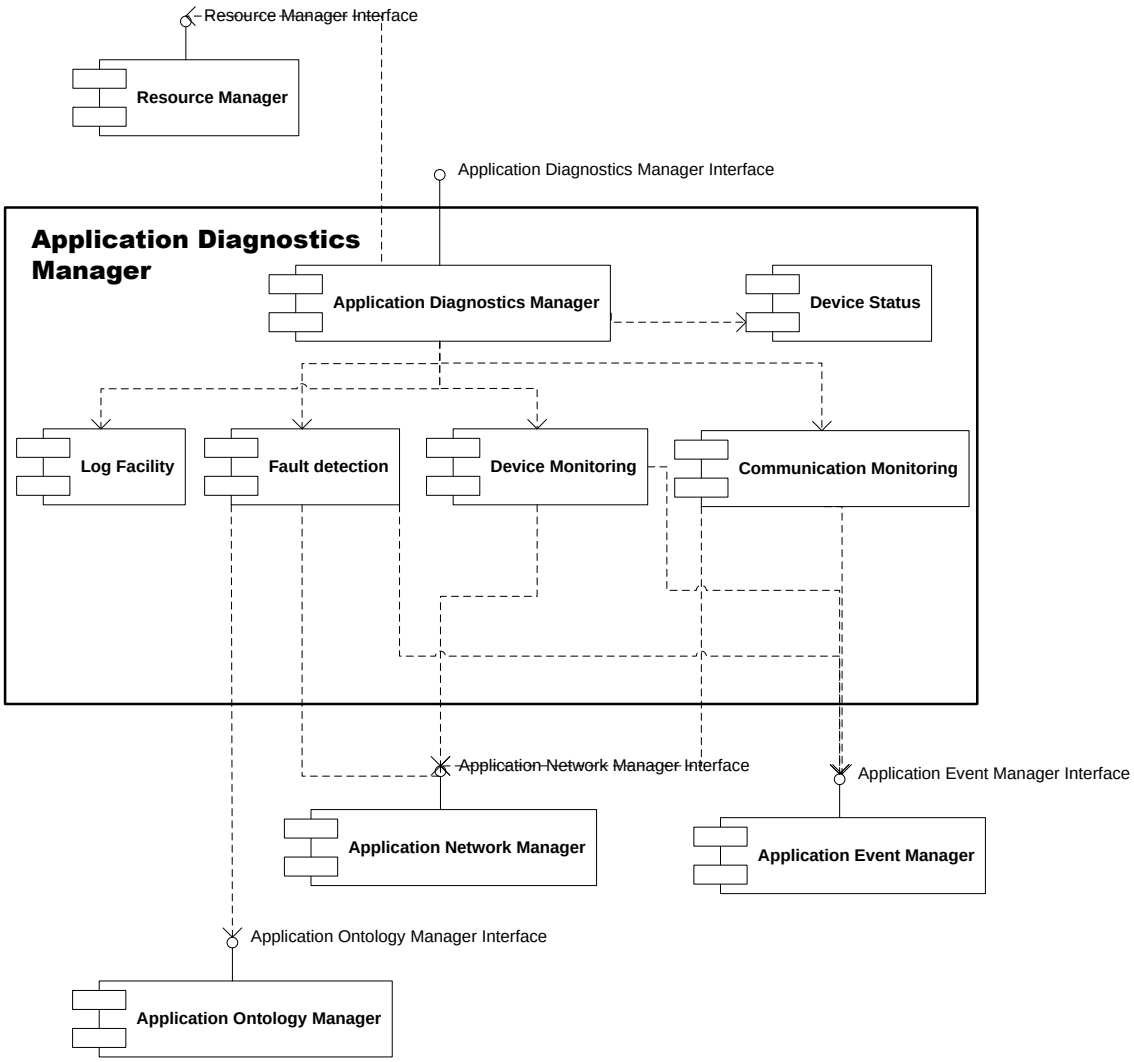


Figure 21: Application Diagnostics Manager

6.5.3.1 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 ResourceManager component in order to get all the interested information.

6.5.3.2 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.

6.5.3.3 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.

6.5.3.4 Device Monitoring

This component is used to process rules or rule sets to monitor devices in order to be preemptive to avoid errors and malfunctions, for instance by monitoring the resource usage of certain devices.

6.5.3.5 Communication Monitoring

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.

6.5.4 Dependencies

Application Ontology Manager, Application Event Manager, Application Network Manager, Resource Manager

6.5.5 Interface

string DiagnosticsManager::checkDeviceCurrent (string *DeviceId*) `[inline]`

Get the current state of a device.

Parameters:

DeviceId The device id as string.

Returns:

A string containing the current state.

string DiagnosticsManager::getDeviceStateMachine (string *DeviceId*) `[inline]`

Get the device state machine.

Parameters:

DeviceId The device id as string.

Returns:

A string containing the current state.

string DiagnosticsManager::checkDevice (string *DeviceId*) `[inline]`

Execute the health monitoring rule for a device.

Parameters:

DeviceId The device id as string.

Returns:

A string containing the status.

string DiagnosticsManager::CheckAllCurrent () `[inline]`

Get all devices current status in order to have a feeling of how the system runs.

Returns:

A string containing the status.

6.6 Device Device Manager

6.6.1 Purpose

The Device Device Manager handles several service requests and manages the responses.

Main Functions:

- Maps requests to device services
- Response generation
- Advertising Hydra device description
- Advertises device services

6.6.2 Related WP6 requirements

[HYDRA-91] Any HYDRA device should have an associated description	
Status:	Part of specification
Requirement Type:	Functional
Workpackage:	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
Workpackage:	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

Workpackage:	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
Workpackage:	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
Workpackage:	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
Workpackage:	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

Workpackage:	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
Workpackage:	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

6.6.3 Components

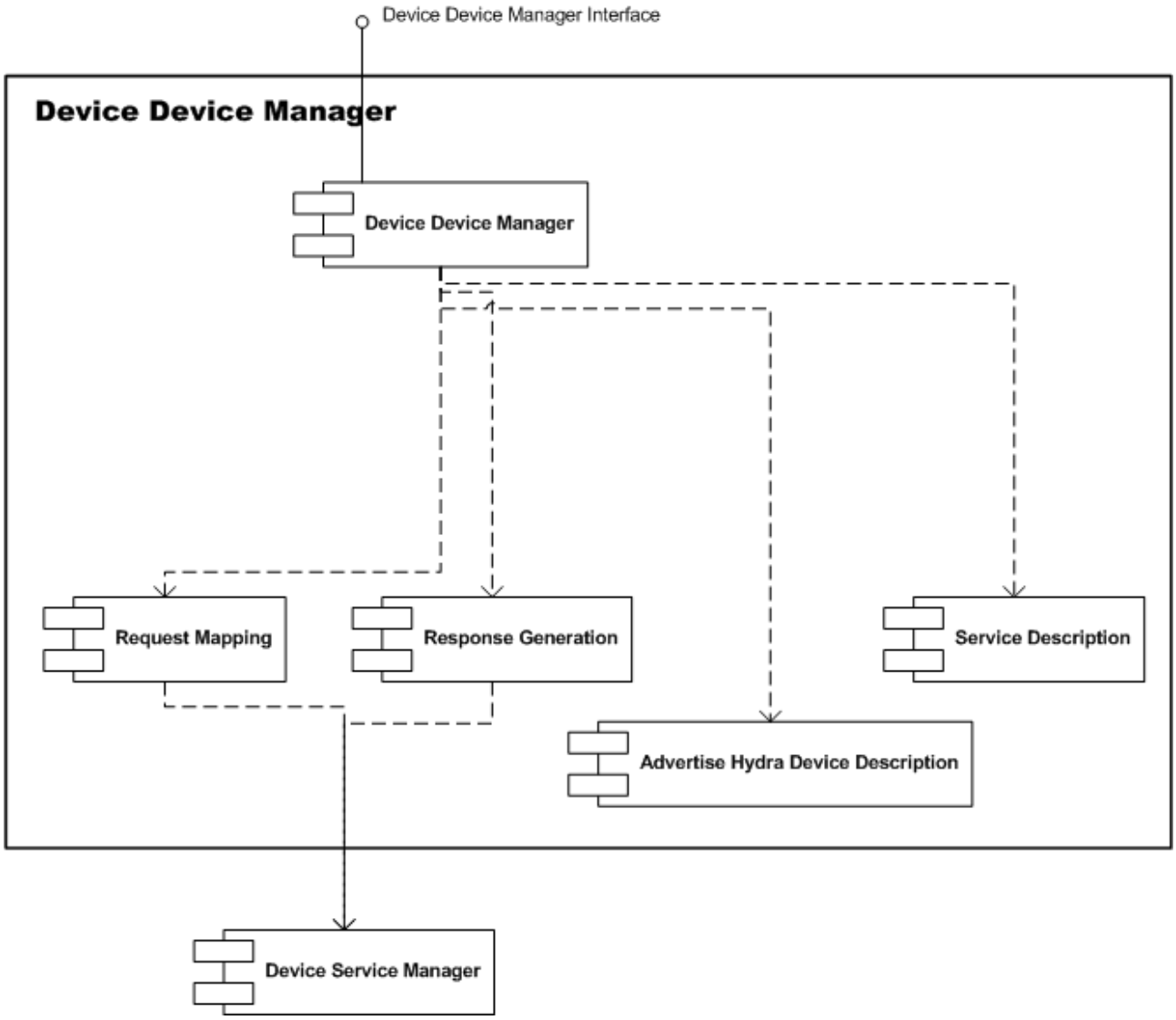


Figure 22: Device Device Manager

6.6.3.1 Advertise

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

6.6.3.2 Request Mapping

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

6.6.3.3 Response Generator

This module maps translates the result of an internal service in the device to a response to the caller.

6.6.3.4 Service Description

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

6.6.4 Dependencies

Device Service Manager

6.6.5 Interface

string DeviceDeviceManager::RegisterError(string *property*, string *errorcode*)

Registers an error condition.

Parameters:

property The error property as string.
errorcode The error code as string.

Returns:

A string containing the registered error.

string DeviceDeviceManager::SendErrorMessage(string *message*)

Sends an error message for a specific device.

Parameters:

message The error message as string.

Returns:

A string containing the sent error message.

string DeviceDeviceManager::Invoke(string *serviceid*, string *methodName*, string *parameters*, string *values*)

Executes a specific method for a service (using the device service manager).

Parameters:

serviceid The serviceid as string.
methodName The methodName as string.
parameters A comma delimited string with the parameter names.
parameters A comma delimited string with the parameter values (Matched against "parameters").

Returns:

A string indicating the result of the execution.

string DeviceDeviceManager::GetDeviceStatus ()

Retrieves the device status (using the device service manager).

Returns:

A string with the device status.

string DeviceDeviceManager::AddDAC(string dacaddress)

Adds a device application catalogue to this device list of catalogues when the device is discovered.

Returns:

A string with the device status.

string DeviceDeviceManager::GetDACList()

Returns the list of device application catalogues where the device has been discovered.

Returns:

A string with the device application catalogues.

6.7 Device Service Manager

6.7.1 Purpose

Implements service interface for physical devices.

Main Functions:

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

6.7.2 Related WP6 requirements

[HYDRA-120] Multiple Device Virtualisations	
Status:	Part of specification
Requirement Type:	Functional
Workpackage:	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
Workpackage:	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

6.7.3 Components

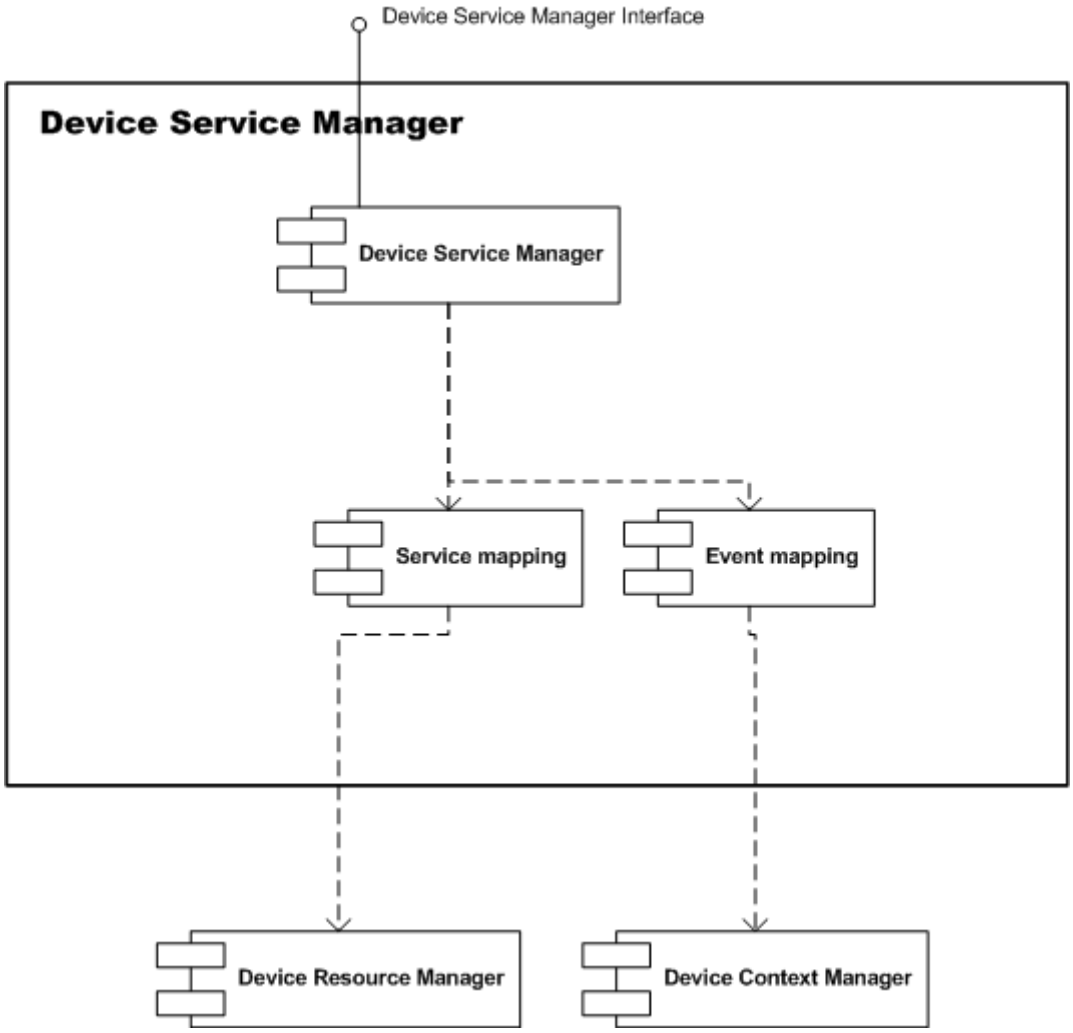


Figure 23: Device Service Manager

6.7.3.1 Service Mapping

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

6.7.3.2 Event Mapping

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

6.7.4 Dependencies

Device Resource Manager, Device Context Manager

6.7.5 Interface

XmlDocument DeviceServiceManager::GetServiceDescription (string *serviceid*)

Retrieves a service description for a service.

Parameters:

serviceid The service id as string.

Returns:
An XmlDocument with service description.

string DeviceServiceManager::Invoke(string *serviceid*, string *methodName*, string *parameters*, string *values*)

Executes a specific method for a service.

Parameters:
serviceid The serviceid as string.
methodName The methodName as string.
parameters A comma delimited string with the parameter names.
parameters A comma delimited string with the parameter values (Matched against "parameters").

Returns:
A string indicating the result of the execution.

string DeviceServiceManager::GetDeviceStatus ()

Retrieves the device status (using the device service manager).

Returns:
A string with the device status.

string DeviceServiceManager::GetProperty(string property)

Retrieves the property value for a given property of the device.

Parameters:
propertyid The propertyid as string.

Returns:
A string with the property value.

string DeviceServiceManager::HandleEvent(string event)

Process an event from the physical device and maps that into a Hydra Event.

Parameters:
event The event as string.

Returns:
A string representing the Hydra event.

6.8 Common XML-Schema

There will be a common XML-Schema derived from the contents and structure of the different ontologies. The schema classes will be used in the different manager web service interfaces and internal interfaces. The purpose of the common XML-Schema is to standardize the representation of different common entities such as the HID, device et c in order to make it interoperable in-between managers. The schema will also provide the base for implementing the data structures that will be used and referenced by the SDK environment.

7. Future work

Future work within this workpackage includes integrating a security ontology in the semantic Model Driven Architecture in cooperation with workpackage 7, as well as further extending the work on software components ontology.

Since the Hydra semantic MDA is centered around ontologies we foresee a need for tools and methods for design and managing these ontologies, therefore for the next development iterations we will investigate how the design and management of the three Hydra ontologies can be carried out efficiently.

7.1 Device Discovery

There are several issues to be further investigated for the management of the DAC and the discovery process. In this (2nd) iteration we have decided that all service composition will occur at design time. In the following iterations, the Hydra middleware may have to resolve 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. The Hydra discovery functions will be 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. However, not all Hydra devices will be capable of this. The more limited devices will be able to handle web services (in order to be Hydra devices), and these may also need some way of announcing themselves on the network.

7.2 Security ontology

The further specification and use of a security ontology is under investigation in cooperation with WP7.

7.3 SW components ontology

The purpose of a 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[2]. We will use this as a basis for a 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 an ontology.

7.4 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

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

ontology management system should be a framework for creating, modifying, versioning, querying, and storing ontologies. It should allow an application to work with an 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.

7.4.1 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.

7.4.2 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. These 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

7.4.2.1 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 a 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).

7.4.2.2 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.

7.4.3 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 an external (device) ontology and be provided with tools for its adaptation and use in application development.

8. 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/.

[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 Markup 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.