Contract No. IST 2005-034891

Hydra

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

D12.9 - Final External Developers Workshops Teaching Materials

Integrated Project
SO 2.5.3 Embedded systems

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

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

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

Dissemination Level: Public

Document file: D12.9 - Final External Developers Workshops Teaching Materials - 1.0 - final.doc

Work package: WP12 - Training

Task: T12.1

Document owner: Atta Badii (University of Reading)

Document history:

0.1	Atta Badii, Junaid Raja Khan, Michael Crouch, Sebastian Zickau (UR)	12-03-2010	Initial TOC
0.2	Atta Badii, Junaid Raja Khan, Michael Crouch, Sebastian Zickau (UR)	17-03-2010	Restructured TOC in preparation for consortium discussion and assigning of responsibilities
0.3	Atta Badii, Junaid Raja Khan, Michael Crouch, Sebastian Zickau (UR)	24-03-2010	Start to add content and source references
0.5	Peeter Kool, Matts Ahlsen (CNet), Sascha Effert (UP), Pablo Antolin Rafael, Francisco Milagro Lardies (TID), Julian Schuette, Tobias Wahl (SIT), Andreas Zimmermann (FIT), Klaus Marius Hansen (UAAR)	22-04-2010	Added contributions by partners, used partner's material
0.7	Atta Badii, Junaid Raja Khan, Michael Crouch, Sebastian Zickau (UR)	29-04-2010	Revision and added content: Commons, Event Context and Policy Frameworks (IDE, SDK)
0.8	Atta Badii, Junaid Raja Khan, Michael Crouch, Sebastian Zickau (UR)	30-04-2010	Formatting, merging, cleaning
0.9	Atta Badii, Junaid Raja Khan, Michael Crouch, Sebastian Zickau (UR)	14-05-2010	Including reviewer's comments
1.0	Atta Badii, Junaid Raja Khan, Michael Crouch, Sebastian Zickau (UR)	14-05-2010	Final version submitted to the EU

Internal review history:

Reviewed by	Date	Comments
Jorge Irazola, Andrea Guarise (INN)	13/05/2010	See comments below
Julian Schütte (SIT)	13/05/2010	See comments below

Table of Content

1. Executive summary 9

2. Introduction 10

2.1 Purpose, context and scope of this deliverable 10
2.2 Background 10

3. Hydra Architecture 11

3.1 Device Classification 12
3.2 Applications and Devices 13
3.3 Applications 13
3.4 Hydra Devices 13
3.5 Semantic Devices 14
3.6 Hydra Apps 14
3.7 Application Templates 14
3.8 The SDK and the DDK 15

4. Installing the Hydra Middleware 16

4.1 Prerequisites 16
4.2 Required Bundles 16
4.2.1 Core required bundles 16
4.2.2 Hydra bundles 17
4.3 Crypto Manager Setup 17
4.4 VM Arguments 18
4.5 Running the framework 18

5. Software Development Kit 21

5.1 Hydra Commons 21
5.1.1 Hydra Middleware API 21
5.1.2 Hydra Middleware Clients 21
5.1.3 Hydra Configurator 22
5.2 Network Manager 24
5.2.1 Hydra Definition of Device To Device Communication 24
5.2.2 The Peer-to-Peer Network Architecture 25
5.2.3 Purpose 26
5.2.4 Main Functionalities 26
5.2.5 Hydra Web Service Provider 26
5.2.6 Crypto HIDs 28
5.3 Device Application Catalogue 31
5.3.1 The DAC browser 31
5.3.2 The Graphical Browser 32
5.4 Discovery Manager (Framework) 38
5.4.1 Physical Discovery 39
5.4.2 External Discovery 40
5.4.3 Semantic Discovery 41
5.5 Ontology Manager 42
5.6 Event Manager 44
5.7 Context Awareness Framework 46
5.7.1 Context Manager 46
5.7.2 Contexts 47
5.7.3 Queries 48
5.7.4 Context-sensitive Actions 48
5.7.5 Data Acquisition Component 49
5.7.6 Subscriptions 49
5.7.7 Plausibility Checking 50
5.7.8 Data Reporting 50
5.8 Access Control Policy Framework 51

5.8.1 Policy Enforcement Point 51
5.8.2 Policy Decision Point 52
5.8.3 Policy Administration Point 52
5.8.4 Policy Information Point 53
5.9 Quality of Service Manager 53
5.9.1 Functionalities 54
5.9.2 Dependencies 54
5.9.3 Used by 54
5.9.4 Prerequisites 54
5.9.5 Installation 54
5.10 Execution in eclipse 59
5.10.1 Usage 60
5.11 Storage Architecture 62
5.11.1 Implementation details 63
5.11.2 Storage Manager Device 64
5.11.3 API 65
5.11.4 Client 66
5.11.5 Command Line Client 67
5.11.6 File System Devices 67
5.11.7 API 67
5.11.8 File System Device types 69
6. Device Development Kit 71
6.1 DDK Components and Tools 71
6.1.1 Limbo 71
6.1.1 Obtaining and Installing Limbo 71
6.1.2 Describing the service in a WSDL file 72
6.1.3 Describing the service-related statemachine 74
6.1.4 Run the Limbo compiler on the WSDL file 75
6.1.5 Implement and deploy the device-specific service 75
6.1.6 Running the Generated Code 76
6.2 Device Ontology 76
6.2.1 Device Creator 77
6.3 Flamenco 80
6.3.1 System Requirements and Installation 80
6.3.2 Design Time Usage 80
6.3.3 The auxiliary page 81
6.3.4 The net 81
6.3.5 The declarations 81
6.3.6 Flamenco/SW 82
6.4 Device Discovery Manager 88
6.5 Hydra-Enabling a Device 89
7. Integrated Development Environment – Java 95
7.1 Network Manager IDE 95
7.1.1 IDE connection 95
7.1.2 Remote connection 95
7.1.3 Hydra Status and Configuration views 97
7.2 Trust Manager IDE 100
7.3 Crypto Manager IDE 103
7.4 Context Manager IDE 105
7.4.1 Context Specifications 105
7.4.2 Context Queries 112
7.5 Obligation Framework IDE 113
7.5.1 Obligation GUI 113
7.6 Access Control Policy Framework IDE 116
7.7 Device Application Catalogue IDE 121
7.8 Installing Limbo in IDE 123

8. Integrated Development Environment - .Net 128
8.1 Creating a Basic Hydra Application 128
8.2 Creating a Hydra application from a template 128
8.2.1 Initiating the Network Manager 129
8.2.2 Initiating the Application Device Manager 129
8.2.3 Working with devices 130
8.2.4 Applications Bindings 131
8.3 Creating an Advanced Hydra Application 132
8.3.1 Initiate Application 132
8.3.2 Searching and finding for devices 133
8.3.3 Invoking Device Services 135
8.4 Understanding the Hydra Device XML 135
8.4.1 Extending the Hydra Device XML 138
8.5 SDK components 138
8.5.1 Application Project Templates 139
8.5.2 HydraBasicApplication 139
8.5.3 HydraEnergyApplication 139
8.5.4 HydraDynamicApplication 139
8.5.5 HydraSensorApplication 139
8.6 Tools integration 140
8.6.1 The DAC browser 140
8.6.2 The Device Ontology browser 140
8.7 SDK Class library for .NET 141
8.7.1 Using the .Net DDK tools 141
8.7.2 Using Intel Service Author for UPnP Technologies 141
8.7.3 Using Hydra .Net DDK tool 144
9. Summary 151
10. References and further Reading 152
11. Glossary 155

Figures

Figure 1: Hydra Architecture (layer model) 11
Figure 2: Flowchart for the device classification process. 12
Figure 3: Run Configuration (Hydra Bundles) 17
Figure 4: Eclipse Run Configuration (Arguments) 18
Figure 5: Network Manager Status page in browser 19
Figure 6: Hydra Status page 20
Figure 7: List of services (in browser) 20
Figure 8: Hydra Status page screenshot 23
Figure 9: Introduction of a property at the XML file of a declarative service 27
Figure 10: Querying the Hydra network for a HID matching some attributes 30
Figure 11: DAC Browser (upper right) in the IDE 31
Figure 12: The Hydra Browser 32
Figure 13: Retrieving discovery information from the physical device 33
Figure 14: Discovery information from a Bluetooth Device 34
Figure 15: Resolving a physical device into a Hydra Device. 35
Figure 16: Resolve information is sent as an XML structure to the Discovery Manager 35
Figure 17: A physical device with unknown functionality has been transformed into Basic Phone Device with services for reading/sending SMS. 36
Figure 18: Sending an SMS through the Basic Phone Device 36
Figure 19: Using the DAC browser to retrieve a WSDL description for the device. 37
Figure 20: A WSDL (Web Service Description Language) for the device 38
Figure 21: 3-layered discovery architecture in Hydra 39
Figure 22: Ontology State Machine Concepts 43
Figure 23: Ontology Browser 44
Figure 24- Event Manager Interface 45
Figure 25: Event Manager Deployment 46
Figure 26: Subscriber Notification 46
Figure 27: Select project 55
Figure 28: Ontology Browser 56
Figure 29: Repository Manager 57
Figure 30: Dialog Box 57
Figure 31: Hydra Status Page 58
Figure 32: Bundles for QoS Manager 59
Figure 33: Basic architecture of storage in Hydra 63
Figure 34: Some examples for Responses 64
Figure 35: API of a Hydra Storage Manager Device 65
Figure 36: API of HydraSMConnector 66
Figure 37: API of a Hydra File System Device 69
Figure 38: Basic architecture of file system 70
Figure 39: Dummy state machine 74
Figure 40: The Device Browser tab. 77
Figure 41: Adding a new instance. 77
Figure 42: Device editing functionality 78
Figure 43: CPN Tools 81
Figure 44: Flamenco 82
Figure 45: Device Protégé 84
Figure 46: Device Rules Protégé 85
Figure 47: Flamenco Planning Layer 87
Figure 48: Create Visual Project 89
Figure 49: Visual Studio – editing file 90
Figure 50: Visual Studio (WebServices) 91
Figure 51: Build application 93
Figure 52: Adding breakpoint 94
Figure 53: Hydra Middleware Connection configuration page 96
Figure 54: Remote connection button 96

Figure 55: Network manager status views 98
Figure 56: Event Manager Status view 99
Figure 57: Hydra Configurator view 100
Figure 58: TrustManager GUI showing the details of a X509v3 certificate 101
Figure 59: Adding a new trust root to using the TrustManager GUI 102
Figure 60: Validating a certificate using the TrustManager GUI (Trust Manager IDE) 102
Figure 61: CryptoManager View...104
Figure 62: Certificate generation wizard 105
Figure 63: Create Context Specification Wizard 106
Figure 64: Context Definition page 107
Figure 65: Data Subscription page..108
Figure 66: Context Rules - Imports and Types 109
Figure 67: Context Rules - Functions 110
Figure 68: Context Rules - Rules LHS...110
Figure 69: Publish Event to Event Manager Wizard 111
Figure 70: Context Queries..113
Figure 71: The Obligation GUI perspective. EventListener view on the left, message console on the bottom and event editor in the middle. 114
Figure 72: Situation editor (empty list of policies on the right) 116
Figure 73: The Policy IDE Dashboard 117
Figure 74: Creating a new Policy 118
Figure 75: Content Assist for selecting Rule Combining Algorithm....................................119
Figure 76: Content Assist selecting Data Type...119
Figure 77: Content Assist in adding new root Policy XACML Elements...............................120
Figure 78: XACML policy with an invalid attribute value 120
Figure 79: XACML Schema Validation reporting errors 121
Figure 80: Device Application Catalogue View 122
Figure 81: DAC configuration preference page 123
Figure 82: Limbo wizard selection in Eclipse..123
Figure 83: Limbo wizard starting point 124
Figure 84: Limbo wizard options..125
Figure 85: Limbo wizard, selecting output 125
Figure 86: Limbo wizard, output directory 126
Figure 87: Template view in Visual Studio 128
Figure 88: Auto generated files for Basic Hydra Application...129
Figure 89: Creating WS clients for device 131
Figure 90: Energy Application Template view 133
Figure 91: Selecting web references to devices 134
Figure 92: DAC Browser (upper right) in the IDE 140
Figure 93: The web-based Device Browser..141
Figure 94: Producing SCPD window..142
Figure 95: Action tab 143
Figure 96: Action Editor...143
Figure 97: Save file window 144
Figure 98: Add new device window 144
Figure 99: Window for setting name and other properties...145
Figure 100: Adding service window...145
Figure 101: Choosing a file in explorer 146
Figure 102: Obex service window 146
Figure 103: Generating a Hydra device dialogue...147
Figure 104: Code Generation Window 147
Figure 105: Hydra .Net-IDE 148
Figure 106: Hydra .Net IDE 149
Figure 107: DAC with example SmartPhone device 150

1. Executive summary

This deliverable D12.9, titled - Final External Developers Workshops Training Materials, is intended to serve external solution and device developers as well as application integrators who wish to examine how to use, set up and configure Hydra components and how these components can be integrated to form the Hydra middleware platform plus the benefits that they could offer for solving specific design and requirement challenges.

D12.9 is a sequel deliverable to D12.8 and brings together the high level architectural and sub-system level descriptions from D12.8 with the lower level implementation, functionality and how-to-configure details as used to build applications based on the Hydra Middleware Platform.

The task of developing internal and external training materials has been an ongoing process during the Hydra project. Internal training material has been in use by the Partners and people who are interested in the technical aspects of the project. Training materials have been subjected to periodic updates as informed by the feedback from the training sessions conducted to-date as well as revisions and new additions that have been added as they have become available throughout the course of the project. In particular the technical details regarding architectural and implementation issues have been updated continuously to reflect the latest developments and improvements as following the iterative cycles of design refinement and re-engineering as the project has evolved.

During the current phase the design and implementation decisions have been finalised and the technologies used within Hydra are now consolidated into what will be delivered as the final version of Hydra Platform at the end of the project. Within its three sections; namely Hydra Technologies, Hydra Components, and Hydra Tools, this document covers the following five areas:

i) How-to use underlying technologies used for the realisation of the components of the Hydra Platform

ii) A Hydra system setup description on how to establish a Hydra environment

iii) A description of Hydra components and their configuration and usage

iv) A description of the various tools (SDK, DDK and IDE) used in Hydra and how they are used to develop Hydra applications and a Hydra-enabled environment

v) General concepts and technologies specifically related to Hydra.

2. Introduction

2.1 Purpose, context and scope of this deliverable

This deliverable D12.9, titled - Final External Developers Workshops Training Materials, brings together a collection of descriptions, tutorials and how-to-dos, to teach on using the Hydra Middleware. It is aimed at a target audience of external developers, and as such is at a sufficiently technical level, describing the interfaces of each Hydra component, and how to use them.

D12.9 is the final update to D12.5 featuring all the latest and most advanced features of the Hydra middleware from the perspective of a third-party developer. The target audience is thus application and device developers.

2.2 Background

Following on from the Executive Summary and Introduction, an overview of the Hydra Research and Development and Technology Development (RTD) Objectives is given in Chapter 3, including an overview of the Hydra concepts and architecture highlighting the main design principles of the middleware.

Chapter 4 presents the explanation of installation and prerequisites for setting up the Hydra middleware.

Chapter 5 sets out a detailed description of how to configure the different SDK components of Hydra and how the enabling technologies have been used thus adding value for the whole middleware platform.

Chapter 6 explains the DDK and the set up of Hydra tools and concepts and how they can be deployed when using Hydra to develop applications.

Chapter 7 is an intensive chapter focussed primarily on Java IDEs for the various components in the Hydra middleware.

Chapter 8 focuses exclusively on .NET IDE development and configuration.

Chapter 9 presents summarises the document and provides the conclusions.

Further reading suggestions and useful sources are given in the reference section in Chapter 10 whereas Chapter 11 provides a glossary with relevant Hydra terminology.

3. Hydra Architecture

The software architecture described is an abstract representation of the software part of the Hydra middleware. The architecture is a partitioning scheme, describing components and their interaction with each other. Figure 1 gives a structural overview of the Hydra middleware and explains how the elements are logically grouped together. "Hydra Managers" constitute the major building blocks that make up the Hydra middleware. A Hydra manager encapsulates a set of operations and data that realise a specific functionality.

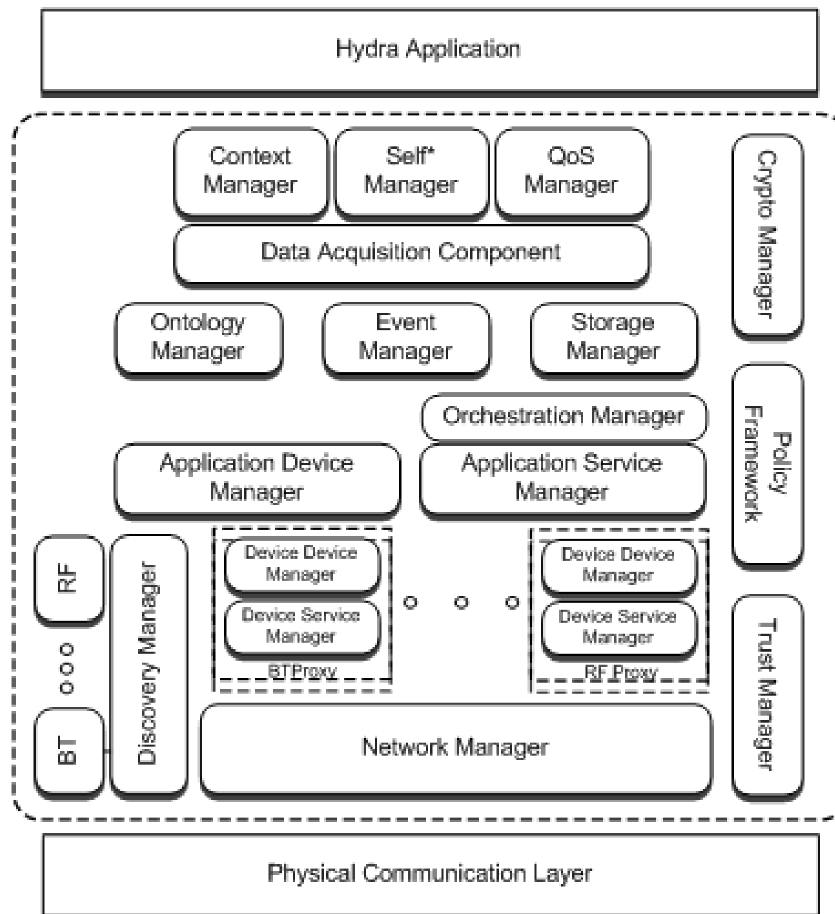

Figure 1: Hydra Architecture (layer model)

The Hydra middleware managers are enclosed by the physical communication layer and the application layer shown at the bottom and at the top of the diagram respectively. The physical layer realizes several network connection technologies such as ZigBee, Bluetooth or WLAN. The application layer contains user applications which could comprise modules such as workflow management, user interface, custom logic and configuration details. These two layers are not part of the Hydra middleware.

The Hydra middleware offers a large collection of reusable core software components to experienced developers. Based on these software components, programming abstractions allow for programming with well-known concepts from the field of pervasive and ambient computing by means of reducing the details of the underlying implementation. From the bottom to the top of Figure 1 the Hydra middleware provides more and more programming abstraction and functionality for the developers:

- The Network Manager implements Web Service over JXTA as the Peer-to-Peer model for device-to-device communication.
- The Device and Device Service Manager in a bundle implement a service interface for a physical device, handle several service requests and manage the responses.
- The Application Device and Application Service Manager provide programming interfaces and information for the different devices to the software developers.
- The Discovery Manager automates and facilitates the discovery of devices in a Hydra network.
- The Ontology Manager is used by the Application Device Manager to get meta-information about devices and also semantically resolves what type of device has been discovered.
- The Event Manager provides a topic based publish-subscribe service in Hydra.
- The Crypto, Trust and Policy Manager takes care of cryptographic operations, the evaluation of trust in different tokens and the enforcement of access control security policies.
- The Data Acquisition Component retrieves the data delivered by the sensors (via push or pull mode).
- The Quality-of-Service (QoS) Manager in Hydra is a component that accesses and particularly processes all non-functional properties-data for services/components, devices, and networks.
- The Self* Manager provides support for automating application management.
- The Context Manager allows for the definition of an application-dependent context model.
- The Hydra Storage Architecture realises the persistent storage of information in Hydra middleware.

3.1 Device Classification

The Hydra middleware is designed to handle all types of devices, with varying capabilities. The figure below, demonstrates how devices are classified into different categories, based on what technologies they can support, which determines how the device can become "Hydra-enabled" (see glossary in chapter 11 for terms used in Hydra).

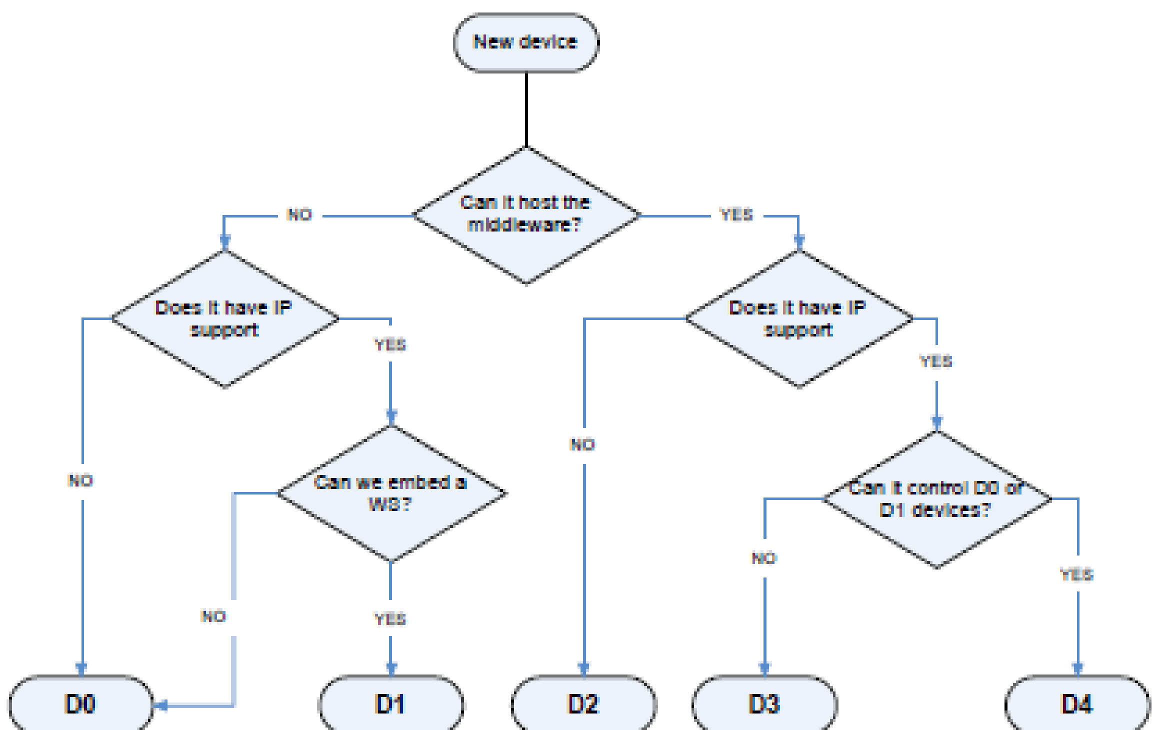

Figure 2: Flowchart for the device classification process.

The significance of the D0--D4 categories is that devices within each category are handled in the same way in relation to the Hydra middleware and the enabling process. For further details on Hydra terminologies please refer to the Glossary section in Chapter 8.

Category-D0 devices are used with a proxy, that is, they can only be reached through a proxy-service residing on a Category-D4 device. The proxy service must implement the communication with the D0 device.

Category-D1 devices can host a web service, and the intention is that such embedded web services are created with the Limbo tool (see chapter 6.1.1)

Category-D2--D4 devices are said to be Hydra enabled. Hydra enabled devices host the network manager and all other managers needed for that device, but differ in their networking capabilities.

3.2 Applications and Devices

Hydra applications are built by programming networked ambient intelligent devices. Devices are made programmable by the Hydra middleware by means of proxies as well as by embedded components. Whatever the method, it is transparent to the application developers, as they access all devices based on a pure service and event based programming model. The API of this programming model is manifested by the Hydra SDK, for application development.

3.3 Applications

An application in Hydra is built around a DAC (a Device Application Catalogue) which functions as a kind of device registry, holding references to the set of devices which has been discovered and are available to the application.

Hydra provides different levels of configuration, depending on the application requirements. A minimal configuration for a Hydra application consists of an Application Device Manager, and a Network Manager running on a Hydra gateway device (aka D4 device), to which one or more other devices are connected or can be connected.

The minimal configuration can be extended by an Ontology Manager, which will add semantic discovery capability to the system. Additional functionality for context management and security can be obtained by the corresponding managers (Context Manager, see 5.7, Security Manager, see 5.2.6, 5.8, 7.2).

3.4 Hydra Devices

A basic idea in Hydra is to differentiate between the physical devices and the application view of the device, in terms of so called Hydra Devices.

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

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

There are five categories of Web Services generated for a Hydra Device,

- A Generic Hydra web service, exposing metadata and management functions common to all Hydra Devices.
- An Energy web service, providing a set of functions for the monitoring and control of energy consumption of devices.

- A Memory Service which allows logging and storing of device internal data such as state variables and energy consumption data.
- A Location Service which can be used to query the device about its location and position.
- A device type specific web service, exposing the device type specific functions

3.5 Semantic Devices

Based on Hydra Devices, the SDK provides the concept of a *Semantic Device* as an application development construct. This allows a programmer to develop new application specific adaptations of the available Hydra Devices.

The services offered by the Hydra devices have been designed independently from the particular applications in which the device might be used, e.g., 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. A semantic device on the other hand represents what the particular application would like to have. We call these logical aggregates of devices and their services for *Semantic Devices*.

A semantic device instantiates itself dynamically on-the-fly, when appropriate physical devices are discovered. A simple example is a programmer that creates a semantic device, "intelligent meat thermometer" based on two physical devices – meat thermometer and a DLNA-enabled TV. The semantic device takes sensor data from the meat thermometer and uses the display capabilities of the DLNA-enabled TV. The programmer expresses some behaviour rules, what should happen when the thermometer reaches certain temperatures. His cool new device will then allow a user to put a steak into the oven and then go watch TV. When the meat temperature reaches 54 degrees the user is alerted with a picture of red meat that is shown on the TV, at 58 degrees a medium cooked steak is shown and at 60 degrees we see a well-done steak. The programmer could also add some monitoring rules, to check if the temperature is raising to fast etc and in that case advice the user to turn down the oven temperature. The TV could be delivered with a whole catalogue of semantic devices which regularly check the network to instantiate themselves. If the user then buys a thermometer and brings it home he is informed that this can be used together with the TV and asked if he wants to install this facility. Since semantic devices are Hydra Devices themselves they can then be recursively combined, and also be discovered by other Hydra DAC.

3.6 Hydra Apps

Hydra Apps are ICT services that users can buy or subscribe to in order to solve specific needs in both private and professional settings, such as vital signs monitoring, energy optimisation, smart home control and more. The Semantic Device construct is a basis for delivering such "Apps" on the Hydra platform. Thus, a Hydra App is a semantic device which has been tailored for some application specific purpose, by a Hydra developer. In principle a Hydra App can be seen as a semantic device designed for a specific purpose and with a user/client interface.

3.7 Application Templates

In order to facilitate automation of developments work, the Hydra SDK includes different types of templates. Among these are a number of application templates,

- Template for a Basic Application: most generic type of application, minimal.
- Template for a Energy Application: monitoring and control of a set of homogeneous devices in a local network.
- Template for a Dynamic Application: a generic application using devices of a certain class at run-time, i.e., not bound to specific devices types by design time.
- Template for a Sensor Application: a generic event-based application

The templates are integrated in the host IDE. The following sections describe the principal use of these templates (see 7)

3.8 The SDK and the DDK

Whereas the SDK is focused on the development of applications of devices, the purpose of the DDK is to adapt various physical devices for use by the Hydra developers. The DDK is described in Chapter 6. Many elements of the Hydra platform are of course common to both the SDK and the DDK. Among them are the Device Application Catalogue (DAC) and the Ontology Manager. The SDK and the DDK are integrated to form the Hydra IDE.

4. Installing the Hydra Middleware

This chapter discusses the requirements for installation of the Hydra Middleware, using the Equinox (Eclipse) implementation of OSGi. The Hydra Middleware will be provided as a stand-alone package which can be run in any generic OSGi framework, not only depending on the eclipse IDE. (Please see chapter 10 for additional information and URLs)

4.1 Prerequisites

Some bundles are not provided with Eclipse (Galileo - 3.5 and earlier), and so will need to be downloaded and placed in the *plugins* folder of your Eclipse installation, if the Hydra Middleware is to be launched from within the Eclipse environment, which is not compulsory.

Most significantly this may include

```
org.eclipse.equinox.cm_1.0.100.v20090520-1800
org.eclipse.equinox.ds_1.1.1.R35x_v20090806
```

4.2 Required Bundles

The following chapters specify the various OSGi bundles required to launch the Hydra Middleware. This includes both core bundles, and the bundles of Hydra Managers and components. The configuration provided here is a very basic one, and extra Hydra bundles can be added to include their functionality.

4.2.1 Core required bundles

These external bundles have to be added in the run configuration:

```
javax.servlet_2.5.0.v200806031605
javax.xml_1.3.4.v200902170245
org.apache.commons.codec_1.3.0.v20080530-1600
org.apache.commons.httpclient_3.1.0.v20080605-1935
org.apache.commons.lang_2.3.0.v200803061910
org.apache.commons.logging_1.0.4.v200904062259
org.apache.log4j_1.2.13.v200903072027
org.apache.xalan_2.7.1.v200905122109
org.apache.xml.serializer_2.7.1.v200902170519
org.eclipse.equinox.cm_1.0.100.v20090520-1800
org.eclipse.equinox.ds_1.1.1.R35x_v20090806
org.eclipse.equinox.http.jetty_2.0.0.v20090520-1800
org.eclipse.equinox.http.servlet_1.0.200.v20090520-1800
org.eclipse.equinox.util_1.0.100.v20090520-1800
org.eclipse.osgi.services_3.2.0.v20090520-1800
org.eclipse.osgi_3.5.1.R35x_v20090827
org.mortbay.jetty.server_6.1.15.v200905151201
org.mortbay.jetty.util_6.1.15.v200905182336
```

4.2.2 Hydra bundles

These Hydra bundles have to be added in the run configuration (see chapter 10 for download links):

```
CryptoManager_1.1.0
HydraManagerConfigurator_1.0.0.qualifier
HydraMiddlewareAPI_1.0.0.qualifier
HydraMiddlewareClients_1.0.0.qualifier
HydraWSProvider_1.0.0.qualifier
Network_Manager_Bundle_1.7.0.qualifier
```

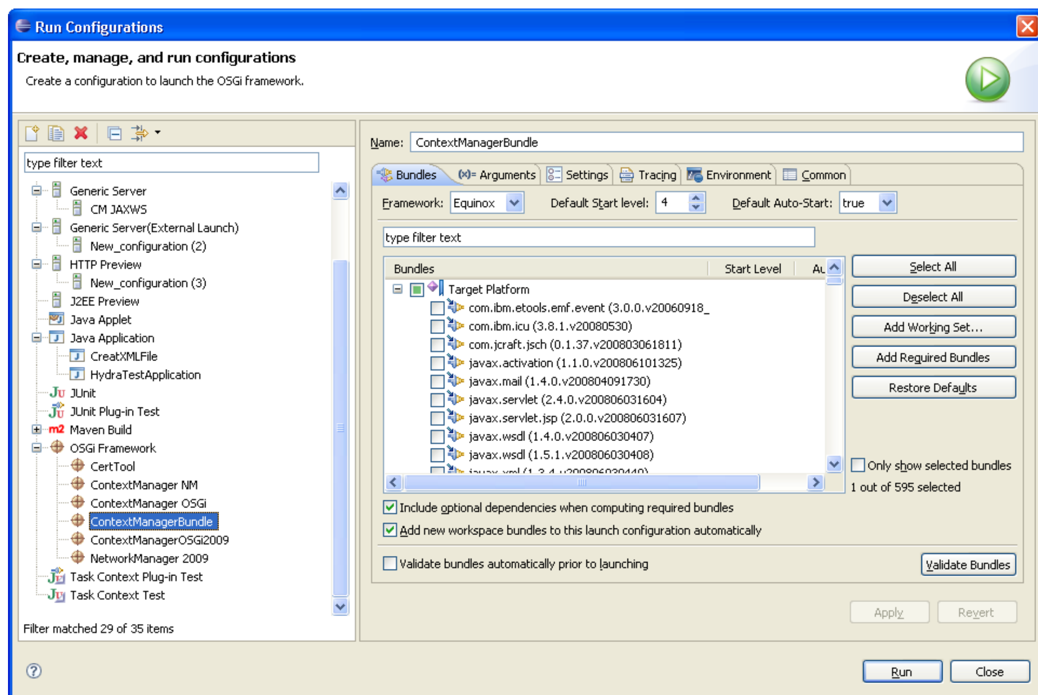

Figure 3: Run Configuration (Hydra Bundles)

These bundles provide the basic functionality of the Network Manager with the CryptoManager. Other bundles must be added as required.

4.3 Crypto Manager Setup

The Crypto Manager (see chapters 5.2.6 and 7.3 for details) requires some initial modification of the default Java distribution, in order to provide the functionalities it requires.

How to register the global crypto provider:

1. In order to use the bouncycastle keystore and cryptographic keys longer than 128bit, the "JCE unlimited strength policy files" needs to be updated. Copy *local_policy.jar* and *US_export_policy.jar* to **$JAVA_HOME/jre/lib/security** (overwriting existing files).

(The following step should be optional. You should try it if you get a "KeystoreException: no match")

2. In order to make the bouncycastle crypto provider available for the whole OSGi framework, it needs to be installed as a global java security provider

    a. Copy *lib/bcprov-jdk14-138.jar* to **$JAVA_HOME/jre/ext**

b. In **$JAVA_HOME/jre/lib/security/java.security**, add bouncycastle to the available crypto providers[1]:

security.provider.5=org.bouncycastle.jce.provider.BouncyCastleProvider

4.4 VM Arguments

-Declipse.ignoreApp=true -Dosgi.noShutdown=true -Dorg.osgi.service.http.port=8082

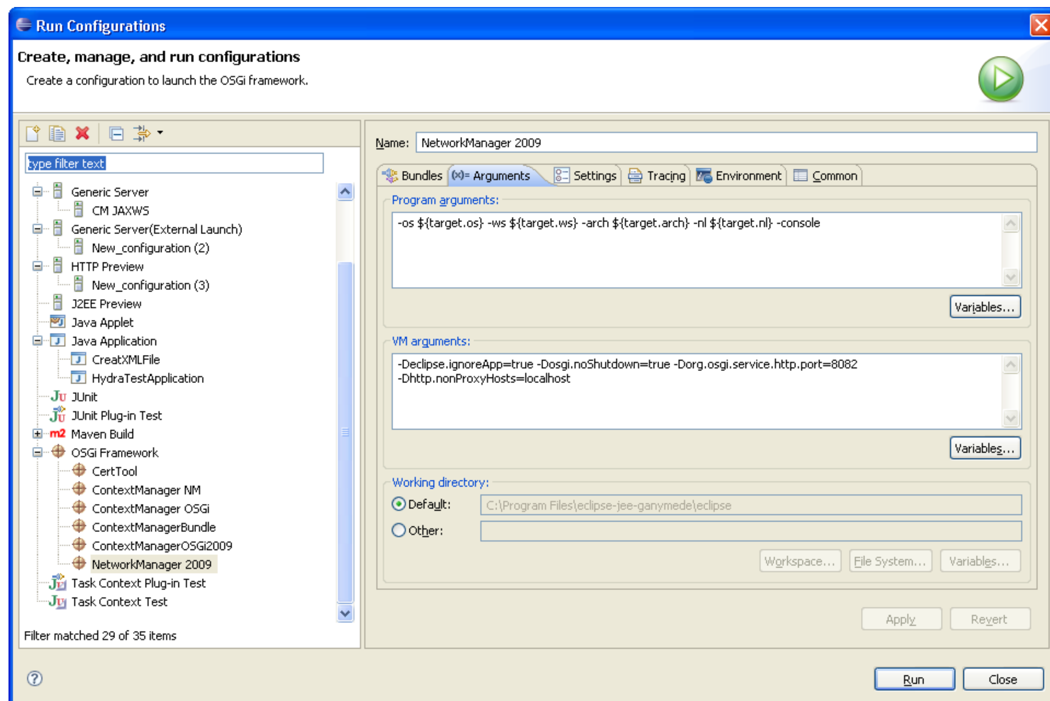

Figure 4: Eclipse Run Configuration (Arguments)

4.5 Running the framework

When the Hydra framework is started and '*ss*' type in to see the list of installed plugins. It should look like this (id numbers may be different, but the order of the bundles in which they start is vital):

```
Framework is launched.

id      State       Bundle
0       ACTIVE      org.eclipse.osgi_3.5.1.R35x_v20090827
2       ACTIVE      CryptoManager_1.1.0
3       ACTIVE      org.apache.log4j_1.2.13.v200903072027
4       ACTIVE      javax.servlet_2.5.0.v200806031605
5       ACTIVE      org.eclipse.equinox.util_1.0.100.v20090520-1800
6       ACTIVE      org.apache.xml.serializer_2.7.1.v200902170519
7       ACTIVE      org.eclipse.equinox.http.jetty_2.0.0.v20090520-1800
8       ACTIVE      org.mortbay.jetty.server_6.1.15.v200905151201
9       ACTIVE      HydraMiddlewareAPI_1.0.0.qualifier
11      ACTIVE      org.eclipse.equinox.ds_1.1.1.R35x_v20090806
```

[1] Don't set bouncycastle as the first provider. This is a known bug and won't work.

```
12     ACTIVE      HydraManagerConfigurator_1.0.0.qualifier
14     ACTIVE      org.eclipse.osgi.services_3.2.0.v20090520-1800
15     ACTIVE      org.eclipse.equinox.cm_1.0.100.v20090520-1800
16     ACTIVE      HydraWSProvider_1.0.0.qualifier
17     ACTIVE      javax.xml_1.3.4.v200902170245
18     ACTIVE      org.mortbay.jetty.util_6.1.15.v200905182336
19     ACTIVE      org.apache.commons.lang_2.3.0.v200803061910
21     ACTIVE      org.apache.xalan_2.7.1.v200905122109
22     ACTIVE      Network_Manager_Bundle_1.7.0.qualifier
23     ACTIVE      org.apache.commons.logging_1.0.4.v200904062259
24     ACTIVE      org.eclipse.equinox.http.servlet_1.0.200.v20090520-1800
25     ACTIVE      org.apache.commons.httpclient_3.1.0.v20080605-1935
26     ACTIVE      HydraMiddlewareClients_1.0.0.qualifier
27     ACTIVE      org.apache.commons.codec_1.3.0.v20080530-1600
```

Useful URLs

Goto: http://localhost:8082/NetworkManagerStatus

It should show something like this

Figure 5: Network Manager Status page in browser

Goto: http://localhost:8082/HydraStatus

It should show something like this

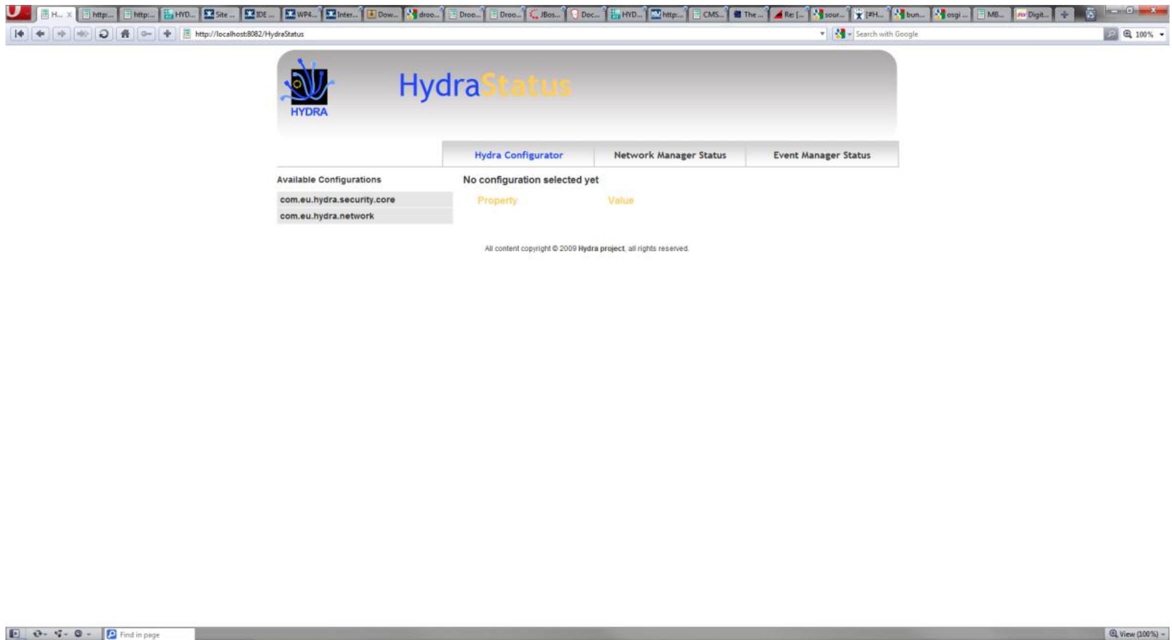

Figure 6: Hydra Status page

Goto: http://localhost:8082/axis/services

It should show something like this

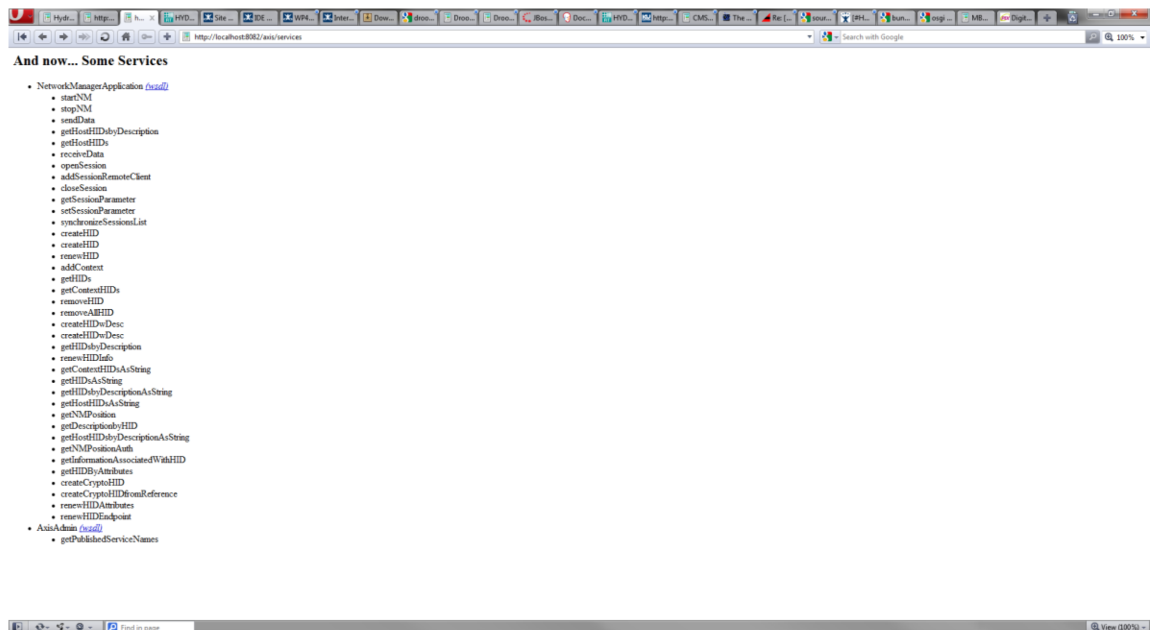

Figure 7: List of services (in browser)

5. Software Development Kit

This chapter provides an introduction to the Software Development Kit (SDK) in Hydra, detailing the software interfaces (Web Services etc) of each Hydra component / manager / tool, and tutorials on how to use them.

5.1 Hydra Commons

The Hydra Commons set of bundles provides the main point of interaction between the developer and the SDK. The commons bundles include:

- Middleware API
- Clients
- Configurator

These bundles make life a lot easier for the developer to use the managers and components of the Hydra Middleware, as well as for the creation of applications.

5.1.1 Hydra Middleware API

The Hydra Middleware API bundle contains the Hydra API, that is, all external interfaces of the Hydra managers and the types used in them. In that way, there is one common bundle containing all relevant Hydra interfaces, separating them from their implementation, which is necessary for a well-structured integrated middleware. This also includes the classes for the various types that act as parameters for calls to middleware components, as well as a set of utilities to aid with usage.

Use of the Middleware API for particular components is discussed in the sections relevant to each component. Typically, this also involves the Hydra Middleware Clients bundle, as described in the following section.

5.1.2 Hydra Middleware Clients

The Hydra Middleware Clients bundle contains all the Web Service clients for calling the various managers and components of the Hydra Middleware. These clients include the generated AXIS files for the creation of Web Service Clients, providing the services as defined in the Hydra Middleware API. The middleware clients are located in bundles named as follows:

Using the Event Manager as an example, the developer can generate the Event Manager client in one of two ways. Firstly, by using the generated *Locator* class for each client, as shown below:

```
EventManagerPortServiceLocator locator =
                                    new EventManagerPortServiceLocator();
locator.setEventManagerPortEndpointAddress(endpoint);
EventManagerPort em = locator.getEventManagerPort();
em.subscribe("ExampleTopic", "0.0.0.235235154145");
```

Here, the locator is configured with an endpoint address. This is the address of the local SOAP Tunnel (exposed by the Network Manager), specifying the *from* and *to* HIDs, as well as the *sessionID*. An example endpoint is given below, with no *sessionID* (0).

```
http://localhost:8082/SOAPTunneling/0.0.0.341243145454252/0.0.0.4124344658
75/0/
```

The second method is to use the RemoteWSClientProvider OSGi service that allows retrieval of the relevant manager objects without the need to create specific Web Service Locator objects. This service offers a method, called getRemoteWSClient, which interface is:

```
public Object getRemoteWSClient(String className,
                        String endpoint, boolean coreSecurityConfig);
```

When calling the method, the interface class name of the Web Service, the endpoint to this Web Service and a Boolean value indicating whether we want to call the service with Hydra security or not must be provided.

The next lines show an example of calling a method of the Network Manager Web Service. The developer first obtains a RemoteWSClientProvider element, and via this Web Service Client object finally makes the call to the desired method of the Network Manager (in this case getHIDs).

```
RemoteWSClientProvider service = (RemoteWSClientProvider)
context.getService(context.getServiceReference
(RemoteWSClientProvider.class.getName()));
NetworkManagerApplication nm = (NetworkManagerApplication)
service.getRemoteWSClient(NetworkManagerApplication.
class.getName(),
endpoint, true);
Vector v = nm.getHIDs();
```

5.1.3 Hydra Configurator

In order to achieve high level of integration between the set of managers that conforms the Hydra Middleware, a common configuration system for all Hydra managers and applications has been implemented.

This common configuration system is based on the use of the configuration admin OSGi service. This service provides a way to dynamically update the configurations, avoiding having to restart the managers in order to update them. It also provides persistency for the configurations.

Thus, an OSGi bundle which provides a common interface for the configuration of all Hydra managers has been implemented. The bundle is called the Hydra Manager Configurator. Adapting the configuration of a particular manager is achieved using the Configurator class provided by the Hydra API. The Configurator class is a class that implements the ManagedService interface, so that it can receive configurations from the configuration admin OSGi service. The Configurator class provides the methods and attributes for managing the configuration of the manager which instantiates it, and a way for communicating with the configuration admin service in order to apply into this service all changes introduced by the user at the Configurator class, registering itself as a managed service.

The Hydra Manager Configurator bundle provides a set of interfaces that makes it possible to modify the configuration of the different Hydra managers previously adapted to the new common configuration system. This bundle provides three interfaces for configuring Hydra managers:

- A web application called Hydra Status
- A Web Service deployed by the Hydra Manager Configurator
- An OSGi console command, currently working on Equinox

The Hydra Status page (Figure 8) is a web application that provides a web interface for configuring the different local Hydra managers adapted to the new common configuration system, based on the configuration admin OSGi service. It also provides all the information provided by the well-known Network Manager Status page and Event Manager Status pages.

Regarding the Network Manager information included, the Hydra Status page provides information about HIDs, hosts where they are deployed, descriptions and endpoints of all devices detected by

the local Network Manager, differentiating between local and remote HIDs (local and remote Network Manager installations).

Regarding the Event Manager, the Hydra Status page provides information about the topics, the endpoints and the dates of subscription of all the Hydra events the local Event Manager is subscribed to.

Regarding the configuration of the managers, the Hydra Status page provides a graphical interface for configuring all Hydra managers adapted to the new common configuration system in a dynamic way. The available sets of configuration options are loaded dynamically if the manager which uses them is running, identifying themselves by their configuration PID. Clicking over a configuration PID, all its options and their values will be loaded, making it possible to modify them and update the modifications made by clicking the Update Configurations button.

Once you update the configuration, the new configuration will be working, without having to restart the managers. However, if you put –clean parameter inside the program arguments of your OSGi configuration all updated configurations will be reset to their initial state. The URL of the Hydra Status page (a screenshot of the service can be seen in Figure 8) is http://localhost:8082/HydraStatus, (given that the web server of the Hydra installation is running in the 8082 port, which is the default port of the Hydra configuration). This functionality is also replicated inside the IDE, as discussed in the relevant section(s).

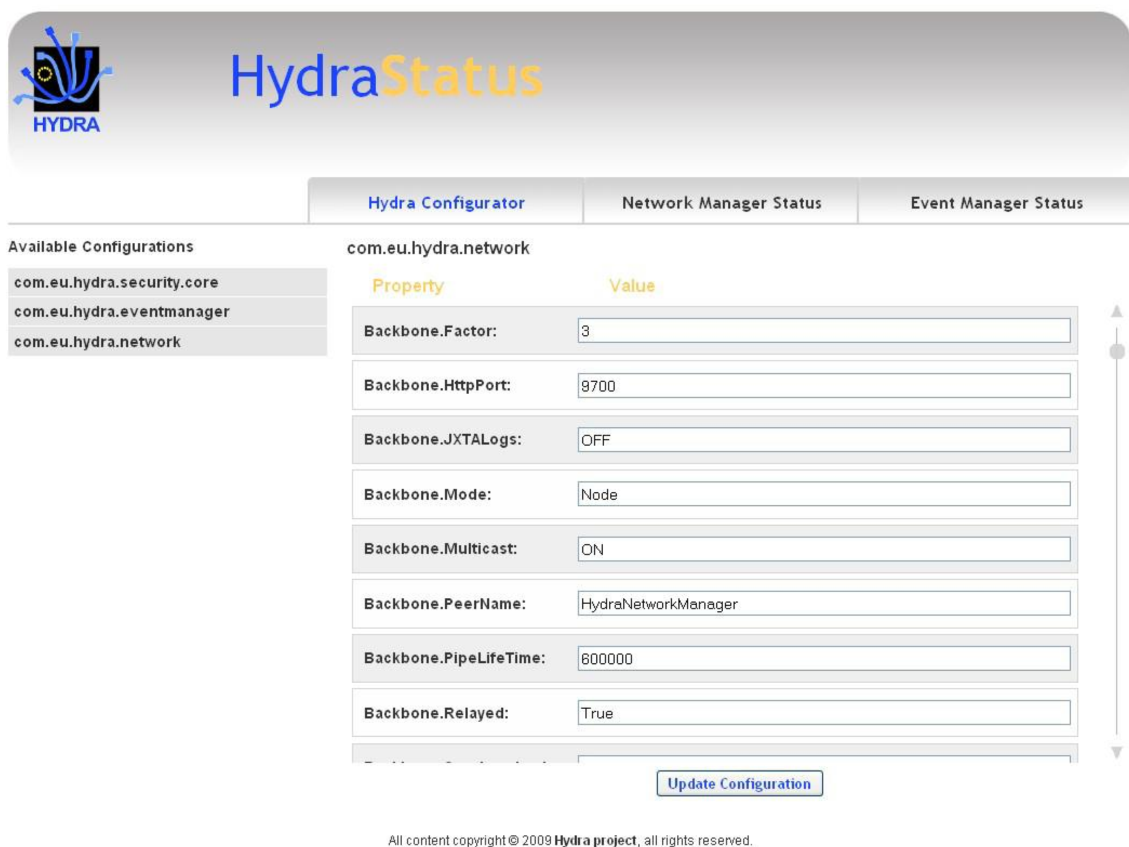

Figure 8: Hydra Status page screenshot

Another configuration tool provided is the Web Service, which is deployed by the Hydra Manager Configurator bundle, and that provides the following methods:

- getAvailableConfigurations(): list the available set of configurations.
- deleteConfiguration(String configuration_pid): delete a concrete configuration from the common configuration system.

- listConfiguration(String configuration_pid): list the options provided by a concrete configuration and their current values.
- setConfiguration(String configuration_pid, String option_key, String value): set the value of a concrete option for a concrete configuration.

Finally, and regarding the configuration tools using Equinox OSGi console, the configure command is used, which provides similar options as the ones provided by the Web Service above. These options are the:

- configure –l: list the available set of configurations.
- configure –d <configuration_pid>: delete a concrete configuration from the common configuration system.
- configure <configuration_pid>: list the options provided by a concrete configuration and their current values, e.g. configuration com.eu.hydra.network will print current configuration of the Network Manager.
- configure <configuration_pid> <option_key> <option_value>: set the value of a concrete option for a concrete configuration.

5.2 Network Manager

The network model complements the runtime platform model regarding the details of the network. In Hydra the underlying network is complex and therefore, it needs to be described in a separate (but related) network model. The purpose of the network model is to define what types of network connections will be supported and if there are constraints that have to be adhered to during implementation and network design.

5.2.1 Hydra Definition of Device To Device Communication

The Network Manager is the incoming and outgoing point of information in the middleware. Therefore, the main purpose of Device to Device communication will be managing the communication between Network Managers. This means that only Hydra-enabled-devices will be involved in this kind of communication.

Devices inside Hydra need to communicate in order to exchange information. Each device offers different resources inside the Hydra network mechanisms which need to be implemented so making possible the discovery of new resources in Hydra-enabled devices inside the network. Moreover, in order to consume these resources, Hydra devices need the means to establish communication between each other. The following sections will present those aspects.

5.2.1.1 Addressing

From the middleware point of view, an addressing method based on Hydra Identifiers (HID) has been defined for Hydra, instead of the usual IP-based one. The Identity Manager is responsible for the management of these HIDs. Its main functionality is providing a unique context-dependant identifier for every device (physical or semantic), resource or service, called HID. It is also responsible for the maintenance of the idTable, a data structure dedicated to store the matching between logical and physical identifiers.

However, this addressing method is useless if there is not a way to propagate this information to other Hydra-enabled devices involved in the Hydra Network. The Backbone Manager is responsible for spreading this information between the different Hydra-enabled devices in the network. Thus, every Identity Manager belonging to the Hydra Network keeps an idTable internally and an updated list of every HID in the network. This process is known as Network Manager Discovery.
The Hydra middleware will be running in dynamical environments, where new resources are susceptible to constantly appear or disappear. In order to detect new resources inside the Hydra network, we need a discovery mechanism.

Inside the Hydra network, devices and resources are identified through a Hydra ID (HID), which varies depending on the context. In order to contact them, one Hydra-enabled device needs to contact the Network Manager of the Hydra-enabled device they belong to. The discovery of Network Managers will be done through use of Device to Device communication.

Through Device To Device communication, we aim to propose an innovative way to discover Network Managers (and thus, Hydra-enabled devices) and also to know more about their features and the services they provided, in a scalable Wide Area Network. This means that the scope of the Hydra network will not be restricted to a Local Area Network.

5.2.1.2 Communication

As mentioned before, the Network Manager is the incoming and outgoing point for information in the middleware. The Hydra network is an "all-IP" network. This means that only devices with IP communication capabilities will be able to communicate directly (through device to device communication) inside the Hydra network.

Moreover, the device to device communication will be restricted to the Hydra-enabled devices that are able to host the Hydra middleware, which in Hydra terms means that this communication will be "inside" Hydra. Thus, the device to device communication can be defined as the data exchange between devices "inside" the Hydra network, which are Hydra enabled and have IP communication capabilities.

5.2.2 The Peer-to-Peer Network Architecture

There exist multiple objectives regarding device to device communication. First, the Hydra middleware needs to offer an efficient way to share resources among the Hydra Network, in a scalable, distributed and efficient way. The Hydra middleware also needs to prevent system failures when a node is not available. And finally, the Hydra Network needs to allow ubiquitous access to the network.

All of these reasons have led us towards a Peer-to-Peer architecture. Several Peer-to-Peer models have been analysed and according to the requirements identified for device to device communication, JXTA P2P communication protocols have been selected as the most suitable mechanism to carry on the communications "inside" Hydra. That is, the communication between Network Managers.

The reasons that have led us to select JXTA are:

- Interoperability: Enables communication between peers independently of network addressing and physical protocols.
- Platform independence: JXTA does not depend on the programming language, network transport protocols and deployment platforms, giving freedom of choice. Java SE and Java ME implementations have been selected for Hydra.
- Ubiquity: JXTA is designed to be deployed on any device, not just PCs.
- Security: for security means regarding authentication, authorisation, and integrity can be implemented based on JXTA. Attacks on the level of the protocol cannot be addressed as that would require changing the JXTA protocol.
- Community support: JXTA is supported by a wide community of developers and the different specifications are fully documented.
- Wide range of services: Most of the P2P models studied have been designed exclusively for providing file sharing services. Instead, in JXTA, thanks to its abstract architecture based on six protocols, it is possible and feasible to create a wide range of interoperable services and applications.

5.2.3 Purpose

The Network Manager is the bottom layer of the Hydra middleware deployed in Hydra Gateways and in Hydra-enabled devices. It is the entry and exit point of information of the Hydra middleware. There is only one Network Manager per device where the middleware is deployed.

The Network Manager provides a Web Service interface (which is the main interface of the Network Manager), which is the information entry point for the middleware. Data transferred between Hydra-enabled devices and gateways should always pass through the Network Manager.

5.2.4 Main Functionalities

The Network Manager is responsible for managing the communication between Hydra-enabled devices. In order to do this, the Network Manager:

- Creates and overlay P2P network, where all the Hydra-enabled devices appear directly interconnected, no matter if they are behind a NAT (Network Address Translator) or Firewall.
- Provides indirection architecture for addressing Web Services hosted by Hydra devices using the HID addressing mechanism. Each service is identified in Hydra through an HID, which is a global and unique identifier. The Network Manager provides interfaces for other managers, applications and Hydra devices for HID creation, modification and deletion. It also offers the possibility to select the transport protocol for the service invocation between TCP, UDP and Bluetooth.
- Provides a transport mechanism over the overlay P2P network for invoking Web Services hosted by Hydra devices (SOAP Tunnelling) using the HID addressing mechanism. The SOAP messages addressed to an HID are routed by the Network Manager through the overlay network to the Network Manager hosting the service. Therefore, using the SOAP Tunnelling and the Network Manager any device or application is able to transparently publish and consume services anywhere, anytime, breaking the network interconnectivity barriers and independently of the service endpoint location.
- Provides a transport mechanism over the overlay P2P network for multimedia content exchange between UPnP AV or DLNA devices.
- Provides session management mechanisms between HIDs during service invocations.
- Provides time reference synchronization between different Network Managers.
- Provides a status page for developers, which the developer can use for monitoring dynamic information about the Hydra Network and the HIDs available.

Each Hydra-enabled device will run one and only one Network Manager. The Network Manager maintains two complex data structures: the Hydra ID (HID) and the Session. The following sections provide an overview on these two data structures.

5.2.5 Hydra Web Service Provider

First of all, in order to make the deployment of Web Services in the Hydra middleware easier, a new OSGi bundle has been created to take the place of the obsolete Axis bundle that the managers were using since the beginning of the project. This bundle is the Hydra WS Provider bundle. The main goal of this component is to provide automatic deployment of Web Services and independence for Hydra managers from Axis.

Now it is possible to deploy Web Services, including the Hydra manager ones, in an automatic way, without the use of a deployer class or WSDD files.

The Hydra WS Provider bundle is still based on the Axis bundle, but it has been adapted to the Hydra middleware, providing transparent interfaces to the developers supporting all the characteristics that the Hydra middleware needs.

The Hydra WS Provider bundle is composed of three packages, as seen in the Table 1:

Package	Definition
com.eu.hydra.security.axis	Provides Core Hydra security to the bundle.
com.eu.hydra.wsprovider.impl	The main package, deals with the detection of OSGi services and their deployment as Web Services.
com.eu.hydra.wsprovider.servlet	Deploys a servlet which represents the Axis administration servlet.

Table 1: Hydra WS Provider package structure

The main class is the Activator class, which can be found under the *com.eu.hydra.wsprovider.impl* package. It deals with the detection of OSGi services and their deployment as Web Services through a *ServiceTracker* object.

The services to be published as Web Services should have been deployed as an OSGi service. It is also recommended but not mandatory to use OSGi Declarative Services. When a service is to be published as a Web Service using the Hydra WS Provider, a set of properties need to be defined:

- *SOAP.service.name*: mandatory property, it defines the name of the service to be deployed. The Hydra WS Provider deploys each service using this defined property. Once this property is set, the service will be deployed with this name at *http://localhost:8082/axis/services*.
- *SOAP.service.methods*: optional property, it provides a list of the names of the methods to be implemented. When defined, the Hydra WS Provider deploys only the methods indicated. Otherwise, the Hydra WS Provider deploys all the methods of the Web Service, i.e. all methods that have an access level of 'public'.
- *Hydra.security.config*: optional property, it defines whether the Web Service is to be deployed with or without security. A Boolean value defines this property. All services will be deployed with security by default.

In order to register and deploy a Web Service in the Hydra middleware, the developer must register the service in the framework with the *SOAP.service.name* property indicating the name of the service. Programmatically, and in the case of non declarative services, a service is registered as follows:

```
Hashtable props = new Hashtable();
props.put("SOAP.service.name", "EventManagerPort");
context.registerService(EventManager.class.getName(),this, props);
```

By using OSGi declarative services, the services are already registered via the framework, but the properties have to be set in the *OSGI-INF/component.xml* file (at least the mandatory *SOAP.service.name* property), as shown in the Figure 9.

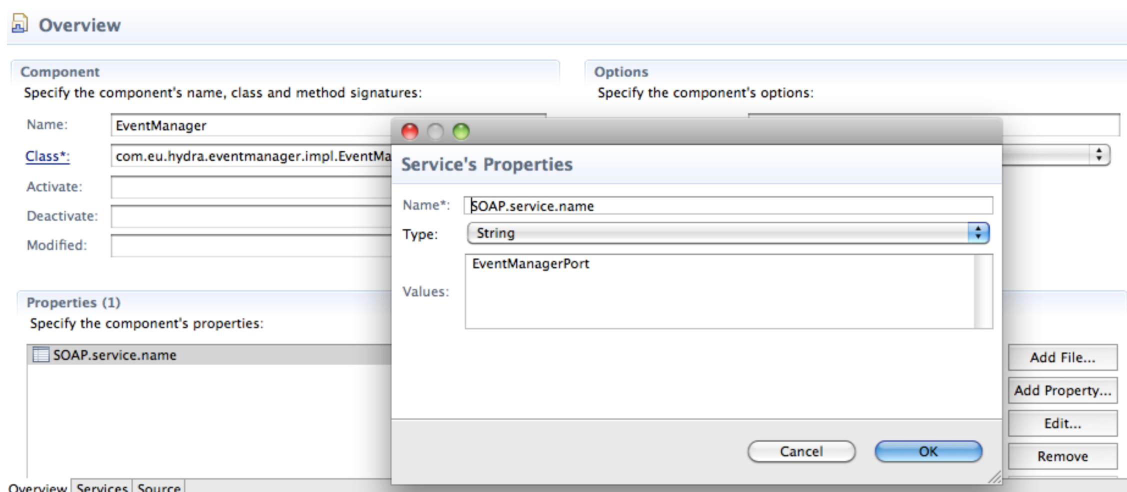

Figure 9: Introduction of a property at the XML file of a declarative service

5.2.6 Crypto HIDs

From the Lessons Learned of the third iteration, we realized that the service addressing mechanisms implemented in Hydra, the HIDs, lack of high security features. The HIDs are identifiers that allow developers and applications to identify each entity evolving in a Hydra network. It was designed to identify each service in a given situation (context) but also to dismiss the real identity of the device offering the service.

The main problem with current implementation of HIDs is that the information related to their description is being exchanged over the network without any encryption between the Network Managers. Thus an attacker of the Hydra middleware would be able to identify the identity of HIDs and the service provided by the owner of the HID by just sniffing the network traffic. Another problem identified with HIDs, is that the description field associated with them is not enough to unambiguously identify a service, as it is just a String with no fixed format. These problems are not very important for building applications that do not have high security requirements, but when moving to domains that require these high levels of security, such as e-Health, the problems become important.

In order to solve these issues, we have extended the HID concept incorporating new security features like certificate linking, HID description through attributes and HID data encryption. These new secure Hydra Identifiers are called Crypto HIDs. The main features incorporated to Crypto HIDs are:

5.2.6.1 Certificate linking

Each HID, when it is created, it is associated with a certificate, generated using the Crypto Manager. This certificate is used to encrypt and decrypt all the information sent to and from this HID. Therefore, before sending any information to an HID, the Network Managers perform a certificate exchange process for encrypting the information that is going to be exchanged. This certificate exchange is performed using the Secure Session Protocol, with which certificates will be distributed using a public key exchange protocol.

5.2.6.2 HID attributes

In order to unambiguously identify a HID in the Hydra Network, we have extended the description of HIDs to attributes. Each HID is created with some attributes, which are securely stored in its certificate. The number of attributes is not fixed, and it is up to the developer to decide which attributes to use. Some examples for attributes would be:

- PID (Persistent Identifier): An identifier for the device providing a service (for example, MAC address of the device)
- SID (Service Identifier): An identifier for the service provided. It could also be a semantic identifier of the service provided.
- UserID (User Identifier): Identifier for the owner of the device providing a service.

These attributes, and any others, are provided during HID generation following the Java Properties class XML schema. An example of attributes for an HID would be:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM
"http://java.sun.com/dtd/properties.dtd">
<properties>
<entry key="PID">03-43-F3-23-24</entry>
<entry key="SID">ThermometerService</entry>
<entry key="UserID">Peter</entry>
</properties>
```

The attributes are not exchanged between Network Managers during the HID exchange process, that is, Network Managers have all the HIDs in the Hydra Network but do not have the information at to what each HID stands for. In order to provide developers and applications the means to know these attributes, we have implemented two mechanisms to retrieve the attributes for a specific HID and to query the network searching for an HID matching some attributes. These two mechanisms have been designed taking security into account:

- Retrieving attributes for an HID: Using this mechanism, developers and applications are able to retrieve the attributes of a specific HID. As mentioned above, attributes for a HID are stored securely into the certificate linked to it. Therefore, in order to retrieve the attributes for a HID, the Network Manager starts the Secure Domain Protocol, exchanging the certificates of the interested parties. Therefore, nobody without a valid Hydra certificate is able to retrieve the attributes for a HID.

- Querying the Hydra network of a HID matching some attributes: This is the situation when an application wants to address a specific HID, with some fixed attributes, but without knowing beforehand which is the HID assigned to it. Imagine an application that wants to retrieve the temperature from a specific thermometer. It first needs to know the HID of that thermometer in order to be able to invoke its service.

The process is simple: a query is generated and sent to all the Network Managers in the network using a multicast channel (step 1 in Figure 10). In the query, the requester has to provide its credentials, this is, its HID and attributes. Each Network Manager receives the query and searches in the local idTable (step 2 in Figure 10) (the table where all the HIDs are stored). If a Network Manager finds a HID that matches the query (step 3 in Figure 10), before answering to the sender, checks with the Policy Manager if there is any policy applied for that HID and provides the sender information (step 4 in Figure 10). The Policy Manager answers the Network Manager as to whether it is allowed or not to send that information to the requester (step 5 in Figure 10). Whether or not it is allowed, a query response containing the HID is sent to the sender over a unicast channel (step 6 in Figure 10). If it is denied, no information is sent to the sender. Therefore, in every step of this process, security is ensured. This process is illustrated in the Figure below.

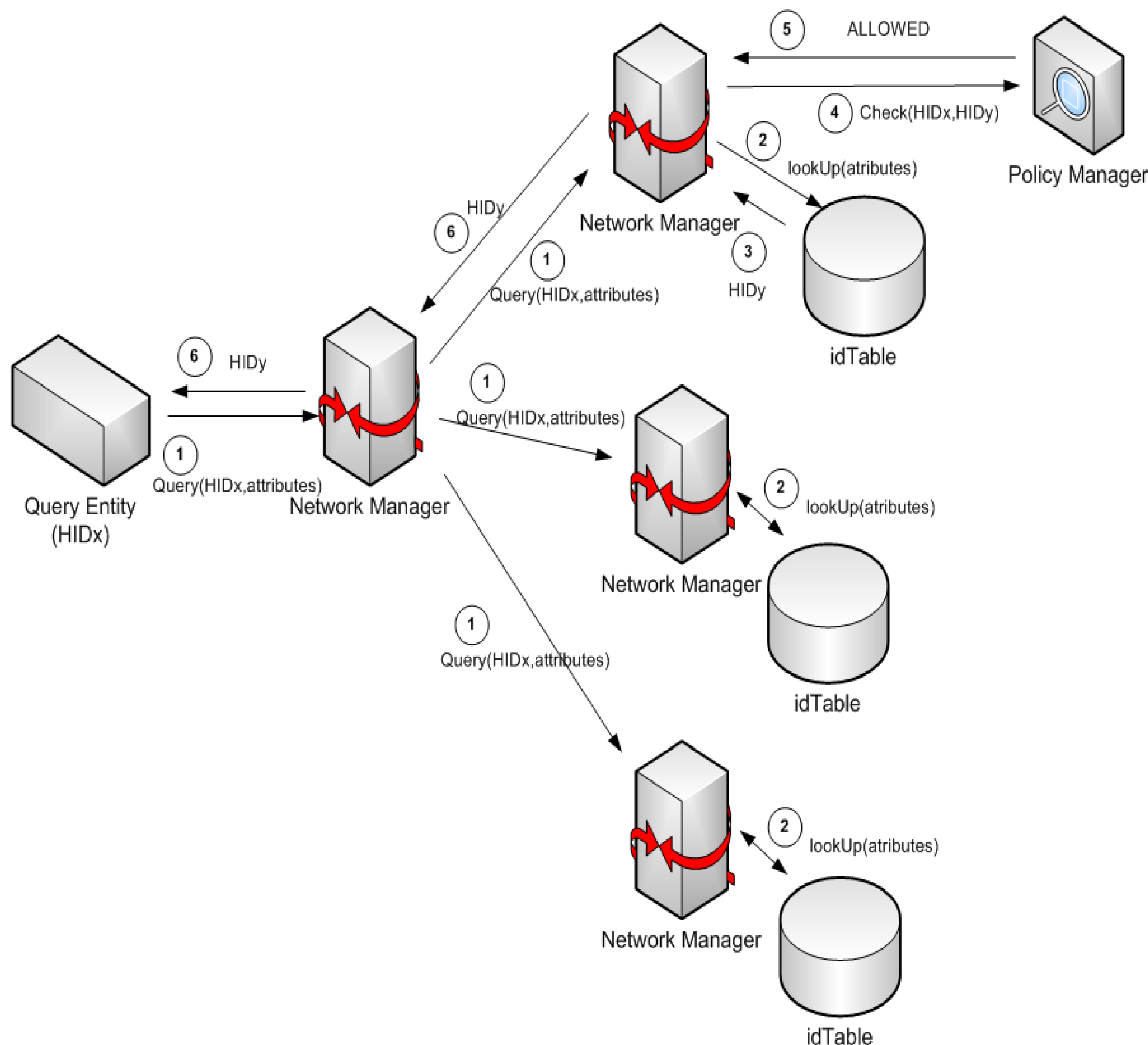

Figure 10: Querying the Hydra network for a HID matching some attributes

In order to provide the developers with the tools for using these new mechanisms, four new methods have been added to the Network Manager API. The old methods for creating and interacting with HID related information are still maintained, for backwards compatibility reasons, but its usage is discouraged as they have been deprecated.

createCryptoHID

```
CryptoHIDResult createCryptoHID(java.lang.String xmlAttributes,
                                java.lang.String endpoint)
                                throws java.rmi.RemoteException
```

Operation to create an crypto HID providing the persistent attributes for this HID and the endpoint of the service behind it (for service invocation). The crypto HID is the enhanced version of HIDs, that allow to store persistent information on them (through certificates) and doesn't propagate the information stored on it. In order to exchange the stored information, the Session Domain Protocol is used. It returns a certificate reference that point to the certificate generated. The next time the HID needs to be created, using the same attributes, the certificate reference can be used.

Parameters:

`xmlAttributes` - The attributes (persistent) associated with this HID. This attributes are stored inside the certificate and follow the Java `Properties` xml schema.

`endpoint` - The endpoint of the service (if there is a service behind).

Returns:

A `com.eu.hydra.network.ws.CrypyoHIDResult` containing `String` representation of the HID and the certificate reference (UUID)

Throws:

`java.rmi.RemoteException`

createCryptoHIDfromReference

```
java.lang.String createCryptoHIDfromReference(java.lang.String certRef,
                                              java.lang.String endpoint)
                                              throws java.rmi.RemoteException
```

Operation to create an crypto HID providing a certificate reference (from a previously created cryptoHID) and an endpoint The crypto HID is the enhanced version of HIDs, that allow to store persistent information on them (through certificates) and

Parameters:

`certRef` - The certificate reference from a previously generated cryptoHID.

`endpoint` - The endpoint of the service (if there is a service behind).

Returns:

The `String` representation of the HID.

Throws:

`java.rmi.RemoteException`

5.3 Device Application Catalogue

5.3.1 The DAC browser

The DAC Browser is now also an integral component of SDK as part of the IDE as seen in Figure 11 below:

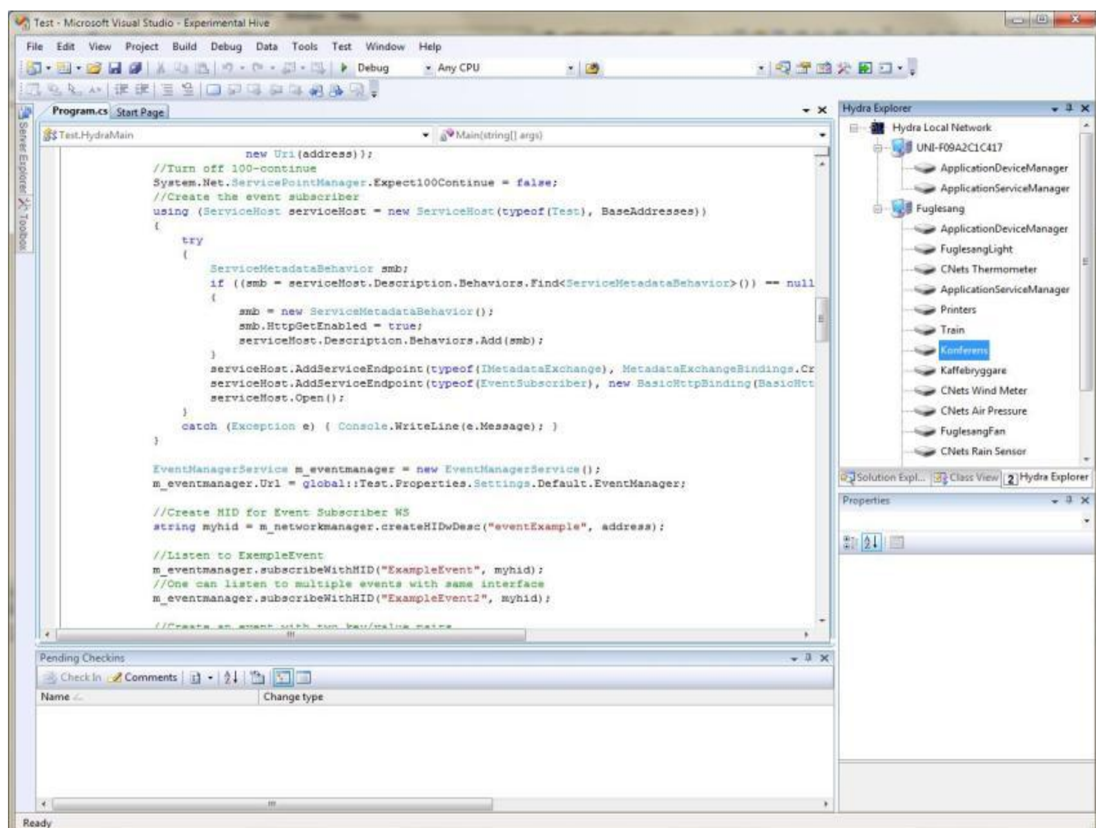

Figure 11: DAC Browser (upper right) in the IDE

It provides the same functions as the stand-alone version and in addition,

- Provide an IDE-view of all devices known to the Hydra Network
- Enables the developer to create proxies by selecting devices

5.3.2 The Graphical Browser

A fundamental part in every Hydra-based application is the Device Application Catalogue (), which is managed by the Application Device Manager, as was explained in previous chapters. 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 metadata they need regarding devices, their services, and their status.

Hydra 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 can be queried by different middleware managers to retrieve a service interface for different devices.

A Hydra browser has been developed to allow a user/developer to graphically browse the Hydra network and inspect properties and services of devices. The browser tool also allows the user to invoke the different services offered by devices.

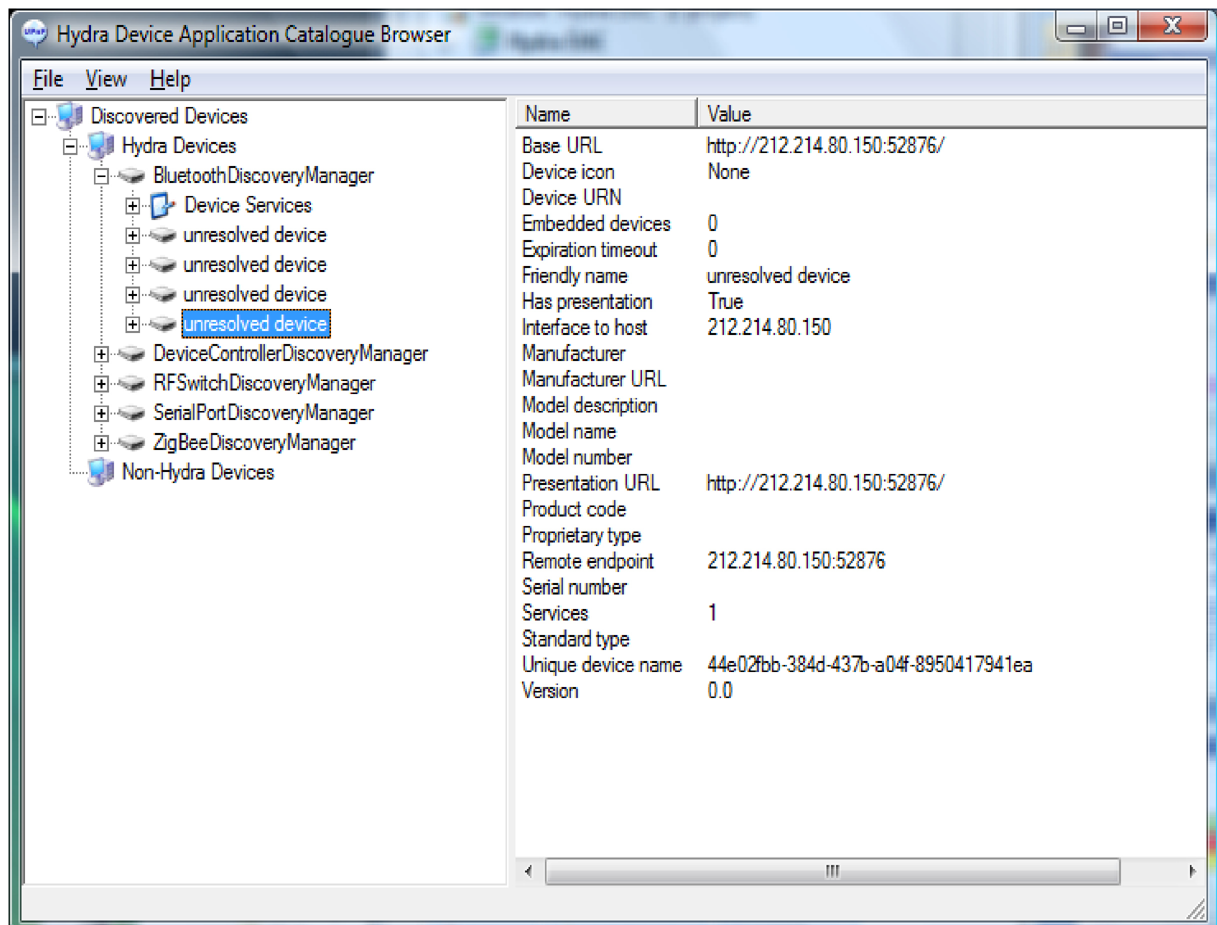

Figure 12: The Hydra Browser

By manually invoking the different services, the actual role the Device Application Catalogue plays in the Hydra middleware can be illustrated. As can be seen in Figure 12 above, 5 different Discovery Managers are available in the network, each of them is dedicated to discover a certain type of physical device (Bluetooth, RF Switches, ZigBee etc).

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

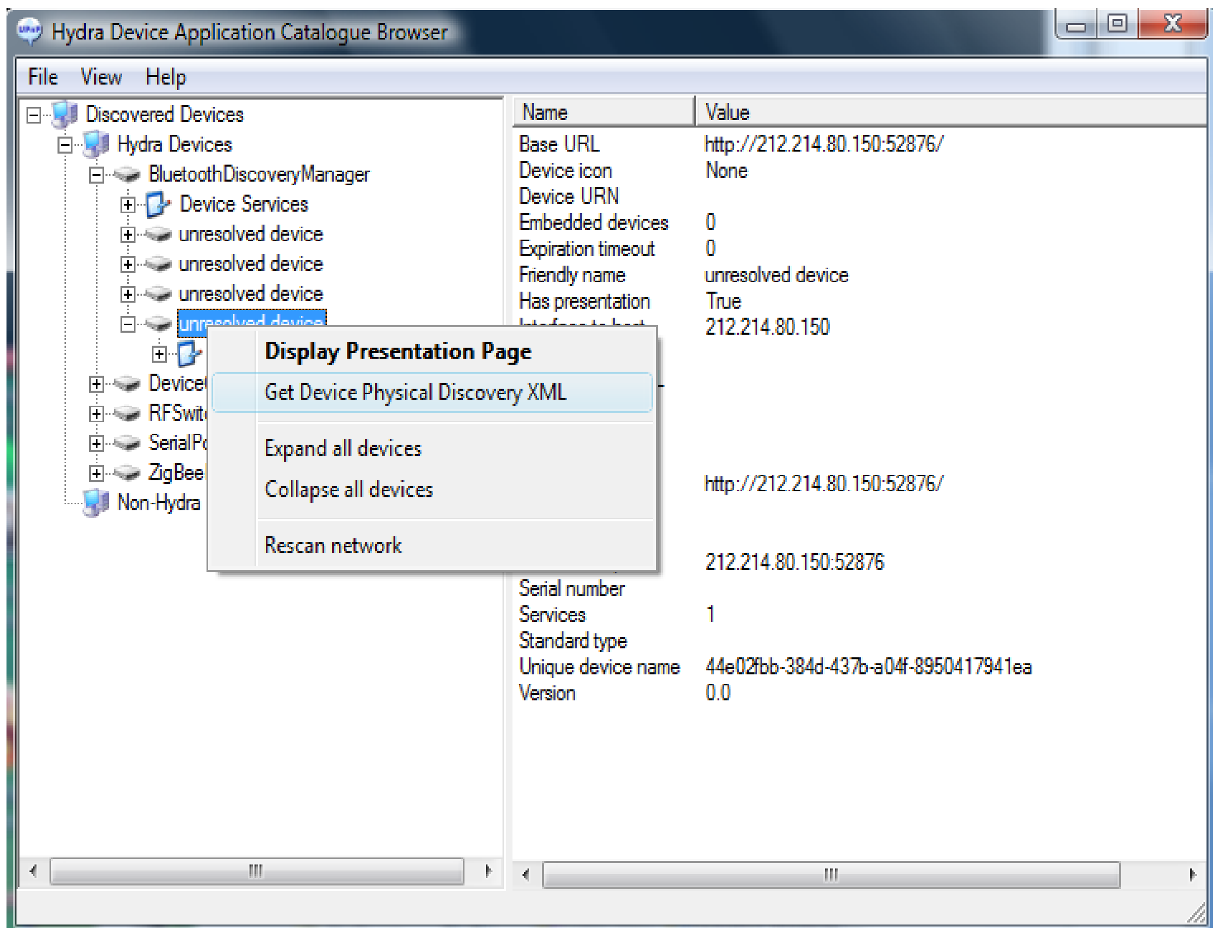

Figure 13: Retrieving discovery information from the physical device

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

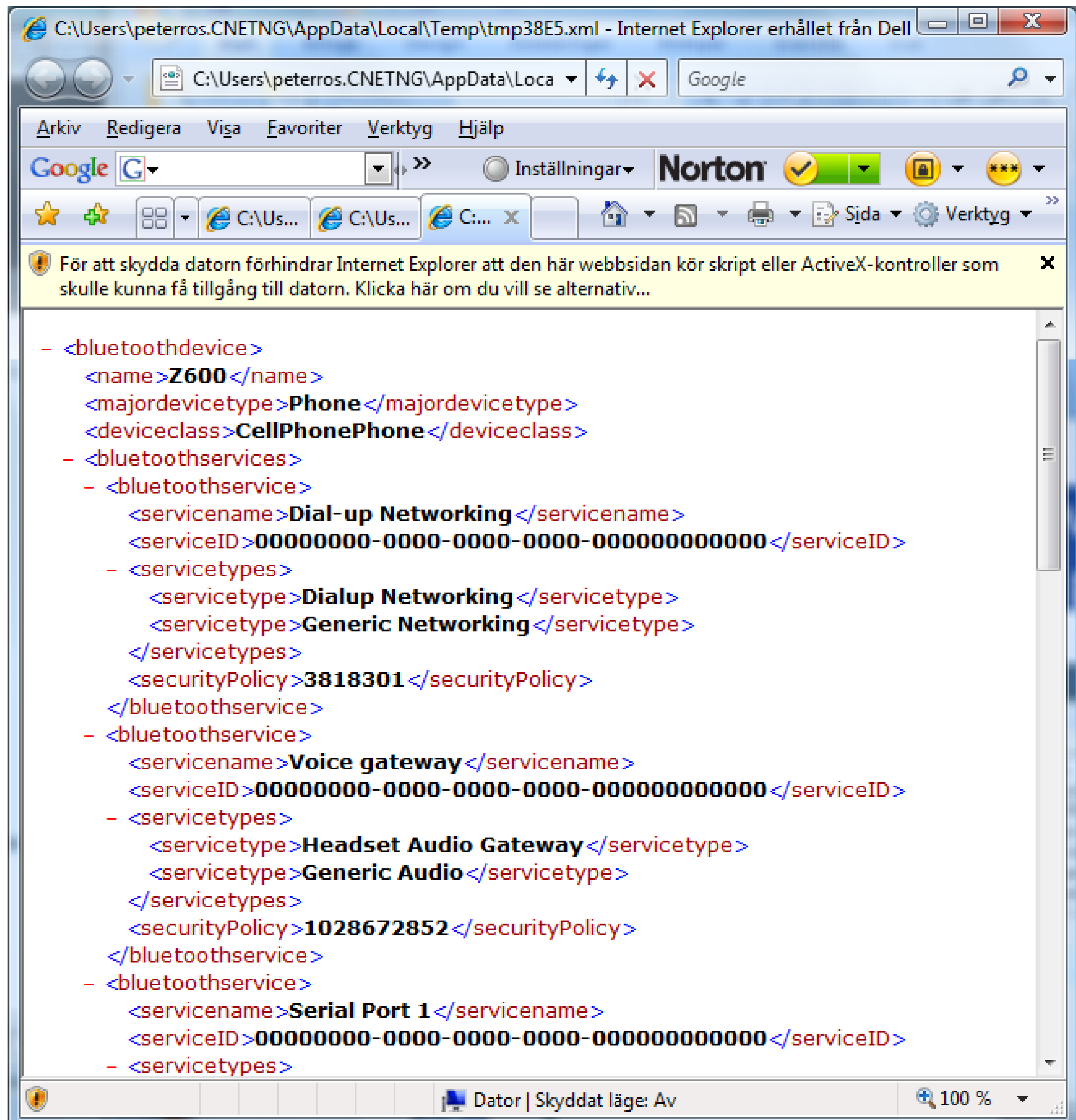

Figure 14: Discovery information from a Bluetooth Device

In Figure 14 we can see that it is a Bluetooth Device that has been discovered, it has the Bluetooth Major DeviceType "Phone" and Minor DeviceType "CellPhonePhone" (Major DeviceType and Minor DeviceType are part of the Bluetooth standard.

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

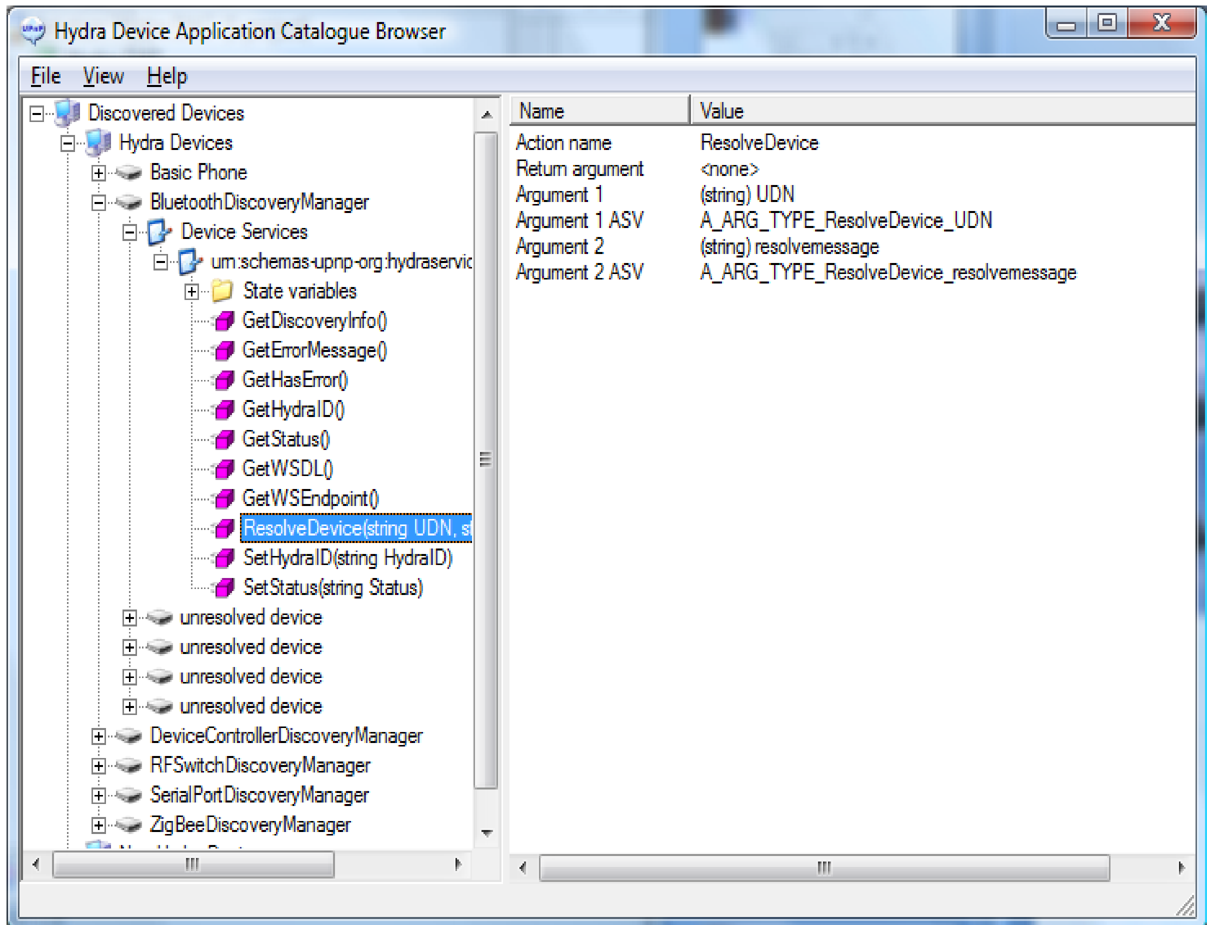

Figure 15: Resolving a physical device into a Hydra Device.

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

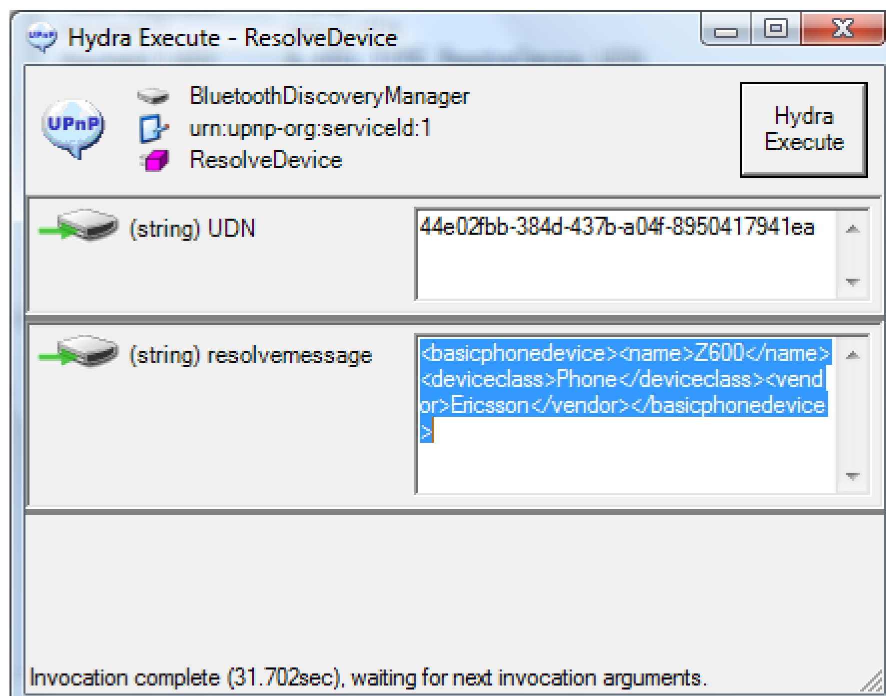

Figure 16: Resolve information is sent as an XML structure to the Discovery Manager

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

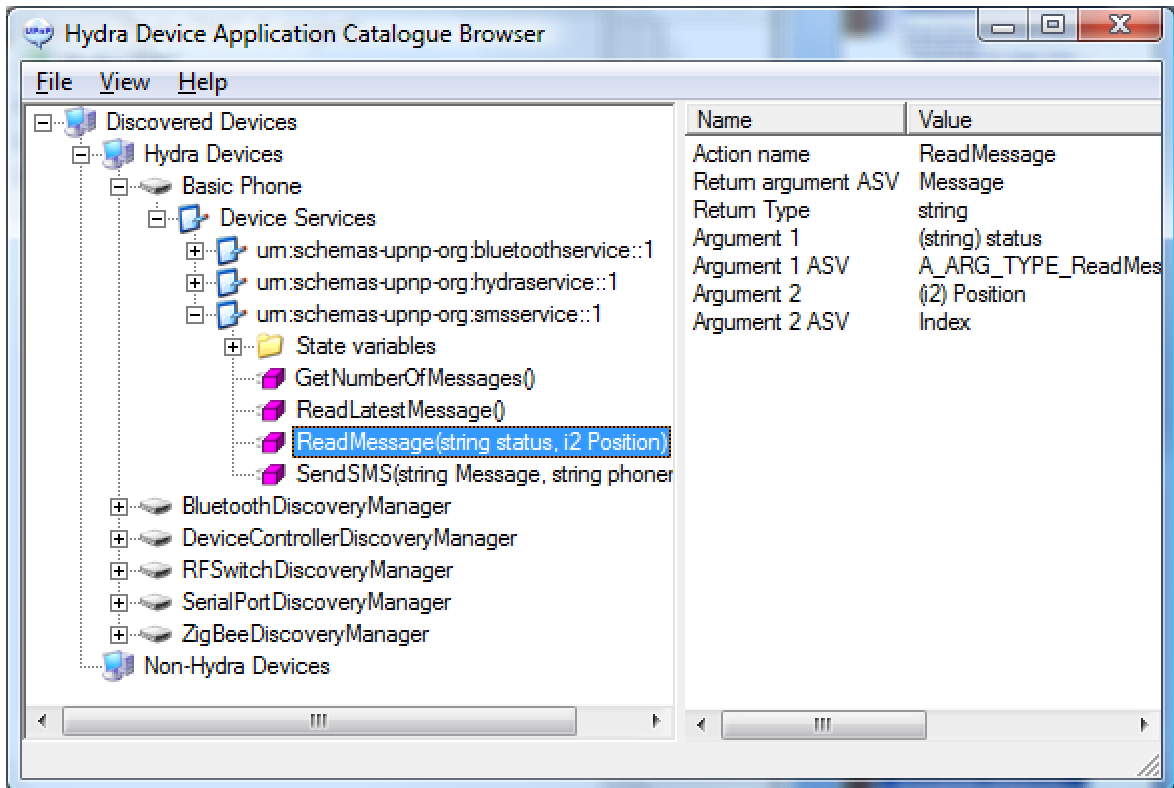

Figure 17: A physical device with unknown functionality has been transformed into Basic Phone Device with services for reading/sending SMS.

These services are now directly invokeable from the Browser, and for instance, an SMS can be sent.

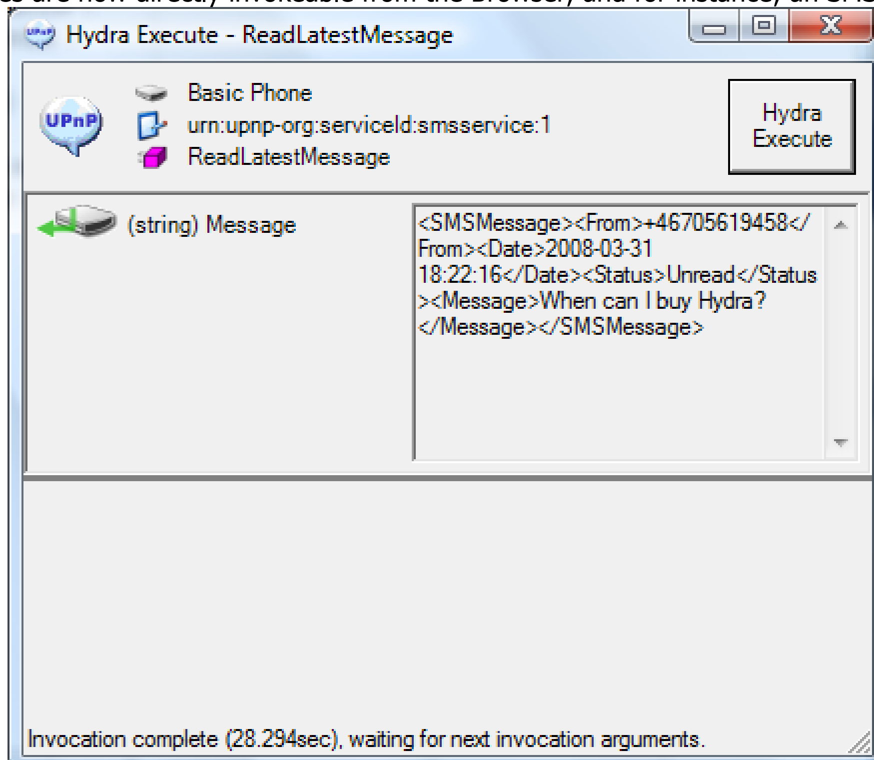

Figure 18: Sending an SMS through the Basic Phone Device

Finally the Browser can be used to retrieve a service description for a web service that allows us to access the device programmatically:

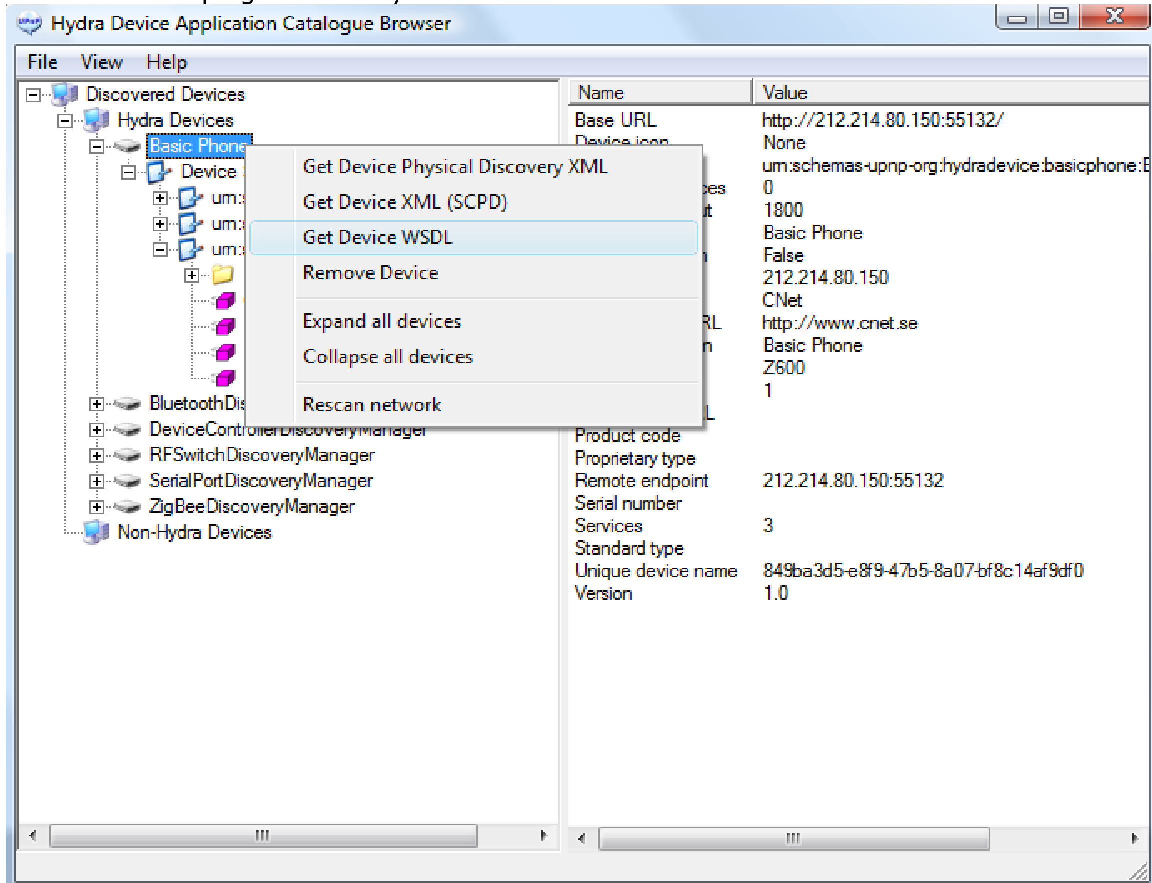

Figure 19: Using the DAC browser to retrieve a WSDL description for the device.

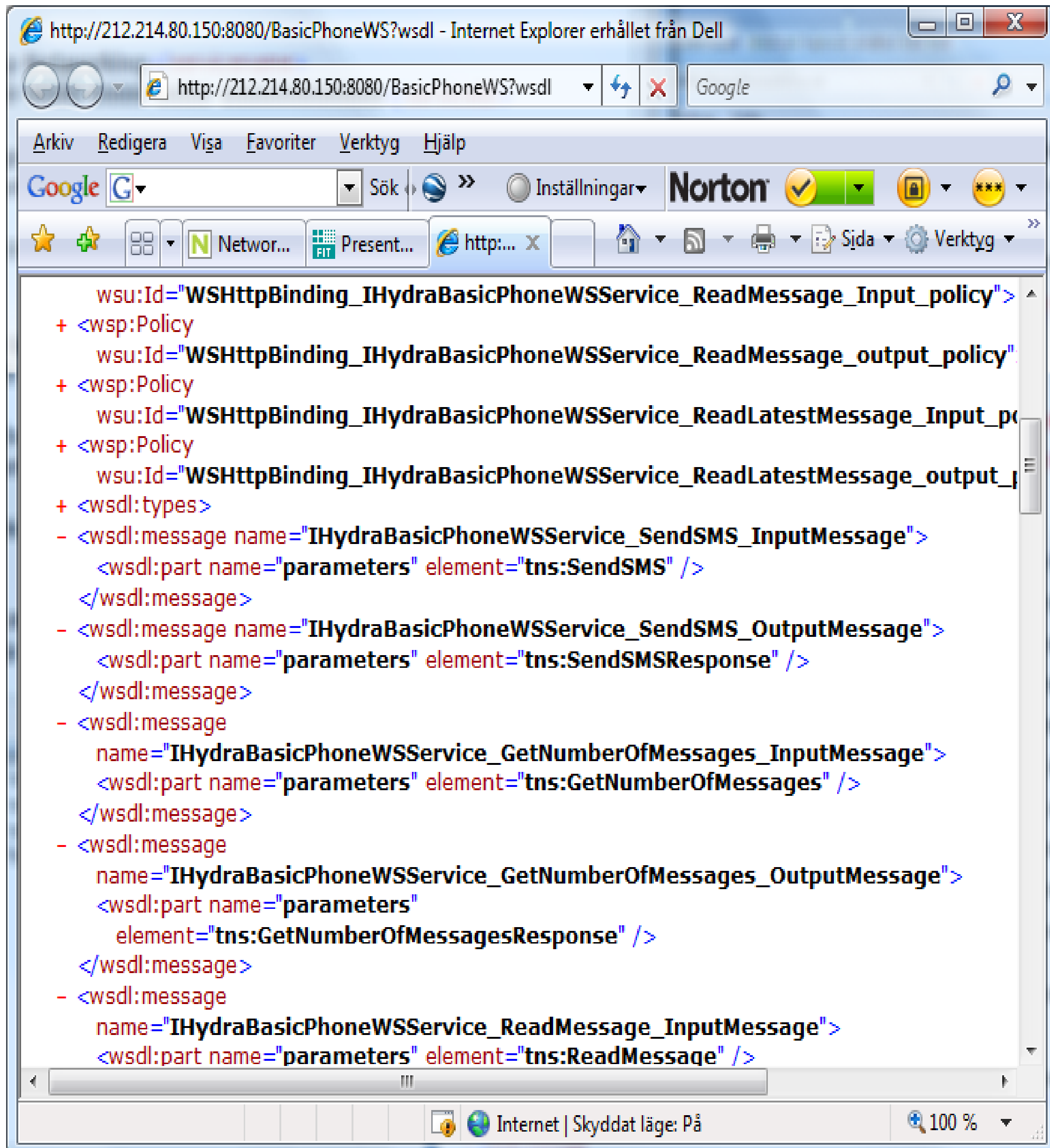

Figure 20: A WSDL (Web Service Description Language) for the device

5.4 Discovery Manager (Framework)

Hydra implements a 3-layered discovery architecture – physical, network and semantic discovery, see Figure given below.

In short the 3-layered discovery architecture works this way: First physical devices are discovered using native discovery protocols such as Bluetooth. Then the Hydra Middleware (Discovery Manager) creates a software wrapper that allows further extraction of metadata from the device and makes it available in a Hydra network. Finally the device ontology is used to fully resolve what type of device and what kind of functions it has and how the service interface looks like.

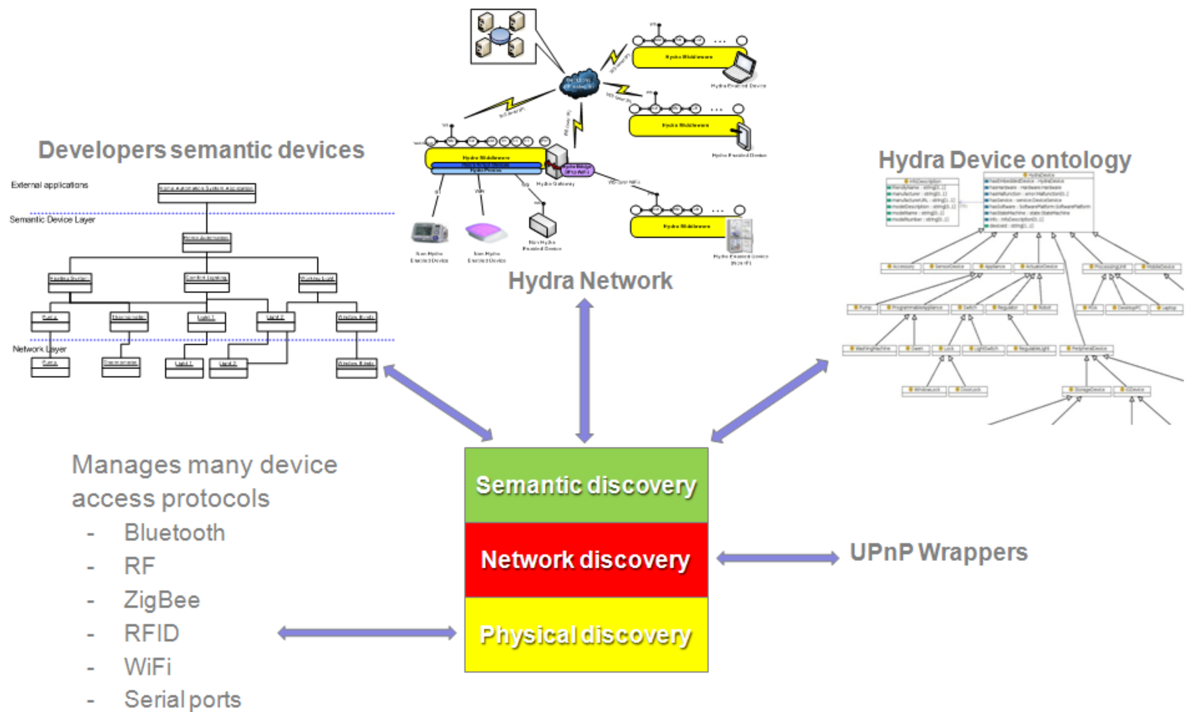

Figure 21: 3-layered discovery architecture in Hydra

5.4.1 Physical Discovery

At the lowest level the Hydra project is developing techniques for the discovery at the physical level. This will allow us to discover devices using communication protocols like Bluetooth, ZigBee, WiFi etc. Each of these protocols is handled by a specific Discovery Manager.

The Discovery Manager is part of the implementation (a sub-manager) of the Application Device Manager. This (sub-) manager also implements the base class for all protocol specific discovery managers in Hydra. A discovery manager keeps track of the devices it has discovered. As long as the devices are unresolved they are treated as Embedded devices of the Discovery Manager. A discovery manager runs locally on a gateway/PC where it looks for remote devices such as Bluetooth devices.

The following discovery managers exist with interfaces available:

- Bluetooth Discovery Manager
- SerialPort Discovery Manager
- RFSwitch Discovery Manager
- ZigBee Discovery Manager
- UPnP Discovery Manager
- RFID Discovery Manager
- External Discovery Manager

The External Discovery Manager now supports discovery of devices over the P2P architecture.

Network discovery based on UPnP

Once a device has been discovered at the physical level it needs to be discovered at a network level. This is done by creating a UPnP (Universal Plug and Play) wrapper to represent the device on the network. The UPnP wrapper then allows the device to be discovered at a network layer.

The UPnP (Universal Plug and Play) architecture offers pervasive peer-to-peer network connectivity of PCs, intelligent appliances and wireless devices. The UPnP architecture is a distributed, open networking architecture that uses TCP/IP and HTTP. It enables seamless proximity networking in addition to data transfer between networked devices at home, in the office and everywhere in between.

It enables data communication between any two devices under the command of any control device in the network. UPnP has a number of characteristics:

- Media and device independence. UPnP technology can run on any medium including phone lines, power lines, Ethernet, IR (IrDA), RF (WiFi, Bluetooth), and FireWire. No device drivers are used; common protocols are used instead.
- Common base protocols. Base protocol sets (Device Control Protocols, DCP) are used, on a per device basis.
- Operating system and programming language independence. Any operating system and any programming language can be used to build UPnP products. UPnP does not specify or constrain the design of an API for applications running on control points. OS vendors may create APIs that suit their customer's needs. UPnP enables vendor control over device UI and interaction using the browser as well as conventional application programmatic control.
- Internet-based technologies. UPnP technology is built upon IP, TCP, UDP, HTTP, SOAP and XML, among others.
- Programmatic control. UPnP architecture also enables conventional application programmatic control.
- Extensibility. Each UPnP product can have value-added services layered on top of the basic device architecture by the individual manufacturers.

The UPnP architecture supports zero-configuration, invisible networking and automatic discovery for a breadth of device categories from a wide range of vendors. Devices can dynamically join a network, obtain IP addresses, announce their names, convey their capabilities upon request, and learn about the presence and capabilities of other devices. DHCP and DNS servers are optional. A device can leave a network smoothly and automatically without leaving any unwanted state information behind.

5.4.2 External Discovery

External discovery enables Hydra gateways to locally represent all Hydra devices in the Hydra network even if they reside in a different physical network. This enables the developer to build applications that use devices in exactly the same way independently of their network location.

The basis for the external discovery process is synchronisation of information in-between the Application Device Managers in the network. For each of the found external Hydra devices a local device proxy is created using the SCPD of the external device. This will also copy all of the Hydra UPnP properties for the device such as the HIDs for the different device services.

The external discovery follows the following procedure:

1. Contact Network Manager to find all Device Application Managers in the network
2. Contact each of the Application Device Managers to retrieve a list of their local devices
3. Contact each device and use the generic Hydra Web Service to retrieve the device XML (SCPD)
4. For each device create a local device proxy using the device XML.

5.4.3 Semantic Discovery

Once the device is discovered as part of the network, it needs to be discovered semantically, i.e., the device needs to be related to the Hydra Device Ontology so that it is known what kind of device has been discovered.
Hydra uses two different XML structures to describe a device and its capabilities. First there is the device description, which contains various metadata regarding the device such as its type, the manufacturer, model etc. An example of a device description is shown below:

```
<device>
<deviceType>urn:schemas-upnp-org:device:waterPump:1</deviceType>
<friendlyName>GrundfosPump</friendlyName>
<manufacturer>Grundfos</manufacturer>
<manufacturerURL>http://www.grundfos.com</manufacturerURL>
<modelDescription>Pump</modelDescription>
<modelName>Grundfos Magna</modelName>
<modelNumber>X1</modelNumber>
<UDN>uuid:dac824ab-bca1-4d5c-93c5-578a0c697ba1</UDN>
<serviceList>
<service>
<serviceType>urn:schemas-upnporg:
service:grundfosPumpService:1</serviceType>
<serviceId>urn:upnp-org:serviceId:grundfosPumpService</serviceId>
<SCPDURL>_grundfosPumpService_scpd.xml</SCPDURL>
<controlURL>_grundfosPumpService_control</controlURL>
<eventSubURL>_grundfosPumpService_event</eventSubURL>
</service>
</serviceList>
</device>
```

Secondly, there is the SCPD (Service Control Point Description), which describes the capabilities of the device and how to invoke its different services. An example of service description is shown below:

```
<?xml version="1.0" encoding="utf-8"?>
<scpd xmlns="urn:schemas-upnp-org:service-1-0">
<specVersion>
<major>1</major>
<minor>0</minor>
</specVersion>
<actionList>
<action>
<name>GetStatus</name>
<argumentList>
<argument>
<name>ResultStatus</name>
<direction>out</direction>
<relatedStateVariable>Status</relatedStateVariable>
</argument>
</argumentList>
</action>
<action>
<name>SetTarget</name>
<argumentList>
<argument>
<name>newTargetValue</name>
<direction>in</direction>
<relatedStateVariable>Target</relatedStateVariable>
</argument>
</argumentList>
</action>
</actionList>
<serviceStateTable>
<stateVariable sendEvents="yes">
<name>Status</name>
<dataType>boolean</dataType>
</stateVariable>
<stateVariable sendEvents="no">
<name>Target</name>
<dataType>boolean</dataType>
</stateVariable>
```

```
</serviceStateTable>
</scpd>
```

A final part of the semantic discovery is the service discovery task to find a suitable service provided by specific device (or device type) in accordance to defined requirements. In the context of Hydra, the service discovery task defined this way can be used in various cases, for example:

- From a developer user point of view: to find the required service provided by a specific device in the process of development of basic communication patterns, such as composed (or orchestrated) services, choreography interfaces or service user interfaces.
- From a system or application point of view: to find the required service provided by specific device when executing the complex process requiring the service orchestration.
- Tools and matchmakers exist supporting the service discovery for both OWL-S and WSMO standards (description of this tools is out of scope of this deliverable), which may be used for particular approach.

In Hydra support for SAWSDL annotations is provided. As the SAWSDL approach does not explicitly support service discovery, there are two basic possibilities, which can be used in this case:

- The Service discovery process is realised by searching the SAWSDL according to provided semantic annotations.
- Using the annotations in SAWSDL file, the model of service is annotated in the Hydra service ontology and the discovery process is realised by matching the ontology concepts in accordance to specified requirements, similarly as in OWL-S/WSMO approach.

5.5 Ontology Manager

This section focuses on the creation of StateMachine ontology, which is used for the creation of state machine stubs to handle device run time status changes, and also for the diagnosis and monitoring rules used for achieving self* properties.

The StateMachine machine ontology is modelling the state machine concept in UML2. The concepts in StateMachine ontology is shown in the following figure:

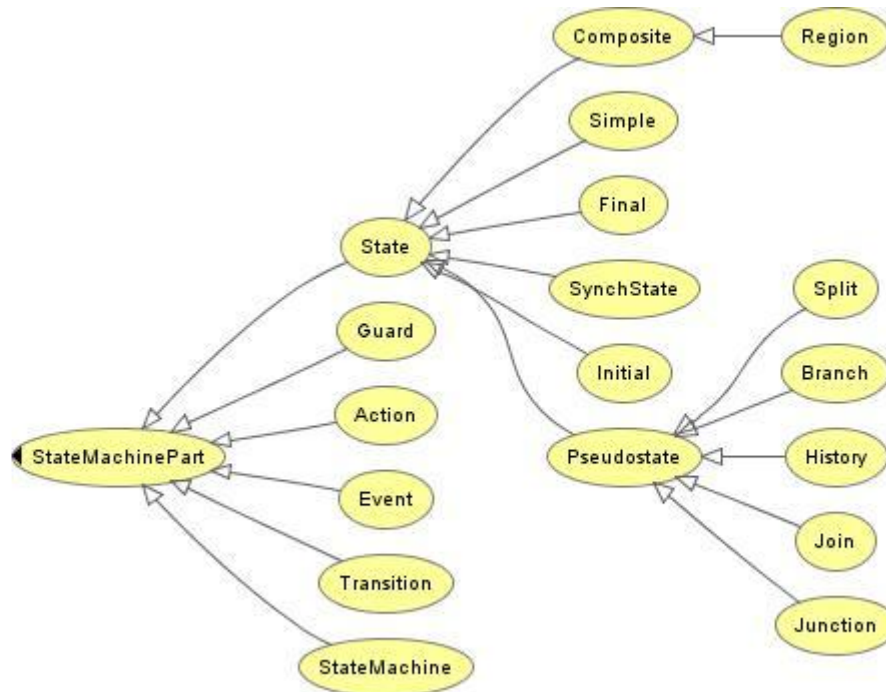

Figure 22: Ontology State Machine Concepts

How to create a state machine instance is described below. Taking the simple Thermometer device as an example, the following steps are involved to add this state machine instance.

1. Add all the states (starting, measuring, stopping) to the "Simple" concept, which means that the Thermometer state machine has these simple states. The adding of instances to an ontology is achieved with the "Individuals" tab, by clicking on the concept (for example "Simple"), and then click on the button "create instance" button. This is shown in the following figure. Protege is used as the OWL development environment. As every state for all devices need to be differentiated, each state is named according to its device, for example, "pico_th03_indoor_measuring". Set the "isCurrent" property to indicate whether this state is a current state or not, and add give it a label.

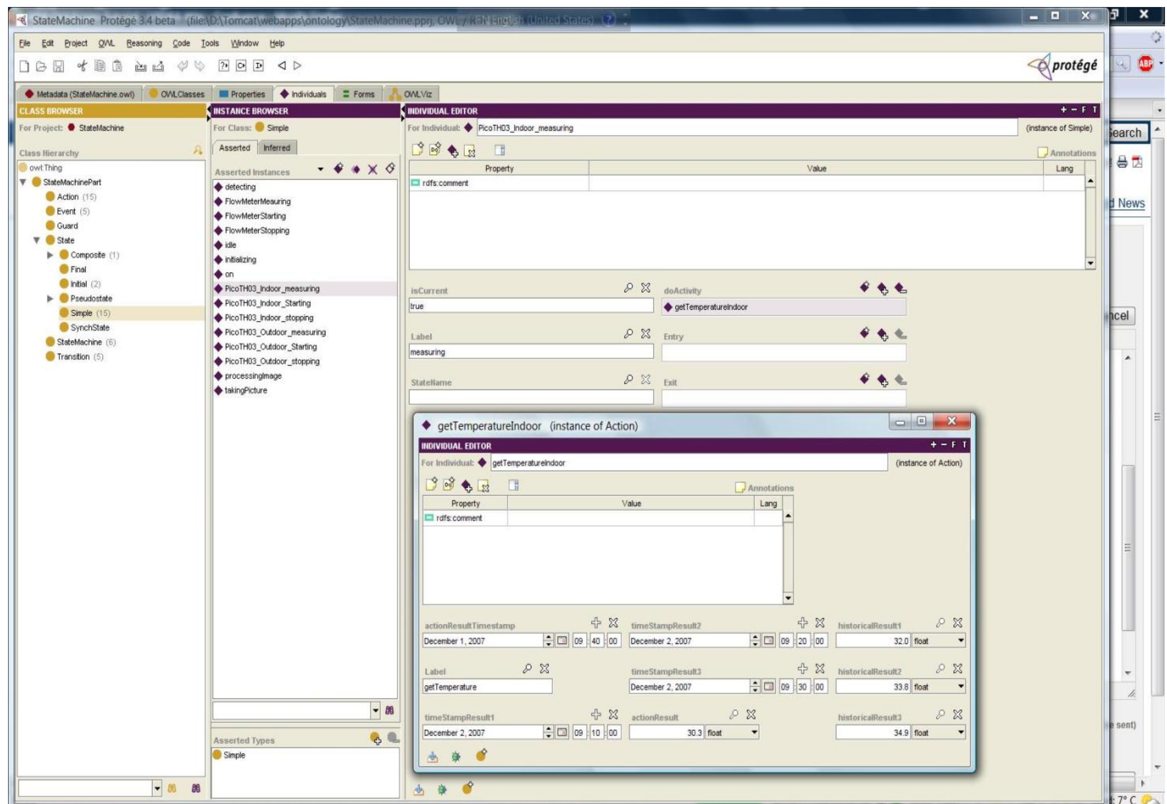

Figure 23: Ontology Browser

2. Add the "doActivity" associated with a state by clicking on the button 'create new resource' followed the "doActivity" property. For the measuring state, this is the '"getTemperatureIndoor' activity, which is used to model the indoor temperature measuring. An instance of data (not mandatory) is shown in the above figure.

3. Add the transition instances to "Transition" concept in the ontology, in the same way as in Step 1, with the meaningful name for example "T_measureToStop". Chose the source state and target state of the transition by clicking the "Select existing resource", choose the state that has just been created to reflect the transition direction.

4. This simple state machine is ready now. For this simple case, there is no need to have instances of "Guard", "PseudoState", etc.

Please see chapter 10 and use the online resources for further information.

5.6 Event Manager

The Hydra Event Manager provides publish/subscribe functionality, i.e., the ability for publishers to send a notification to multiple subscribers while being decoupled from them (in terms of, e.g., not holding direct references to subscribers). The specific variant of publish/subscribe implemented is topic-based publish/subscribe where the events are key/value pairs.

The Event Manager is deployed as a service in the Hydra network and implements the following interface:

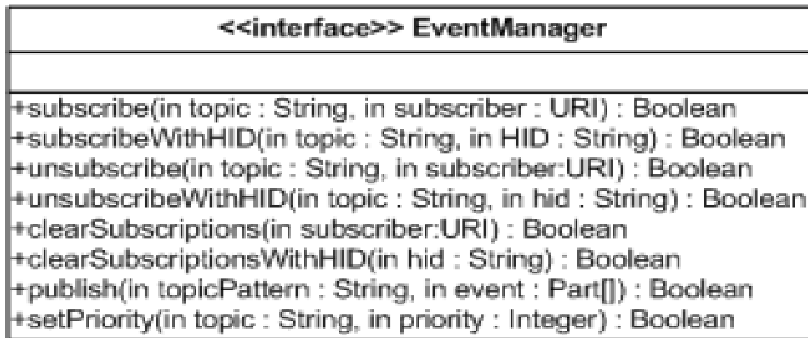

Figure 24- Event Manager Interface

Interaction with the Event Manager can be performed using this service, by creating an Event Manager client handling the calls to the Event Manager. To publish an Event to the Event manager, the *publish* method must be called, passing the *topic* of the Event, as well as an array of key-value *Part* objects, that specify additional data associated with the Event.

```
EventManagerPort em = { Get Event Manager Client };
Part[] parts = { Get Part Array };
em.publish("ExampleTopic", parts);
```

The code snippet above gives an example of using the EventManagerPort interface to publish an Event. An application can subscribe to receive notifications of events by calling the *subscribe* or *subscribeWithHID* methods. These methods take the *topic* of the events being subscribed to, along with callback information, such the Event Manager can send notifications when the events are published. With the *subscribe* method, this information is provided as a Web Service endpoint address, whereas the *subscribeWithHID* method takes the HID of the subscriber, to then call back through the Hydra Network Manager.

```
EventManagerPort em = { Get Event Manager Client };
em.subscribeWithHID("ExampleTopic", <Subscriber HID> );
```

The above code snippet gives an example of a subscriber subscribing to the Event Manager. Furthermore, subscribers must implement the following interface:

The figure below shows the resulting deployment:

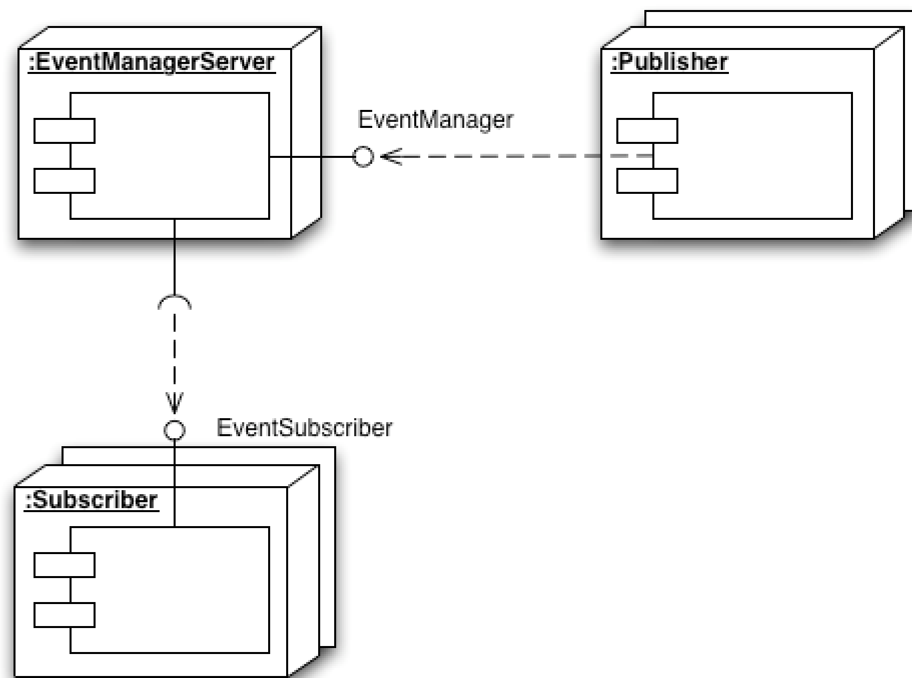

Figure 25: Event Manager Deployment

Given such a deployment, the figure below shows a typical interaction with the EventManagerServer (where the address is the address of the Subscriber web service that later should be notified):

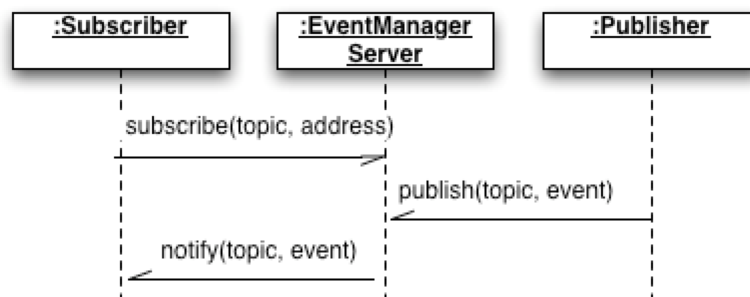

Figure 26: Subscriber Notification

5.7 Context Awareness Framework

The Context Awareness Framework (CAF) isused within Hydra to define and make use of certain context data, which is produced by sensors and devices at runtime. The user and developer can use this framework to define context situations and their individual actions (application behaviours) following these pre defined sets (there is also the term 'rule' used in the scope of the CAF). The CAF can also be queried to extract context information used within Hydra or at the application level.

5.7.1 Context Manager

The Context Manager component, as described in D12.8, brings the ability for context-awareness, that can be utilised by applications in whichever domain. The SDK component provides the interface for communicating with the Context Manager, for publishing created contexts (created with the IDE), to the Context Manager, as well as being able to query contexts, through Context Provisioning.

The Context Manager client can be retrieved using the methods described in the Hydra Middleware Clients chapter 5.1.2, providing access to the Context Manager interface as described by the Context Manager service of the Middleware API. The functionalities include:

- Adding contexts to a Context Manager
- Management of existing contexts
- Adding named queries
- Interface for executing named queries, and single queries

Existing Context Managers on the Hydra network have the SID (Service identifier - part of the CryptoHID) as shown below:

SID = com.eu.hydra.caf.cm

The Context Manager has a relatively simple configuration, using the Hydra Configurator. The main configurations that can be set, are:

- **ContextManager.PID**
  - Persistent Identifier of the Context Manager. If not given, this is automatically configured as "ContextManager:<machine-name>"
- **ContextManager.DaqcPID**
  - The PID of the Data Acquisition Component the Context Manager should use to get data. If no PID is given, then the Context Manager will try and find DAqC service in its local OSGi registry, and use that instead. Failing that, the Context Manager will be unable to subscribe for data.
- **ContextManager.DefaultEventManagerPID**
  - The PID of the Event Manager that the Context Manager should publish events to (as a result of reasoning performed by rule), if the rule action does not explicitly state an Event Manager to use.

5.7.2 Contexts

Contexts are represented by the *ContextSpecification* in the Hydra Middleware API. This object contains a definition of the context, including a set of properties and members of the context, as described in D12.8. In addition to the definition, it may also contain a set of subscriptions or rules (or both), depending on the type of the context (Application / Device / Semantic). Subscriptions are used for the subscriptions for data from the Data Acquisition Component (see chapter 5.7.2).

Context Specifications can be persisted as XML documents, but are transferred as objects, as described by the Context Manager WSDL, and also included in the Middleware API. The specifications can be created using the Context IDE components, as described in the relevant chapter of this document. These XML documents (with *.ctx* extensions) can be programmatically processed into the *ContextSpecification* object using the *ContextBuilderFactory* class also provided as part of the Middleware API. This class provides the functionality for marshalling the Java objects to and from the XML representation.

```
ContextManager cm = <Get ContextManager client>
...
InputStream input = Class.class.
      getResourceAsStream("contexts/exampleContext.ctx");
ContextSpecification ctx =
ContextBuilderFactory.buildContext(input);
ContextResponse response = cm.createContextSpecification(ctx);
```

The code snippet above gives an example of using the *ContextBuilderFactory* to build the *ContextSpecification*, and then sending it to a previously specified Context Manager, using the

createContextSpecification method. The *ContextSpecifcation* is built using a created InputStream to a locally stored file, retrieved using the Java ClassLoader mechanism. Alternatively, the builder can also take Strings as input. Additionally, the builder provides functionality for storing contexts locally, to marshall them back from the object to a String, or to a provided OutputStream.

The *ContextResponse* element returned contains a Boolean variable stating the success of the operation (the example given above being creating a context in the Context Manager). Additionally, the response may also contain a set of *ContextManagerError* objects, specifying any errors encountered during the processing of the request.

Examples of defined contexts are given in the relevant IDE section (see chapter 7.4).

5.7.3 Queries

The querying of context is a significant part of the Context Provisioning functionality of the Context Manager, allowing for applications to use contextual information - to become *"context-aware"*. The Context Manager provides multiple different options for querying context, that can be used using the interfaces exposed by the Context Manager. Mostly, queries are based around the *ContextQuery* object (typically sent as part of a *QuerySet*) found in the Middleware API.

Queries, like Rules, are based around Drools language rules, albeit the query only being a LHS (left-hand-side) of a rule, without the RHS action. As with contexts, *QuerySet* objects are created using the Context IDE components, and can be persisted and loaded using the *ContextBuilderFactory*. Query results are returned as encoded XML of the specified query outputs.

```
ContextManager cm = <Get ContextManager client>
...
InputStream input = Class.class.
      getResourceAsStream("queries/exampleQuerySet.ctq");
QuerySet querySet = ContextBuilderFactory.buildQuerySet(input);
ContextResponse response = cm.installQuerySet(querySet);
```

The code snippet above gives an example of loading a local *QuerySet* object, and installing it in the previously configured Context Manager. The interfaces of the Context Manager supporting querying of contextual data, are as follows:

Method	Description
executeNamedQuery(String name, Parameter[] params)	Executes a previously configured query, referenced by its *name*, using the provided parameters - if the query takes any.
executeSingleQuery(ContextQuery query)	Executes the single query passed that is not persisted in the Context Manager.
queryKeyValue(String contextId, String keyId, String keyType)	Queries for the value of the specified key-value pair, within the given context. *keyType* refers to the type of the pair being queried (Property or Member).

All queries return the *QueryResponse* object, containing the String result of the query, which may be encoded XML, or simply the string-value of some data. Additionally, as with the *ContextResponse*, it contains a set of *ContextManagerError* objects.

Example queries are given in the relevant IDE section (see chapter 7.4).

5.7.4 Context-sensitive Actions

The other modality of making an application *"context-aware"*, is through configuration of context-sensitive actions as the output of context rules, reasoning over contextual parameters to determine a significant change in context (as defined in the rules), that should then be reported to an external *Context Consumer* (e.g. application). This is therefore an asynchronous architecture, with the *Context Consumer* waiting to be reported to, rather than querying itself.

This itself can be achieved in two different ways. Firstly, the Context Manager may fire an event to a specified Event Manager, to which the context-aware application is subscribed, such that it receives the alert, and can interpret it as required in the flow of the application. This enables multiple applications to respond to a single change in interpreted context, utilising the many-to-many architecture of the Event Manager. As discussed in chapter 5.6 on the Event Manager, the *Context Consumer* would then need to implement and expose the *EventManagerSubscriber* interface, with the single method shown below:

```
public boolean notify(String topic, Part[] parts);
```

Secondly, the context-aware application may expose services on the Hydra network that should be called as the output of context rules, rather than calling an Event Manager. This can be called using the predefined Web Service call action, in the RHS of rules.

5.7.5 Data Acquisition Component

The Data Acquisition Component (DAqC) performs the acquisition of data, from data sources including sensors and services, as well as events published by a particular data source. The acquisition is based on subscriptions made to the Data Acquisition Component. It is primarily used by the Context Manager for the retrieval of data, but may also be used by other managers or applications that require frequent updates of data or events from a data source.

Configuration of the Data Acquisition Component is minimal, with the **Daqc.PID** configuration being the only entry of significance that can be set using the configurator, to define the persistent identifier of the

```
DataAcquisitionComponent daqc = <get DAqC Service>
...
DaqcSubscription daqcSub = <get Subscription>
DaqcSubscriptionResponse response = daqc.subscribe(daqcSub);
```

The code snippet above gives an example of using the *DataAcquisitionComponent* service to pass a set of subscripitions, as a *DaqcSubscription* object, to the DAqC, and receive the response. To then receive the reports of the acquired data through this description, be it Events or pulled data, the application must first implement the *DataReportingService* (of the Middleware API), as described later.

5.7.6 Subscriptions

The SDK functionality of the DAqC provides the interface for configuring and cancelling subscriptions for data, defined by the *DaqcSubscription* object in the Middleware API. This object contains the HID of the subscribing entity (for acquired data to be reported back to), as well as a set of subscriptions for data from the two protocols - PUSH and PULL (see D12.8).

Attribute ID	Description	Required?
EventManager.PID	PID of the Event Manager to subscribe to	✓
Event.Topic	Topic of the published Event	✓
EventSource.PID	PID of the data source publishing the Event	×

EventSource.SID	SID of the data source publishing the Event	✗
EventSource.HID	HID of the data source publishing the Event	✗

Table 2: Push Protocol Attributes

Attribute ID	Description	Required?
Datasource.PID	PID of the service to 'pull' data from	✓
Datasource.SID	SID of the service to 'pull' data from	✗
Pull.METHOD	Name of the method to call	✓
Pull.FREQUENCY	Frequency (in ms) at which to 'pull' data	✓
Pull.NAMESPACE	Namespace of the service	✗
Pull.SOAPACTION	SOAP Action of the service	✗
Pull.RETURNTYPE	Type of the data returned	✓
PlausibilityExpression	Regular Expression to determine plausibility	✗

Table 3: Pull Protocol Attributes

The two tables above show the attributes understood by each protocol, that are passed in the subscriptions to the DAqC.

5.7.7 Plausibility Checking

The Data Acquisition Component provides the facility for low-level analysis of retrieved data, to check its plausibility status. As this is at the level of data directly from a data source, the level of semantics involved within the plausibility checking mechanism, is relatively minimal.

The analysis of the data is performed by matching the data against a *Regular Expression*, which is a pattern used to match strings of data.

```
^(-[0-9]|(-[01])?[0-9]|([0-9])?[0-9]|1[0-2][0-9])$
```

The example regular expression, above, demonstrates an example of a numerical plausibility expression for a thermometer device that returns sensed temperature in Celsius. It specifies that the data has a plausible value of between -19 and 129 degrees. Anything outside this range would be flagged up as being implausible, and the subscriber notified, such that they can take appropriate action.

5.7.8 Data Reporting

Data retrieved by the Data Acquisition Component is reported back to the subscriber asynchronously, and therefore (as with the Event Manager) any component seeking to use the Data Acquisition Component to retrieve raw data and events from devices, must implement the *DaqcReportingService* interface, which is defined in the Middleware API, and shown below. This is a simple interface containing a single method, *reportAcquiredData*, that passes the retrieved data in the form of a Data Report.

```
public boolean reportAcquiredData(DataReport dataReport)
```

The *DataReport* object, as shown above, is the object in which the retrieved data is stored, to be reported back to the subscriber, and is generic to handle data from both *Push* and *Pull* protocols. It references the *Data Id* specified for the data being reported, so that the subscriber can identify the data which is being reported, as well as the protocol used to retrieve it. The data itself is passed as a set of key-value attributes. When reporting data from the *Push* protocol, these key-value attributes represent the "parts" content of the Event about which the protocol was notified. In the case of the

Pull protocol, this is simply a single key-value piece of data, holding the value retrieved. All data sent in the report is represented as a String.

As well as containing the data being reported, the *DataReport* object also contains both an update on the status of the subscription that it is reporting for. This provides the mechanism for informing the subscriber of a potential malfunction in the data source - when implausible data has been retrieved, and also of when a data source is no longer available and data could not be retrieved. This is essential to ensure that the subscriber is aware that they are no longer receiving data updates from this device. It is then the responsibility of the subscriber to re-establish the subscription, as and when the data source is available again.

5.8 Access Control Policy Framework

The Policy Framework provides policy-driven, access-control protection for Hydra devices and applications. Policies can be utilised to ensure access to devices and applications is limited only to those permitted access, including the ability to restrict the level of discoverability of an end-user's devices and applications.

The Policy Framework, consisting of its various components, provides the functionalities to create, update, and maintain Policies, in addition to its core of evaluating access requests, and enforcing the decisions made. The SDK functionality of the Access Control Policy Framework comes with three distinct interfaces, as well as another interface for extension, these being:

- Policy Enforcement Point
  - Called at the point of interception of a request
  - Formats the credentials of the request in to an XACML RequestCtx object, and calls the PDP for a decision
  - Enforces the returned decision, handling any obligations specified in the Policy
- Policy Decision Point
  - Receives the XACML RequestCtx object from the PEP
  - Analyses the request against the policies stored in its policy repository
  - Returns the determined decision
- Policy Administration Point
  - Interface exposed by the PDP for the administration of XACML policies
  - Active / deactivate XACML policies
- Policy Information Point
  - Extension interface for PDP
  - Adds functions for the PDP to use when they are referred to in XACML policies

5.8.1 Policy Enforcement Point

Typically, in the context of communication in Hydra, the Access Control Policy Framework is used to provide access control at the level of the Network Manager, such that access decisions can be made on receiving a request, through the SOAP Tunnel, for a hosted service, before actually forwarding the payload of the request to the endpoint service. The Policy Enforcement Point (PEP), therefore, is utilised by the Network Manager, when it receives a call, forwarding the various credentials it has about the request, to the Policy Enforcement Point.

Although the PEP itself does not expose a service to the Hydra Network, it does register itself with an HID and certificate, such that it can be identified as being a PEP. The SID of the PEP is as follows:

SID = com.eu.hydra.policy.pep

Configuration of the PEP bundle specifies the following important configurations:

- **Pep.PID =** The PID of the registered PEP service
- **Pep.PdpPID =** The PID of the PDP service that the PEP should use to retrieve an access decision

The PEP exposes a couple of methods also, used by the Network Manager, to pass the credentials of a request. These are:

```
public PepResponse requestAccessDecision(String senderHid,
                                          String senderCert,
                                          String receiverHID,
                                          String receiverCert,
                                          String soapMsg,
                                          String sessionId);

public PepResponse requestAccessDecisionWMethod(String senderHID,
                                                String senderCert,
                                                String receiverHID,
                                                String receiverCert,
                                                String method,
                                                String sessionId);
```

Both methods request an access decision, but for different contexts. The Network Manager uses the *requestAccessDecision* method, passing on the complete SOAP Message received from which the PEP extracts the credentials of the action to be performed, whereas the *requestAccessDecisionWMethod* method passes the name of the method directly instead. The *senderCert* and *receiverCert* arguments are the encoded CryptoHID certificates for the two entities involved at either end of the request.

5.8.2 Policy Decision Point

The Policy Decision Point (PDP) is a manager on the Hydra network that registers two different services, one for the process of access requests, returning a decision, and another for the administration of the XACML policies that the PDP uses in these decision making processes. This administration service is described in the next chapter.

The *PolicyDecisionPoint* interface, of the Hydra Middleware API, declares just one single method, for the evalutation of the XACML RequestCtx, as follows:

```
public String evaluate(String requestXml);
```

This method takes the RequestCtx object, encoded as a String, and evaluates it against the set of policies in the policy repository. The ResponseCtx, containing the decision made along with any obligations with the decision, is returned as encoded XML.

The SID of the PDP service is:

SID = com.eu.hydra.policy.pdp

The PDP has minimal configuration, using the configurator, as follows:

- **PdpService.PID =** PID of the PDP
- **Pdp.UseDatabase =** true / false depending on whether XMLDB based storage or file-based storage is to be used for XACML policies

5.8.3 Policy Administration Point

The interface for actually authoring XACML policies is part of the IDE, and discussed in the chapter 7.7. It uses the interface exposed by the PDP that is distinctly separated from the service performing

the decision functionality, as described in the previous chapter. It sets out the following methods as the *PdpAdmin* interface of the Middleware API:

```
public boolean activatePolicy(String policyId);
public boolean deactivatePolicy(String policyId);
public String[] getActivePolicyList();
public.String[] getInActivePolicyList();
public boolean publishPolicy(String policyId, String policyXML);
public boolean removePolicy(String policyId);
public String getPolicy(String policyId);
```

The key method involved here, is the *publishPolicy* method that publishes the policy with the given id, *policyId*, with the XML-encoded content provided with the *policyXML* argument.

The SID of the PDP service is:

SID = com.eu.hydra.policy.pap

The PAP has even less configuration than the PDP, using the configurator, as follows:

- **PdpAdminService.PID =** PID of the PDP Administration service

5.8.4 Policy Information Point

The Hydra PDP is designed to be extensible, to easily allow for new functionality to the PDP through adding additional Policy Information Point (PIP) components, which includes the ability to resolve certain attributes, add additional functions that can be used in policies, add new data types, and so on.

PIPs are implemented as OSGi bundles that register services, recognised by the PDP, that it uses to extend the functionality, adding new Functions and Attribute Finders to the PDP at runtime. These interfaces are *PipFunction*, and *PipModule*.

PipFunction provides a method that the PDP can use to retrieve the custom XACML Functions (com.sun.xacml.cond.Function) that it then installs to the PDP Function factory, such that they are then immediately available for use. Therefore, the interface is simply:

```
public interface PipFunction {

      public Set<Function> getFunctions();
}
```

Implemented *PipModule* components define XACML AttributeFinders, that can retrieve attributes that are not available in the request, but are specified in an XACML policy. The *PipModule* itself is essentially just an extension of the AttributeFinderModule defined by the XACML 1.x implementation by Sun, providing a service name unique to the Hydra Access Control Policy Framework. Therefore, the *PipModule* interface is:

```
public abstract class PipModule extends AttributeFinderModule {
}
```

5.9 Quality of Service Manager

Hydra operates within the scope of network embedded devices, like mobile phones, laptop and desktop computers, etc. These devices have different capabilities in terms of computational power, screen size and memory, etc. To address this fact and, e.g. playing media files, the QoS Manager can be used to optimize the quality of the file depending on which device it is played on.

5.9.1 Functionalities

The QoSManager provides three functionalities:

1. To request the best-suitable service out of a range of Hydra services with same functionality.
2. To request a ranking list of best-suitable services.
3. To request a set of quality views of service parameters that are regularly updated from ontology.

5.9.2 Dependencies

First of all the QoSManager requires the new Hydra Commons and the Network Manager as entry point to Hydra middleware.

This tutorial and the usage of the QoSManager require knowledge of new Hydra Commons. For processing specific service requests, the QoS Manager needs to query Hydra Ontology. For this the Ontology Manager provides an accessible web interface for retrieving data values of QoS properties of Hydra services and the (embedded) devices on which these services are running.

5.9.3 Used by

Inside an application built on Hydra, the Application Service Component is intended to consume the QoSManager for retrieving the best-suitable services under consideration of QoS properties.

The self*-management component requires an regular update of the quality views of services parameter, and thus needs to request a specific set of QoS properties for self-adaption.

5.9.4 Prerequisites

You need to have installed:

1. OSGi environment (e.g. Equinox)
2. MySQL (MySQL 5.1; the typical and complete version as well) and
3. Protégé 3.4.1

5.9.5 Installation

1. Install new Hydra Commons and latest version of Network Manager.
2. Download the QoS Manager.
3. Create and Adapt QoSManager.properties configuration file in ECLIPSE\QoSManager\config\
4. Start a database server (For example, MySQl)

Start Protégé and load Ontology from 'Ontology' folder inside QoSManager.

Configuration

Before running QoS Manager some configuration needs to be done inside Protégé:

1. Select from the menu **File -> Open...**

2. Navigate to the destination of the recently checked out QoS Manager Bundle in order to select the 'HydraOntologyManager.pprj' project file included in the 'Ontology'

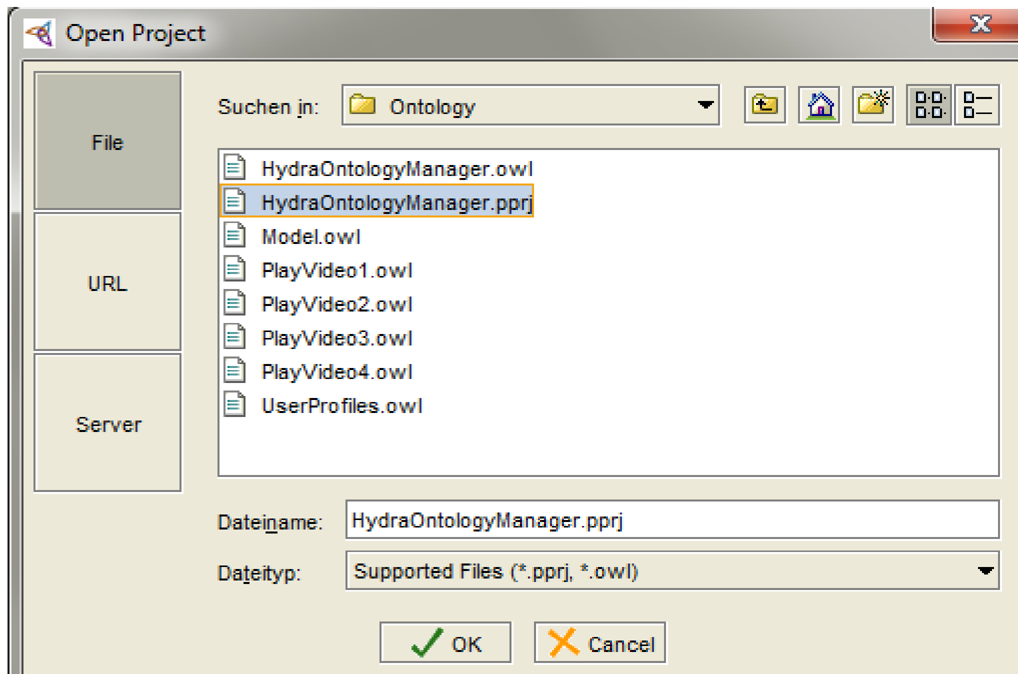

Figure 27: Select project

And the OntologyProject is loaded:

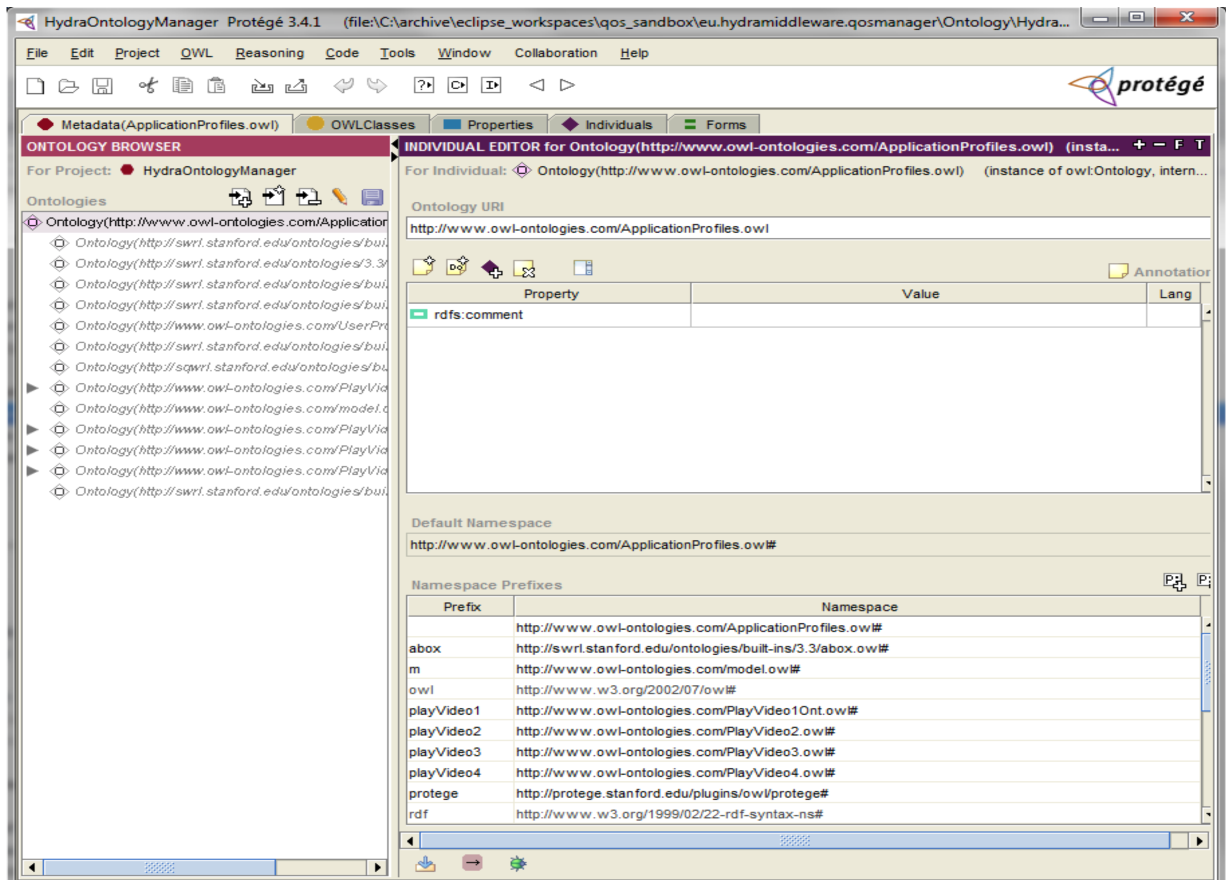

Figure 28: Ontology Browser

3. Select from the menu OWL -> Ontology repositories..., and the following dialogue is loaded

Repository Manager
Project repositories Global repositories
Project repositories
Local folder C:\archive\eclipse_workspaces\qos_sandbox\eu.hydramiddleware.qosmanager\Ontology
Force Read-Only
http://www.owl-ontologies.com/ApplicationProfiles.owl
(C:\archive\eclipse_workspaces\qos_sandbox\eu.hydramiddleware.qosmanager\Ontology\HydraOntologyManager.owl)
http://www.owl-ontologies.com/PlayVideo1Ont.owl
(C:\archive\eclipse_workspaces\qos_sandbox\eu.hydramiddleware.qosmanager\Ontology\PlayVideo1.owl)
http://www.owl-ontologies.com/PlayVideo2.owl
(C:\archive\eclipse_workspaces\qos_sandbox\eu.hydramiddleware.qosmanager\Ontology\PlayVideo2.owl)
http://www.owl-ontologies.com/PlayVideo3.owl
(C:\archive\eclipse_workspaces\qos_sandbox\eu.hydramiddleware.qosmanager\Ontology\PlayVideo3.owl)
http://www.owl-ontologies.com/PlayVideo4.owl
(C:\archive\eclipse_workspaces\qos_sandbox\eu.hydramiddleware.qosmanager\Ontology\PlayVideo4.owl)
http://www.owl-ontologies.com/UserProfile.owl
(C:\archive\eclipse_workspaces\qos_sandbox\eu.hydramiddleware.qosmanager\Ontology\UserProfiles.owl)
http://www.owl-ontologies.com/model.owl
(C:\archive\eclipse_workspaces\qos_sandbox\eu.hydramiddleware.qosmanager\Ontology\Model.owl)
Local folder C:\archive\eclipse_workspaces\qos_sandbox\eu.hydramiddleware.qosmanager\Ontology
Force Read-Only
http://www.owl-ontologies.com/ApplicationProfiles.owl
(C:\archive\eclipse_workspaces\qos_sandbox\eu.hydramiddleware.qosmanager\Ontology\HydraOntologyManager.owl)
http://www.owl-ontologies.com/PlayVideo1Ont.owl
(C:\archive\eclipse_workspaces\qos_sandbox\eu.hydramiddleware.qosmanager\Ontology\PlayVideo1.owl)
http://www.owl-ontologies.com/PlayVideo2.owl
(C:\archive\eclipse_workspaces\qos_sandbox\eu.hydramiddleware.qosmanager\Ontology\PlayVideo2.owl)
http://www.owl-ontologies.com/PlayVideo3.owl
(C:\archive\eclipse_workspaces\qos_sandbox\eu.hydramiddleware.qosmanager\Ontology\PlayVideo3.owl)
Close

Figure 29: Repository Manager

4. On each tab, i.e. on the Project repositories and Global repositories tab, press the '+'-button, and select 'Local Folder' as the radio button option, and select as your destination the 'Ontology' folder.
5. Close the dialogue and another message dialogue asking you to perform a reload appears.

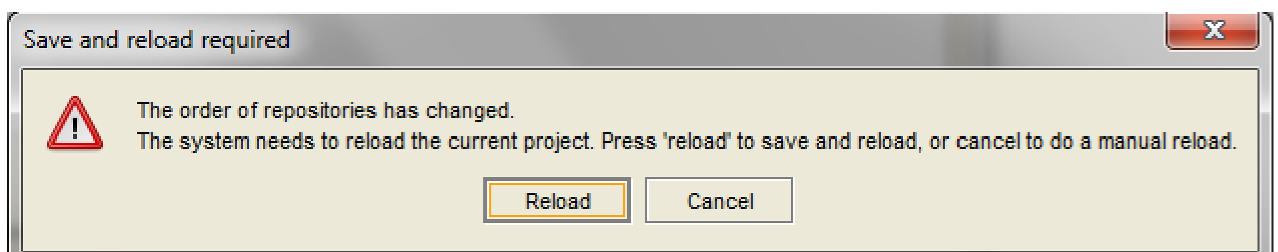

Figure 30: Dialog Box

6. Select the 'Reload' button and finally save the project, and close Protégé.

While running configure via Hydra Commons Configurator

(First, setup the run configuration of the OSGi bundle set, as described in next section)

For providing a unique and better interface for configuration, the QoS Manager has been ported to the Hydra Commons Configurator.

While running the OSGi configuration, all the listed properties can be changed via http://localhost:8082/HydraStatus#.

Just type in a value field of a property and push the '**Update Configuration**' button.

To make the QoS Manager perform correctly, the following fields (PID, ontology folders, MySQL configuration etc) need to be adapted accompanied with exemplified values:

- QoSManager.PID= QoSManager:Hydra
- hydraontology.path=C:/archive/workspace/hydra/Ontology/HydraOntologyManager.owl
- globalrepositories.path=C:/archive/workspace/hydra/Ontology/
- mysql.username=root
- mysql.password=otto81
- mysql.port=3306
- mysql.defaultdb=hydra

See also this screenshot given below:

Figure 31: Hydra Status Page

5.10 Execution in eclipse

The bundle has been tested using the equinox OSGi framework.

1. Create in the 'Run Configurations...' menu a new launch configuration for the OSGi framework. As all related bundles have been developed as OSGi Declarative Services there is no need for setting start levels.
   Choose the prepared QoSManager lauch configuration, or create a new run configuartion and select the following bunldes depicted in the screeenshot below (please consider to select the two mortbay.*-bundles - for them the screen was too small):

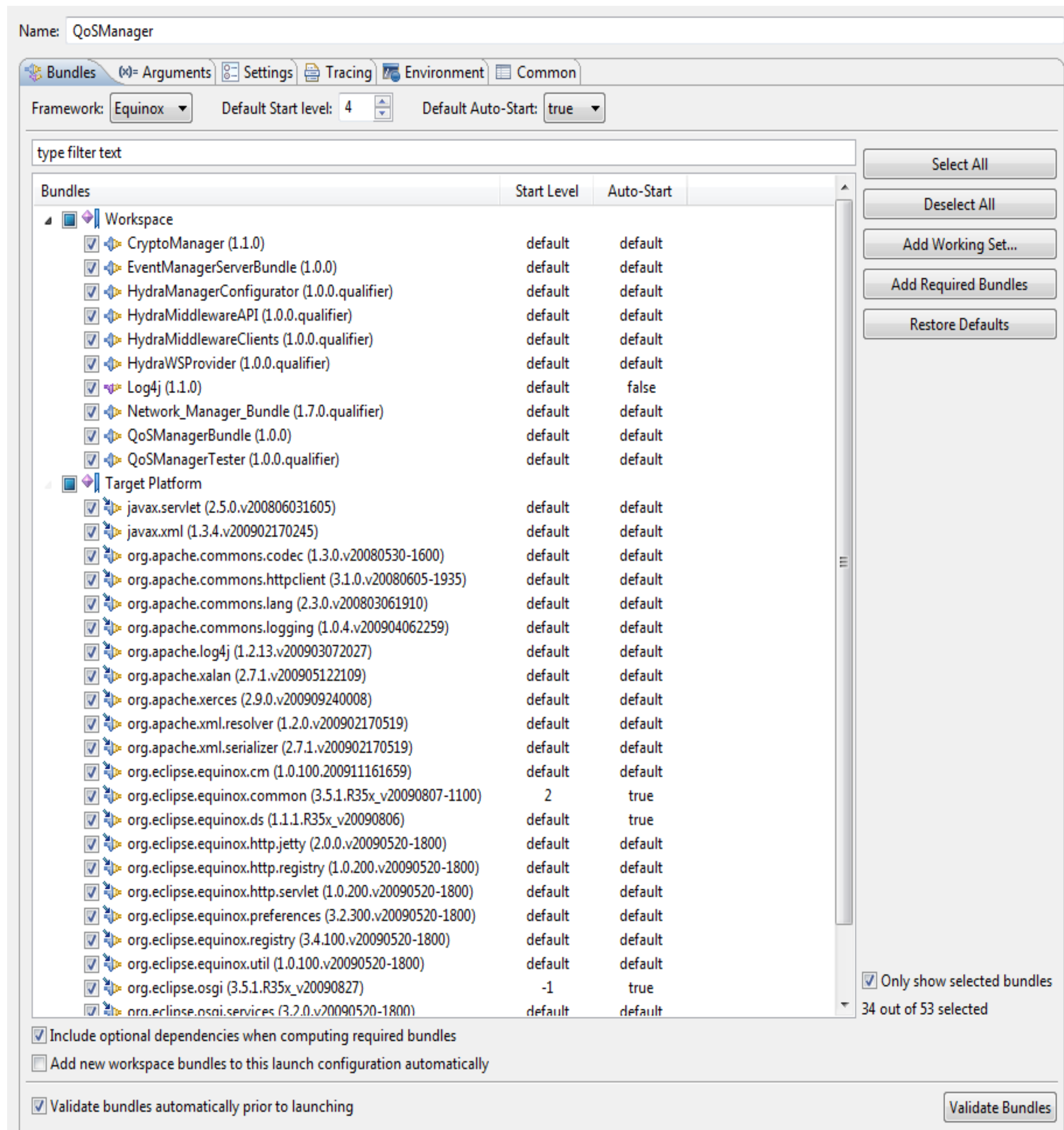

Figure 32: Bundles for QoS Manager

Switch to the 'Arguments' tab and put inside the VM Arguments:

```
-Declipse.ignoreApp=true -Dosgi.noShutdown=true  -
Dorg.osgi.service.http.port=8082
```

As Protégé related libraries require a large amount of user memory it is recommended that the VM line with -Xms and Xmx, in particular -Xmx option in order to prevent a Java heap space exception:

```
-Declipse.ignoreApp=true -Dosgi.noShutdown=true  -
Dorg.osgi.service.http.port=8082 -Xms512m -Xmx1024M
```

5.10.1 Usage

A QoS tester bundle can be downloaded. It has been implemented according to new Hydra Commons.

There is the following code in component class activate-method calling the getRankingList functionality of QoS Manager:

```
protected void activate(ComponentContext context){

         RemoteWSClientProvider service = (RemoteWSClientProvider)
context.locateService(RemoteWSClientProvider.class.getSimpleName());

                 try {
                         nm =
                                 (NetworkManagerApplication) service
                         .getRemoteWSClient(
                                 NetworkManagerApplication.class.getName(),
//
  "http://localhost:8082/axis/services/NetworkManagerApplication"
                                 null,
                                 false);

                         createCryptoHID();

                         // Get Remote QoSManager
                         String qosManagerHID = getQoSManagerHID(myHID);


                         //Change if QoSManager is running on remote
machine
                         String qosManagerIP="localhost";

                         String targetUrlHydraEventManager =
                                 "http://" +qosManagerIP+ ":" +

  System.getProperty("org.osgi.service.http.port")+
                                                 "/SOAPTunneling/0"
                                                         + "/" +
qosManagerHID +
                                                                 "/0/hola";

                         QoSManager qosManager =
                                 (QoSManager)
service.getRemoteWSClient(QoSManager.class.getName(),
targetUrlHydraEventManager, false);

                         String result =
qosManager.getRankingList(xmlQoSRequest);
```

```
                        LOG.debug("result="+result);

                } catch (IOException e) {
                        LOG.error(e.getMessage());
                } catch (Exception e) {
                        LOG.error(e.getMessage());
                }


  }
```

The appropriate response to this request looks like:


```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<ResultList xmlns="http://qosmanager.hydra.eu.com">
<Rank>
<Position>1</Position>
<Device>Dell Laptop</Device>
<ServiceName>PlayVideo3</ServiceName>
<HID>http://en.wikipedia.org/</HID>
<Rate>5</Rate>
<AveragePercentage>62.33%</AveragePercentage>
<Details>
<Detail>
<Property>cost</Property>
<Value>2.0</Value>
<Unit>euro</Unit>
</Detail>
<Detail>
<Property>powerconsumption</Property>
<Value>90.0</Value>
<Unit>watts</Unit>
</Detail>
<Detail>
<Property>screensize</Property>
<Value>15.0</Value>
<Unit>inch</Unit>
</Detail>
</Details>
</Rank>
<Rank>
<Position>2</Position>
<Device>Pioneer Plasma</Device>
<ServiceName>PlayVideo1</ServiceName>
<HID>http://www.youtube.com/</HID>
<Rate>4</Rate>
<AveragePercentage>34.61%</AveragePercentage>
<Details>
<Detail>
<Property>cost</Property>
<Value>3.0</Value>
<Unit>euro</Unit>
</Detail>
<Detail>
<Property>powerconsumption</Property>
<Value>380.0</Value>
<Unit>watts</Unit>
</Detail>
<Detail>
<Property>screensize</Property>
<Value>50.0</Value>
<Unit>inch</Unit>
```

```
</Detail>
</Details>
</Rank>
<Rank>
<Position>3</Position>
<Device>Samsung Projector</Device>
<ServiceName>PlayVideo2</ServiceName>
<HID>hid2</HID>
<Rate>3</Rate>
<AveragePercentage>66.67%</AveragePercentage>
<Details>
<Detail>
<Property>cost</Property>
<Value>4.0</Value>
<Unit>euro</Unit>
</Detail>
<Detail>
<Property>powerconsumption</Property>
<Value>48.0</Value>
<Unit>watts</Unit>
</Detail>
<Detail>
<Property>screensize</Property>
<Value>80.0</Value>
<Unit>inch</Unit>
</Detail>
</Details>
</Rank>
</ResultList>
```

5.11 Storage Architecture

The Hydra Storage Architecture is designed to enable developers to integrate any kind of storage into the Hydra middleware. Therefore storage is realised as a virtual devices. These devices have to be Hydra enabled, so they can be recognised by the Hydra Discovery Manager and accessed using the Network Manager. From the application developer's view these devices behave like any other Hydra enabled device. Figure 33 below shows a short overview over the basic architecture.

The most important part for developers integrating storage is the Storage Manager Device. As the Storage Devices the Storage Manager Device is realised as virtual Hydra enabled device. One of these exists on any physical Hydra device bringing storage into the network. The Storage Manager Device is responsible for the administration of the local Storage Devices. A closer description of the Storage Manager Device is given in the following sections.

The Storage Devices are connectors to some kind of storage in Hydra. There can be different devices using different APIs for different kinds of storage. The File System Device for example is designed to realise an easy to use access to storage structured in files and directories. It is also shown that the API is specialised to support this kind of storage. A Database Storage Device would offer an API specialised for database storage by using, e.g. SQL queries.

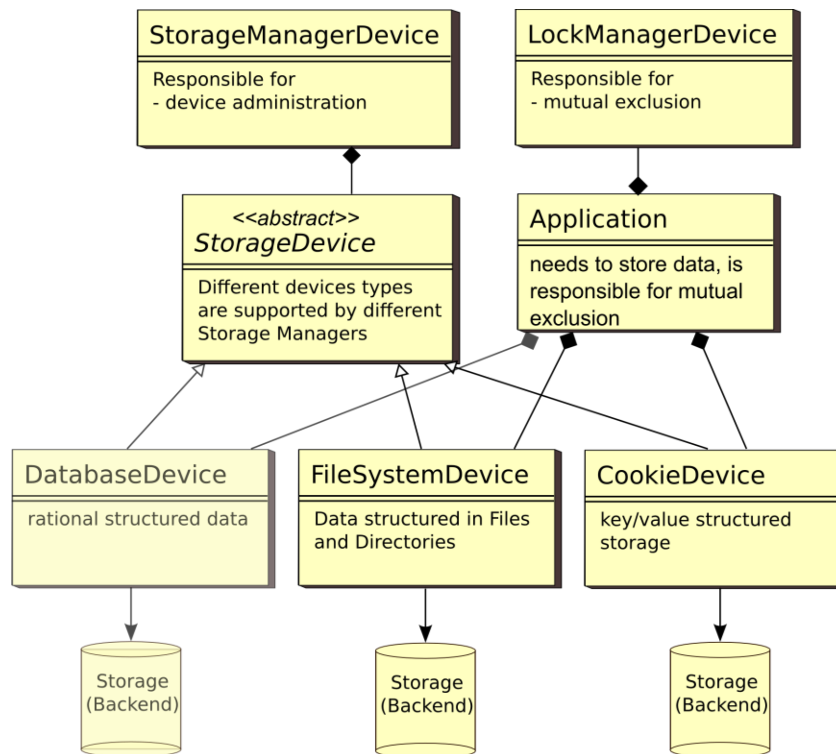

Figure 33: Basic architecture of storage in Hydra

5.11.1 Implementation details

The managers implemented in the prototype of the Hydra Storage Architecture were developed as UPnP devices created by Limbo and are available as OSGi bundles. The managers can be reached using UPnP or using the created Web services using the same API. Therefore the API of all managers has to use only those two devices which use UPnP.

The first limitation concerns numbering. UPnP supports only 32 bit integers, while the Hydra Storage Architecture often needs 64 bit. Therefore all numbers are sent as strings. Further UPnP does not support complex types. Therefore all complex data types are converted into XML and sent as a string. The StorageManagerCommonBundle is an OSGi bundle that implements all the complex data types. It can be used by Java applications or other OSGi bundles to convert Strings in Objects and Objects in Strings. The complex data structures used most often are the response types. As UPnP does not support exceptions, any method that could fail has to return an error code.

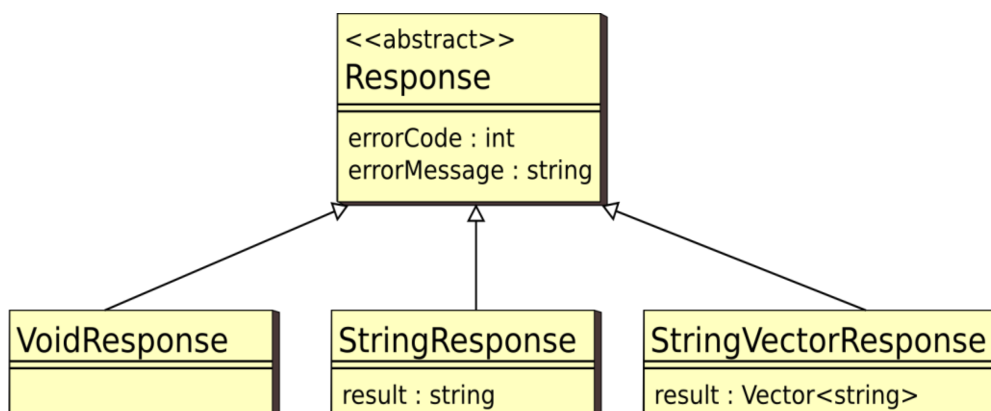

Figure 34: Some examples for Responses

The figure above shows a small subset of the used responses. The abstract class Response holds the error code of an operation. The class ErrorCodes defines the legal values and the meaning of the code in different areas. The error code 0 is reserved as marker for a successful operation. Additionally a response includes an error message. Here the sender of the response may store error information, which could help users or developers to discover why an operation failed. The subclasses of Response differ in the type of the delivered result. The type can be none (as in VoidResponse), a simple type (as in StringResponse), or a complex type stored as XML data in a string (as in StringVectorResponse). As you will see later on, there are a variety of different subclasses of responses for different response types.

Listing 5.11.1: StringResponse as an example of the XML representation of a response

```
<stringResponse>
<error errorcode="1 ">
error message
</error>
<result>
<value>
response
</value>
</result>
 </stringResponse>
```

Listing above shows the XML representation of a StringResponse. At first it is important to know that each subclass sets its own title of the root element of the response. The title should be equal to the class name and should be the type of the result followed by Response. In the example in line 1 the root type is set to StringResponse. In line 2 the element error can be seen, which holds the error code as an attribute. The error message is stored as the content of this element in line 3. An error can also be an empty element if no error message is returned. The element result in line 5 is owned by the subclass. It exists in any subclass but attributes and content differ. A StringResponse stores an element of type value here if the result of the StringResponse was not null.

Listing 5.11.2: Dictionary representation in XML

```
<properties>
<key1>
 value1
</key1>
<key2>
value2
</key2>
</properties >
```

The class offers methods to build Strings containing responses without caring about the response objects. This can be useful if a developer uses the Limbo generated stubs of the devices directly. This class also offers methods to convert special objects into XML. An example for such an object is shown in the Listing 5.11.2. Each dictionary is placed in element properties. The contents of this element are the key/value pairs of the dictionary. The key is used as name of the tag while the value is used as content of the tag.

5.11.2 Storage Manager Device

As explained before the Storage Manager Device is responsible for the administration of the Storage Devices. Therefore it supports a number of Storage Device types. The implementation supports the

File System Devices. The Storage Manager Device is implemented as prototype in the OSGi Bundle StorageManagerDeviceServer.

5.11.3 API

The Storage Manager Device is used to administrate storage. Therefore its API has to support the configuration of all supported Storage Devices.

StorageManagerDevice

createFileSystemDevice(filesystemname : string, config : string) : VoidResponse
createFileSystemDeviceLocal(filesystemname : string, config : string) : VoidResponse
deleteFileSystemDevice(filesystemname : string) : VoidResponse
deleteFileSystemDeviceLocal(filesystemname : string) : VoidResponse
getSupportedFileSystemDevices() : StringVectorResponse
getFileSystemDevices() : StringVectorResponse
findFileSystemDevice(filesystemname : string) : StringResponse
updateFileSystemDevice(filesystemname : string, config : string) : VoidResponse
updateFileSystemDeviceLocal(filesystemname : string, config : string) : VoidResponse

Figure 35: API of a Hydra Storage Manager Device

Figure 35 above shows the API of the Storage Manager Device. The following list includes a description of the single methods:

createStorageDevice(config : string) : StringResponse This method is responsible for setting up a new storage device.

createStorageDeviceLocal(config : string) : VoidResponse This method is only used for the communication between Storage Manager Devices. By this method a Storage Manager Device can propagate the creation of a Storage Device to other Storage Managers, which should also hold the device.

deleteStorageDevice(id : string) : VoidResponse This method removes an existing device.

deleteStorageDeviceLocal(id : string) : VoidResponse This method is only used for the communication between Storage Manager Devices. By this method a Storage Manager Device can propagate the deletion of a Storage Device to other Storage Managers, that should also delete the device.

getStorageDevices() : StringVectorResponse Gives a list of the IDs of all local hosted storage devices.

findStorageDevice(id : string) : StringResponse This method delivers the configuration of a Storage Device.

updateStorageDevice(config : string) : VoidResponse This method is responsible for updating an existing storage device.

updateStorageDeviceLocal(config : string) : VoidResponse This method is only used for the communication between Storage Manager Devices. By this method a Storage Manager Device can propagate the update of a Storage Device to other Storage Managers, which also hold the device.

HydraSMConnector
HydraSMConnctor(wsAddress : String) createFileSystemDevice(filesystemname : String, config : String) : VoidResponse deleteFileSystemDevice(filesystemname : String) : VoidResponse getSupportedFileSystemDevices() : StringVectorResponse getFileSystemDevices() : StringVectorResponse findFileSystemDevice(filesystemname : String) : StringResponse updateFileSystemDevice(filesystemname : String, config : String) : VoidResponse

Figure 36: API of HydraSMConnector

5.11.4 Client

As part of the prototype there is also a Client for the Storage Manager. This client holds Limbo generated method stubs for the Storage Manager Device. Developers can use these by including the class torageManagerLimboClientPortImpl. To make it easier for Java developers to access the Storage Manager the StorageManagerDeviceClientBundle extends the automatically created client by the class HydraSMConnector.

Figure 36 above shows the API of the HydraSMConnector. It only holds the methods meant to be used by users of a Storage Managers Device, not the methods meant for communication between Storage Manager Devices.

HydraSMConnector (wsAddress : String) This constructor creates a new HydraSMConnector denoting to the given Storage Manager Device.
Using this class it is quite easy to access a Storage Manager Device in Java.

Listing 5.11.3: Example of a Java application using a Storage Manager Device

```
package de.douglas2a.hydra.testkram;
import java.io.IOException
import com.eu.hydra.limbo.storagemanagerdevice.client.
HydraSMConnector;
import com.eu.hydra.storage.helper.ErrorCodes;
import com.eu.hydra.storage.helper.StringVectorResponse;
public class StorageManagerTester {
 public static void main (String args[]) {
 String path = "http://localhost:8083/services/storagemanager";
 try {
       HydraSMConnector smClient = new HydraSMConnector(path);
       StringVectorResponse svr =
       smClient.getSupportedStorageDevices();
 if ( svr.getErrorCode()!= ErrorCodes.EC_NO_ERROR) {
       System.out.println(svr.errorOut());
 } else {

 if (svr.getResult().isEmpty()) {

 } else {
       System.out.println ("Supported Devices:");
 for ( Stringtype:svr.getResult()) {
       System.out.println("_" + type);
 }
 }
```

```
    }
    } catch (IOException e) {
         e.printStackTrace () ;
   }
    }
   }
```

5.11.5 Command Line Client

The StorageManagerClientCLI is a command line interface to access a Storage Manager Device. It can be used to maintain Storage Devices hosted by the Device. The eclipse project includes a description for exporting the command line client including all dependent libraries as jar file. A call of the command line client has to follow the form:

host> java -jar StorageManagerClientCLI.jar \\
SMD_WS_ADDRESS command (parameters ...)
SMD_WS_ADDRESS is the Web Service Address of the Storage Manager Device.

5.11.6 File System Devices

A File System Device is the representation of storage structured in files and directories in the Hydra Storage Architecture. There are different kinds of File System Devices available, which store the data in different ways, but all File System Devices share the same API. Therefore, from the developer's point of view, it does not matter which kind of File System Device is accessed.

5.11.7 API

The API of the File System Device is designed to be close to functions to access files and directories in operating systems. The File System Devices are accessed using a network and must be reachable using web services and UPnP. Therefore, it is impossible to use streams and all accesses to files have to be block oriented.

Listing 5.11.4: A HydraFile denoting a directory

```
<hydraFile
 path="/"
 isDirectory ="true "
 aTime="1234221"
 mTime="1205346"
 cTime="1131237"\>
```

Listing 5.11.5: A HydraFile denoting a file

```
<hydraFile
 path="/test.txt"
 isDirectory="false"
 aTime="1234201"
 mTime="1205002"
 cTime="1131248"
 size="4096">
 <properties>
<eu.com.hydra.storage.fsd.encoding.method>

 text
```

```
</eu.com.hydra.storage.fsd.encoding.method>
<mimeType>
Text:ascii
</mimeType>
</properties>
</hydraFile>
```

The HydraFile is an important part of the File System Device API. This data structure is used to exchange metadata information about a file or directory. It is implemented in the StorageManagerCommonBundle.

Listing 5.11.4 shows the XML representation of a directory as a HydraFile. Each file holds the path to the file or directory on the File System Device in the attribute path. The field isDirectory holds a boolean value which is true if path denotes a directory. If it is set to false, path links to a file, as the File System Devices only support files and directories.
Links, sockets, pipes, and other entities supported by some operating systems are not supported by File System Devices. The fields aTime, mTime and cTime hold the timestamps of the last access, last modification, and creation of the file or directory.

If the HydraFile denotes a file, it has some additional fields, like shown in Listing 3.5. The field size holds the size of the file in bytes. Files can also have properties in the Hydra Storage Architectures. These are submitted in the child element properties. The properties are stored as dictionaries. In Listing 5.11.5 line 9 holds the property eu.com.hydra.storage.fsd.encoding.method with the value text in line 10. This property has a special meaning. If it is set to base64 the File System Device will encode any data read from the file in base64 format. Any data written to the file is supposed to be in base64 format and will be decoded before writing. This way binary data can be submitted to the file. If the property is set to text, no conversion will be performed.

Another structure used by the File System Device is called StatFS. This data structure is used to give a client the most significant information about a File System Device with one method call. In the implemented prototype this method only delivers the size of the File System device, the free space, and the available space.

Figure 37 below shows the API of the File System Devices.

FileSystemDevice

clearFile(path : string) : VoidResponse
copy(sourcePath : string, destinationPath : string) : VoidResponse
createDirectory(path : string) : VoidResponse
createFile(path : string, properties : Dictionary) : VoidResponse
existsPath(path : string) : BooleanResponse
getDirectoryEntries(path : string) : HydraFileVectorResponse
getFile(path : string) : HydraFileResponse
getFreeSpace() : LongResponse
getId() : string
getSize() : LongResponse
getStatFs() : StatFSResponse
move(sourcePath : string, destinationPath : string) : VoidResponse
readFile(path : string, start : long, size : int) : StringResponse
removeDirectory(path : string, recursive : boolean) : VoidResponse
removeFile(path : string) : VoidResponse
setFileProperties(path : string, properties : Dictionary) : VoidResponse
setFileProperty(path : string, propertyName : string, propertyValue : string) : VoidResponse
truncateFile(path : string, size : long) : VoidResponse
writeFile(path : string, start : long, data : string) VoidResponse

Figure 37: API of a Hydra File System Device

5.11.8 File System Device types

A new different File System Devices has been developed to satisfy different storage demands in the prototype. Figure 38 below shows an overview over the different File System Devices.

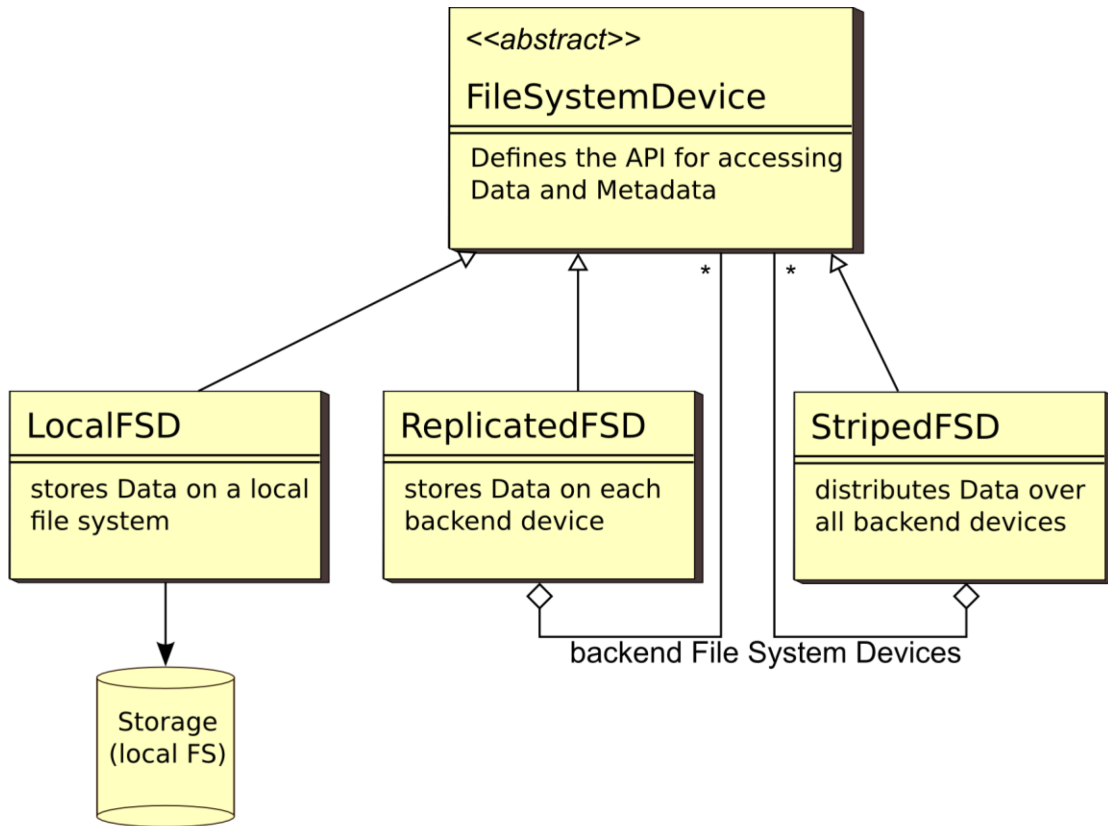

Figure 38: Basic architecture of file system

6. Device Development Kit

This chapter presents the Device Development Kit (DDK), and specifically the process of Hydra-enabling a device. The Hydra Middleware aims at developers who want to use network embedded devices and built applications on top of the the layer that communicates with these devices. There are a huge variety of network devices, with even ore on the horizon. These devices communicate with different protocols and standards. To make use of a unified method, the Hydra DDK provides the necessary functionalities to abstracts from their different levels and standards and makes it easier for application developer to communicate with different devices in the same way.

6.1 DDK Components and Tools

6.1.1 Limbo

This tutorial aims at giving Limbo users a guide for generating services using the Limbo compiler. The Limbo compiler is used to create Web Service interfaces for devices, in order to communicate and set the necessary functionalities to discover the device and its services within the Hydra Framework. For this purpose, a simple example of a thermometer service is used. The device used is a PICO TH03 thermometer, which is the same one used in the first Hydra prototype.

Two operations, getStatus and getTemperature, are defined in the service and they both take as argument a thermometerID which is a string that identifies a thermometer for the case of having more than one. This example has also been used to develop the state machine part of Limbo that will be explained further ahead in this document.

The thermometer service will provide the following functionality (written as Java code):

```
  public interface th03 {
   public boolean getStatus(String thermometerId);
   public double getTemperature(String thermometerId);
}
```

6.1.1 Obtaining and Installing Limbo

The prerequisites for running Limbo are:

- Java 5 or later
- the Hydra Event Manager (if using the state machine part of Limbo)

Once the limbo is downloaded, to install, it needs to be unzipped to an installation directory, called <limbo>. It is assumed that commands are invoked in the <limbo> directory.

Using Limbo

- Using Limbo includes the following steps:
- Describe targeted device in Hydra's device ontology (optional)
- Describe the service in a WSDL file. Reference the device description from the WSDL file
- Describe the service-related statemachine in the device ontology (optional)
- Run the Limbo compiler on the WSDL file
- Implement and deploy the device-specific service

- Run the service

Describing the device in the Hydra ontology

A device has basic device type information (modelled with Device.owl), its associated software platform (SoftwarePlatform.owl), hardware platform (Hardware.owl), and also a state machine to model the device state transitions at run time (StateMachine.owl). Therefore when adding a device, the related hardware and software, state machine information should be encoded in the related ontologies.

The Hydra ontologies may be found in <limbo>/resources/.

In the device ontology (Device.owl), the following code is added for an indoor thermometer (the model number is pico th03):

```
<InfoDescription rdf:ID="PicoTh03_info">
    <modelDescription
rdf:datatype="http://www.w3.org/2001/XMLSchema#string">3
channels</modelDescription>
    <manufacturerURL
rdf:datatype="http://www.w3.org/2001/XMLSchema#string">www.picotech.com</m
anufacturerURL>
    <friendlyName
rdf:datatype="http://www.w3.org/2001/XMLSchema#string">PicoTh03</friendlyN
ame>
    <modelName
rdf:datatype="http://www.w3.org/2001/XMLSchema#string">th03</modelName>
    <manufacturer
rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Pico technology
limited</manufacturer>
    <modelNumber
rdf:datatype="http://www.w3.org/2001/XMLSchema#string">th03</modelNumber>
 </InfoDescription>
<Thermometer rdf:ID="PicoTh03_Indoor">
  <deviceId rdf:datatype="&xsd;string">PicoTh03_Indoor</deviceId>
  <hasHarware rdf:resource="&Hardware;PicoTh03_hardware"/>
  <hasStateMachine rdf:resource="&state;PIcoTh03_Indoor_sm"/>
</Thermometer>
```

The hardware information is added in the hardware ontology (Hardware.owl):

```
<DeviceHardware rdf:ID="PicoTh03_hardware">
  <primaryCPU>
    <CPU rdf:ID="PIC16C54C">
      <cpuName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
        PIC16C54C</cpuName>
    </CPU>
  </primaryCPU>
</DeviceHardware>
```

The software information can be left empty as this thermometer does not have a software platform that supports web service deployment.

6.1.2 Describing the service in a WSDL file

In this case, a simple WSDL file is used for a thermometer service that contains two operations, one for getStatus and another for getTemperature. The WSDL file, which is also in <limbo>/tutorial/wsdl/, is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
 xmlns:ns="http://hydra.eu.com/th03/"
targetNamespace="http://hydra.eu.com/th03/"
xmlns:hydra="http://hydra.eu.com/" >
      <message name="thermResponseDouble">
            <part name="result" type="xs:double"/>
      </message>
      <message name="thermRequest">
            <part name="thermometerId" type="xs:string"/>
      </message>
      <message name="thermResponseBoolean">
            <part name="status" type="xs:boolean"/>
      </message>
      <message name="thermRequest1">
            <part name="thermometerId1" type="xs:string"/>
      </message>
      <portType name="TH03Port">
            <operation name="getTemperature">
                  <input message="ns:thermRequest" name="thermRequest"/>
                  <output message="ns:thermResponseDouble"
name="thermResponseDouble"/>
            </operation>
            <operation name="getStatus">
                  <input message="ns:thermRequest1" name="thermRequest1"/>
                  <output message="ns:thermResponseBoolean"
name="thermResponseBoolean"/>
            </operation>
      </portType>
      <binding name="TH03SOAP" type="ns:TH03Port">
            <hydra:binding
device="file:./resources/Device.owl#PicoTh03_Indoor"/>
            <soap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
            <operation name="getTemperature">
                  <soap:operation
soapAction="http://hydra.eu.com/th03/getTemperature" style="rpc"/>
                  <input name="thermRequest">
                        <soap:body use="literal"
namespace="http://hydra.eu.com/"/>
                  </input>
                  <output name="thermResponseDouble">
                        <soap:body use="literal"
namespace="http://hydra.eu.com/"/>
                  </output>
            </operation>
            <operation name="getStatus">
                  <soap:operation
soapAction="http://hydra.eu.com/th03/getStatus" style="rpc"/>
                  <input name="thermRequest1">
                        <soap:body use="literal"
namespace="http://hydra.eu.com/"/>
                  </input>
                  <output name="thermResponseBoolean">
                        <soap:body use="literal"
namespace="http://hydra.eu.com/"/>
                  </output>
```

```
            </operation>
      </binding>
      <service name="TH03Service">
            <port name="TH03Service" binding="ns:TH03SOAP">
                  <soap:address location="http://dmz-
168.daimi.au.dk:8084/th03"/>
            </port>
      </service>
</definitions>
```

6.1.3 Describing the service-related statemachine

For the current implementation of state machine stub code generation, a dummy state machine instance is needed for one type of devices. For example, the thermometer has a dummy generic state machine as shown in the following figure called Thermometer_sm, and then there is a state machine instance for every device, for example PicoTh03_indoor_sm.

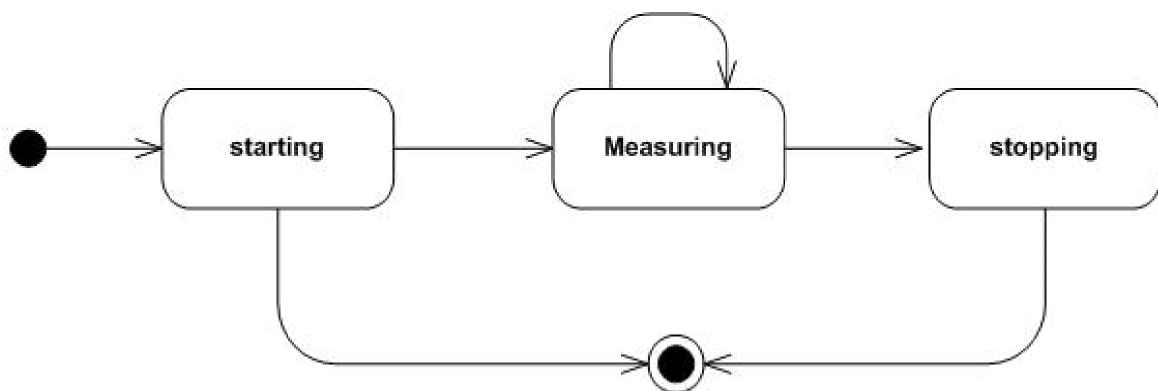

Figure 39: Dummy state machine

The dummy state machine Thermometer_sm can be added to the state machine ontology as follows:

Only State instances are used to generate code (and their related doActivity).

```
<StateMachine rdf:ID="Thermometer_Indoor_sm">
   <hasStates rdf:resource="#ThermometerStopping"/>
   <hasStates rdf:resource="#ThermometerStarting"/>
   <hasStates rdf:resource="#ThermometerMeauring"/>
</StateMachine>
<Simple rdf:ID="ThermometerMeauring">
   <StateName rdf:datatype="&xsd;string">
      ThermometerMeauring</StateName>
   <doActivity rdf:resource="#getTemperature"/>
</Simple>
<Action rdf:ID="ThermometerStart"/>
<Simple rdf:ID="ThermometerStarting">
   <StateName rdf:datatype="&xsd;string">
      ThermometerStarting</StateName>
   <doActivity rdf:resource="#ThermometerStart"/>
</Simple>
<Action rdf:ID="ThermometerStop"/>
<Simple rdf:ID="ThermometerStopping">
   <StateName rdf:datatype="&xsd;string">
      ThermometerStopping</StateName>
   <doActivity rdf:resource="#ThermometerStop"/>
</Simple>
```

6.1.4 Run the Limbo compiler on the WSDL file

In <limbo>, invoke:

```
java -jar limbo.jar <arguments> tutorial/wsdl/th03r.wsdl
```

where <arguments> is on the form "-<argument> <value>". For the tutorial, the argument list is left empty.

The following arguments are supported:

Option: name	Option: values	Default	Meaning
limbo.language	jse, jme	jse	Determines which programming language code is generated for
limbo.platform	standalone,osgi	standalone	The target platform. In the osgi case, the standard HTTP service will be used, in the standalone case, Limbo will generate a simple HTTP server
limbo.generationtype	server,client,all	all	Determines whether a skeleton is created for a server, a stub is created for a client, or both
limbo.protocol	TCP,UDP,BT	TCP	Determines which transport layer protocol will be used: TCP, UDP, or Bluetooth (RFCOMM)
limbo.loghandler	true,false	false	If set to true, the generated server code will log requests
limbo.outputdirectory	any directory	generated	Specifies where generated code is put

Per default, Eclipse project resources are created so the generated code may be used as the basis of a project in Eclipse.

The following files are the most important files that are generated for the thermometer code in the case a standalone server project:

- TH03PortOpsImpl - A default implementation of the service methods
- LimboMain - A main program that will run the server created
- TH03PortLimboServer - A TH03-specific web server

In the OSGi configuration an Activator is generated instead of a main program (and a Servlet is created instead of using a LimboServer).

6.1.5 Implement and deploy the device-specific service

The generation created two directories in generated:

- th03rClient
- th03rServer

Each of these contains a project that can be imported in Eclipse. The projects are self-contained and may be copied to one's workspace.

The actual device binding is implemented in the com.eu.hydra.limbo.TH03PortOpsImpl class. Here the getStatus and getTemperature methods are implemented. In the following, just a dummy implementation is shown:

```
/**
 * getStatus method - returns the current staus of the thermometer.
 *
 * @param thermometerId1 - ID of the thermometer that status is
required.
 *
 * @return the status of the thermometer.
 *
 */
public boolean getStatus(String thermometerId1 ) {
  return true;
}

/**
 * getTemperature method - the current temperature measured by the
thermometer.
 *
 * @param thermometerId - ID of the thermometer that temperature is
requested.
 *
 * @return the temperature given by the thermometer.
 *
 */
public double getTemperature(String thermometerId ) {
  return -1.0;
}
```

6.1.6 Running the Generated Code

To run the server, run the com.eu.hydra.limbo.LimboMain class from Eclipse.

To interact with the server, locate the com.eu.hydra.limbo.client.TH03PortLimboClient class in the th03rClient project and replace /*Insert method calls here*/ with calls to the thermometer service. An example would be:

System.out.println("The temperature is " + theClient.getTemperature("42"));

Thereafter, the TH03PortLimboClient should be run in Eclipse. To change the default behaviour of the generated server skeleton, change the com.eu.hydra.limbo.TH03PortOpsImpl class as described above.

6.2 Device Ontology

The Device Ontology tools interface is a tool for manipulation of the device ontology via the web. It can be used by the device developer for defining a new device description or editing an existing device description. It can be used to browse devices in the ontology as well.

The Device Ontology tools interface can be used as a flexible ontology browser from the point of view of the device taxonomy. A user can navigate in the device classification (ontology concepts) and browse the device instances (concrete instance of some ontology concept, e.g.: HTCP3300 phone is an instance of MobilePhone concept). The user can view the instance properties. An example of the browser view is in Figure below. The device browser enables to navigate through any

part of the ontology. It is possible to browse through the hierarchy of sub-concepts of the HydraDevice concept. For each instance it is possible to browse through all of the instance properties and related property values. If the property values refer to another instance, the device browser enables us to recursively traverse through the properties of any connected instance. Furthermore, once an instance is selected, the browser shows all properties and connected values in the right part of the page. Using the device browser, it is easy to effectively navigate through all ontology concepts and the populated instances.

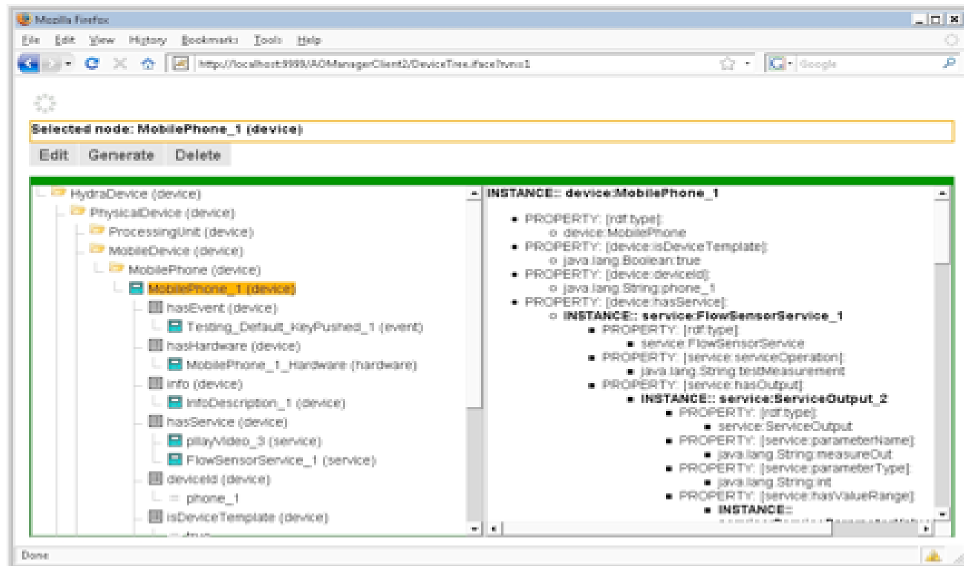

Figure 40: The Device Browser tab.

6.2.1 Device Creator

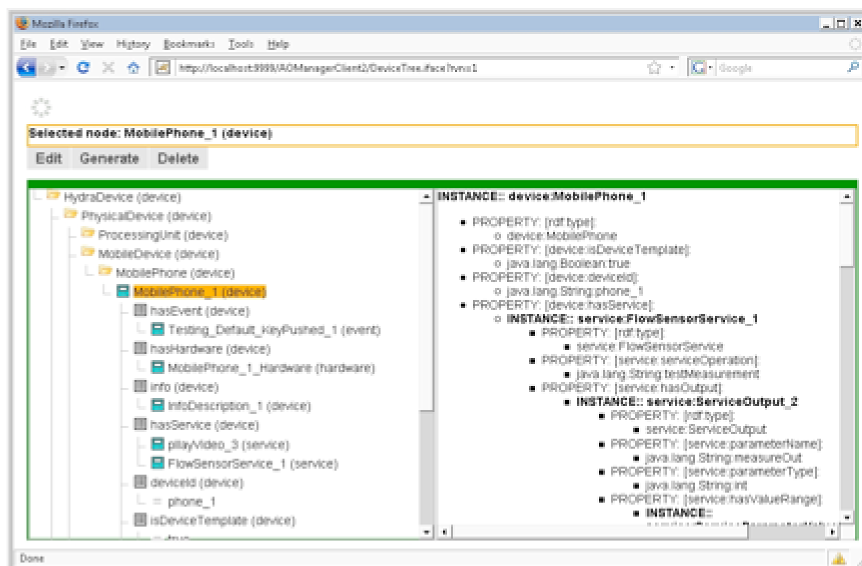

Figure 41: Adding a new instance.

The tool enables a user to create new devices and manually update some of the device properties. The device creator provides the following functionality:

Manually create new device enables to add the device into a selected device type (adding a new instance to the selected concept) by filling in the new device properties. See Figure above for the new device form.

Upload/Update SCPD (Service Control Point Description) info The device can be created semi automatically by uploading the specific XML document containing all device relevant information, including basic device description, specification of services, but also models for events and energy profile information. The content of XML can be generated automatically by several tools, but can be extended and further specified also manually. It is also necessary to be aware of the update XML structure.

Update Discovery info This functionality just simulates the semantic discovery process (normally performed by the Discovery Managers in the Hydra architecture) and should be used only for verification purposes. If a user fills-in the text box with the device discovery information formulated as XML with predefined structure, the tool will perform the semantic discovery matching and create the device run-time instance in the case of success. The tools supporting the device discovery process will be described in the next section with more details.

Update (SA)WSDL info This tool enables the user to automatically generate device service models from WSDL or SAWSDL files (i.e., Semantically Annotated WSDLs). The existing device instance has to be selected and the URL address, or a local file containing the (SA) WSDL file has to be specified. Services of the device will be substituted by a service description contained provided in the (SA) WSDL file. This tool will be described in the next section in more detailed way.

Update Malfunctions, Events and Energy Profile info Various parts of device models can be updated manually by providing specific XML documents describing the particular models. XML documents have a prescribed structure. This way, the basic device information, models for malfunctions, events and energy profiles can be updated. The illustration example of editing capabilities for the selected device is in Figure below:

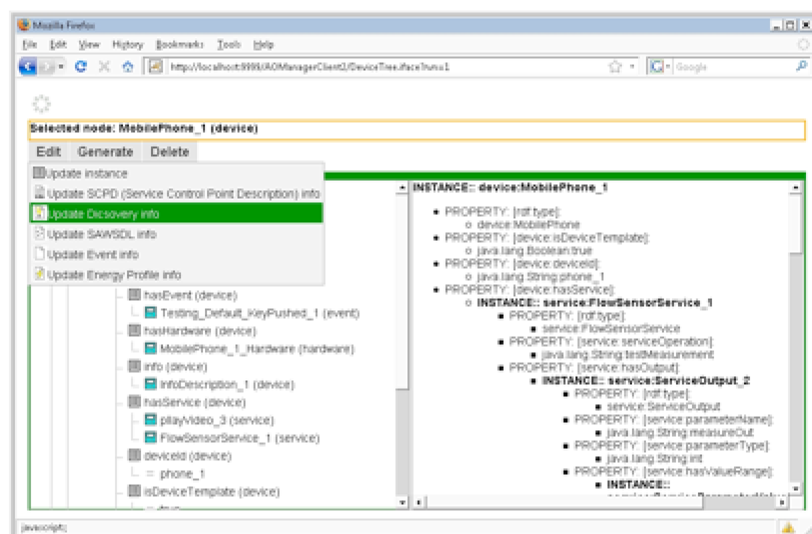

Figure 42: Device editing functionality

Import/Export functions

For the purposes of easy and effective device ontology population and device instances update/maintenance, various import/export generators were developed. The generators should be used both by application and device developers. Generators are also used by the ontology manager web interface. The generators consume device descriptions, from which are generated the related parts of ontology models, or, ontology models can be used to generate specific descriptive information. All of the generators are used to create ontology models/descriptions automatically rather than manually.

(SA)WSDL to ontology generator

A developer may populate the device service ontology with the description of services contained in a related WSDL file. For a selected device, the WSDL file is parsed and the device instance is extended

with the models of services specified in the WSDL file. Each time a new WSDL file is processed by the generator, the device services are completely replaced by the services in the newly provided WSDL. A WSDL file can be also semantically annotated using the SAWSDL standard. This functionality can be particularly useful in order for Device Developers to facilitate Hydra enabling of a device. In the actual implementation, the following annotations can be used:

Annotations for wsdl:operation element each WSDL operation stands for one specific service ontology model using sawsdl:modelReference attribute directly in the wsdl:operation element. This annotation is used to link the operation with the specific service type. Each operation should be mapped only to one service type.

Extending wsdl operation element with sawsdl:attrExtension annotations: to enable the annotations of service to various quality properties, such as Quality of Service or security properties, the sawsdl:attrExtension elements are used. Each of them can map the service to a specified ontology instance representing the property of service.

Annotation of input/output parameters to ontology instances each input or output parameter may be annotated to the specific part of the Quality of Service ontology, which represents the parameter types. Each parameter may be linked to the specific quality, e.g. output parameters can be annotated to the measurement units. The generator parses the SAWSDL file and generates the service models exactly as in the case of WSDL processing. Further more, the annotations are used to create the relations between particular service parts and the annotated ontology concepts or instances.

Discovery information to ontology generator

For the purposes of discovery process improvement, the developer may use a tool to generate the device discovery ontology model directly from the discovery information acquired from the device by one of the discovery managers. The device specific discovery information is parsed and the related device instance is extended by the ontology model modelling the discovery information. This model will be used in the semantic discovery process. The device discovery information acquired by a discovery manager is specified as an XML file, which is parsed by the model generator. Generation of discovery models should be realised for each device model newly added to the ontology. The model generator makes it possible to process and update the discovery model in a fast automatic way.

SPCD to ontology generator

For the purposes of adding or updating the devices model fast, the generic ontology generator consuming SPCD XML documents, was implemented. The SPCD contains several device relevant information, such as basic device description and manufacturer information, initial description of services, but also discovery, events and energy profiles information. The SPCD generator uses the specific ontology generators mentioned above processing the (SA) WSDL documents, but also discovery information.

Ontology model to SCPD file generator

Another tool supporting the automatic semantic discovery of devices is the so called SCPD generator. Each time, when a new device is discovered by some low-level discovery manager, the discovery information is retrieved from the device and passed to the ontology manager which tries to process the semantic resolution of the discovered device. This is done by comparing of the discovery information with the various discovery models in the ontology. If semantic discovery is successful and a suitable semantic model according to discovery information is found, the ontology manager returns the description of the identified device and the services, which are provided. This is done by generating so called SCPD file containing all the mentioned information. The SCPD file is generated directly from the related device and service models in ontology.

6.3 Flamenco

Flamenco is a tool for supporting self-management in Hydra-based systems. In fact, Flamenco is a way to model the desired behaviour of a device as a Coloured Petri Net (CPN), to track it at runtime and to emit events for certain state transitions. It currently exists in two versions:

- Flamenco/CPN in which Petri Nets is used as a basis
- Flamenco/SW in which Semantic Web technologies are used as a basis

In the following the users are guided through a simple example of self-management and how to use Flamenco to realise a scenario of managing a flow meter-based agricultural system. Furthermore, an explanation is provided on how to make use of Flamenco to choose optimised solutions according to multiple (conflicting) objectives, for example QoS requirements on memory consumption, throughput, reliability and so on.

6.3.1 System Requirements and Installation

- Windows XP or Vista (CPN Tools only runs on Windows or Linux)
- Java 5 or later
- CPN Tools. A license and download the tool can be requested from: http://wiki.daimi.au.dk/cpntools/cpntools.wiki. Version 2.3.5 which is an internal version is needed. The CPN Tools people or UAAR need to be contacted for instructions on how to download.
- Access to the Hydra Middleware software
- Eclipse Europa or later for running Flamenco

6.3.2 Design Time Usage

CPN Tools may be used directly to Flamenco/CPN nets. There is a template for such nets available in HYDRA/sdk/flamenco/flamencocpn/resources/cpn/flamenco-template.cpn. The figure below shows the result of opening the template:

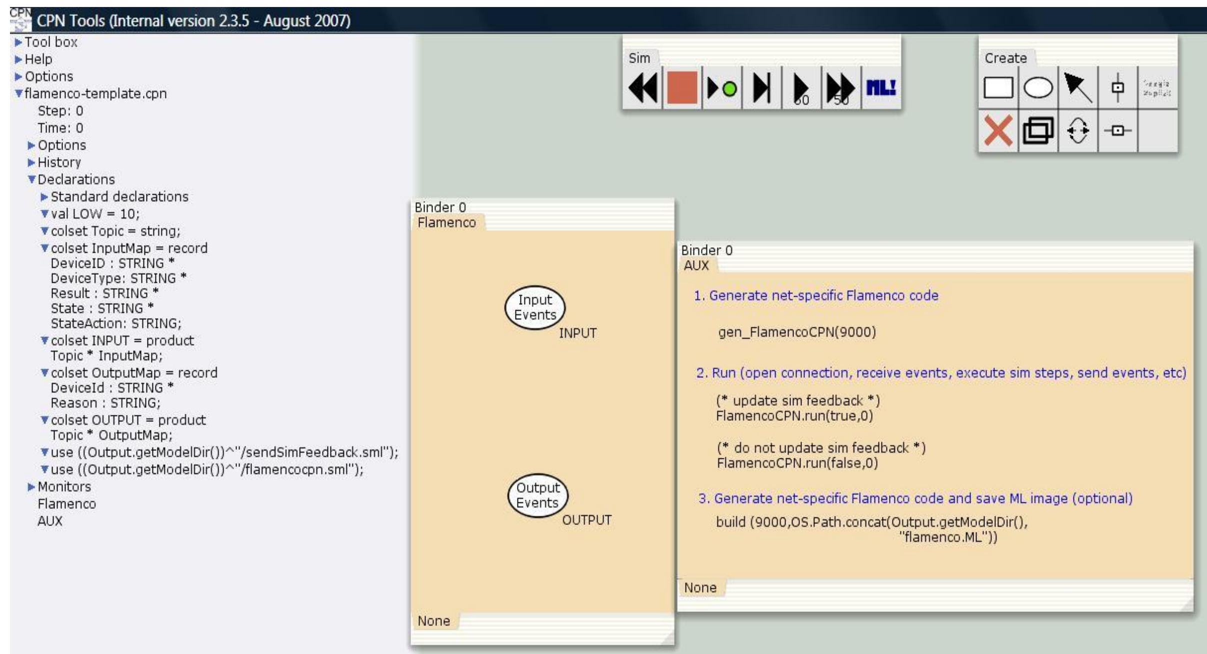

Figure 43: CPN Tools

6.3.3 The auxiliary page

The right hand side is an auxiliary page that is used to generate specific Standard ML code for a Flamenco/CPN net. When the net is changed to use it at runtime in Flamenco/CPN, the first expression needs to be evaluated. Evaluating one of the expressions under "2." will start CPN Tools and wait for an attachment from the Java part of Flamenco/CPN on port 9000. Depending on which one you choose, you will be able to see the net being updated or not while Flamenco/CPN runs.

Lastly, the third expression may be evaluated to run Flamenco/CPN entirely without a user interface. Evaluating the expression will generate a Standard ML image that contains the specific net.

6.3.4 The net

The template net is shown in the middle. It only contains two template places ("Input Events" and "Output Events"). These places will receive and send events from the Hydra middleware respectively at runtime. In general there should be one place with the colour INPUT and one place with the colour OUTPUT.

6.3.5 The declarations

The declarations to the right define the colour of the input and output events and should be extended as needed.

Runtime Usage

The net for a full Flamenco/CPN example is shown in the figure below:

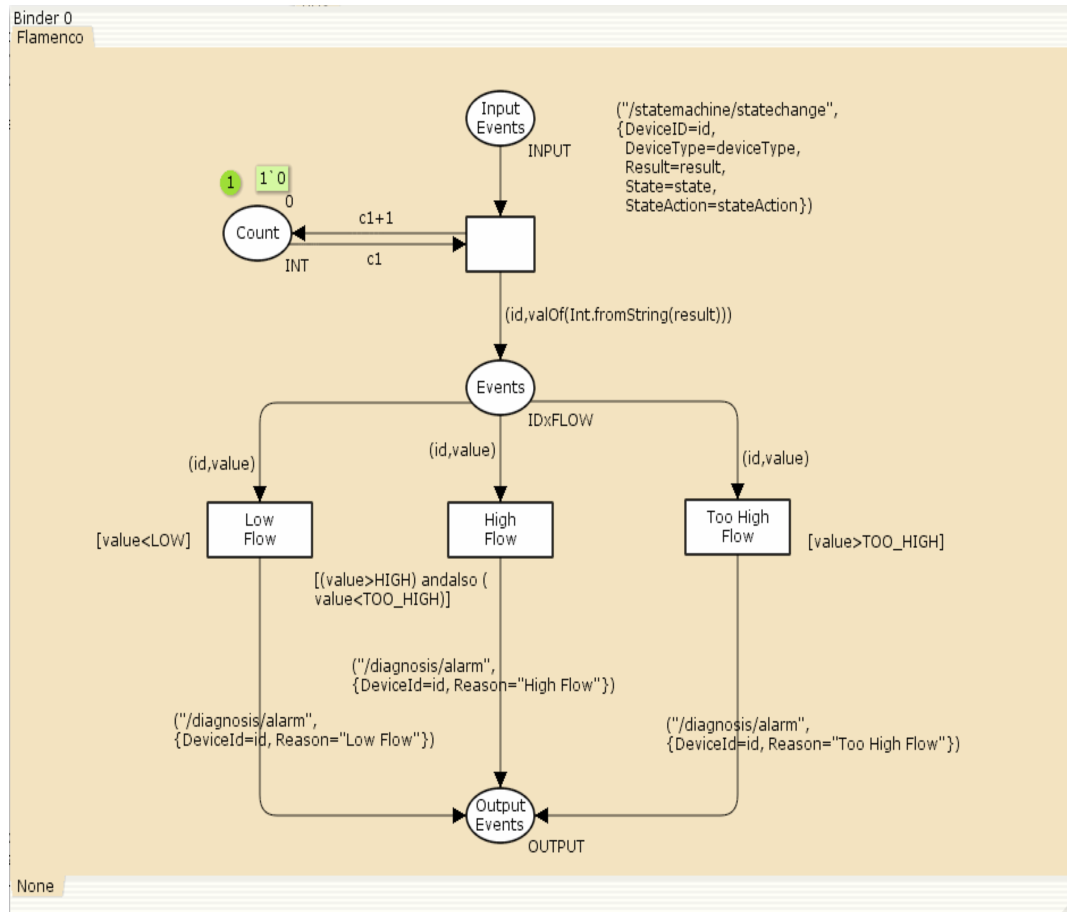

Figure 44: Flamenco

To run Flamenco/CPN, the following steps need to be taken:

- open CPN Tools on a Flamenco/CPN net as described in the previous section and evaluate the appropriate auxiliary declarations
- run the Hydra Event Manager
- run the Java part of Flamenco/CPN. The easiest way currently is to start Eclipse on the FlamencoCPN project and then run com.eu.hydra.flamenco.cpn.Flamenco.
- run a number of devices that will produce events

For simulating the last step, the FlamencoTest project can be used. The class com.eu.hydra.flamenco. cpn.FlowTester will produce events from the flow meter scenario. If the CPN Tools are started with user interface updates, the tokens being produced and consumed in the net will be seen.

6.3.6 Flamenco/SW

System Requirements

Currently the OWL/SWRL based Diagnosis manager is tested on Windows Vista and Java 6, but it should run on any operating system with Java 5 or later.

Below is the list of tools needed for running it:

- Tomcat 5 or later
- Protege 3.4 build 130 or later

Installation

- Install Tomcat. Change the HTTP port to 9999, create a directory called 'ontology' under the directory 'webapps', Tomcat can be download from this link (version 5.5): http://tomcat.apache.org/download-55.cgi
- Install Protege. The current SWRL APIs needs to access the ontologies coming with Protege, therefore, the running of the OWL/SWRL based Diagnosis Manager needs to point to the Protege installing directory. This is not necessary since protege 3.4 version 500, as the SWRL related ontologies can be accessed directly through the internet. However this is still recommended to improve performance for starting the Flamenco. Protege can be downloaded from this link: http://protege.stanford.edu/download/registered.html
- Download the Flamenco/SW. All ontologies are located in \ontodiagnosis\resources directory.
- Copy all ontologies (including rule ontologies) to the newly created 'ontology' directory. Thus, all the rules and ontologies are ready for use.
- Install the testing client.

Usage

Flamenco/SW listens to topic of '/statemachine/statechange', '/flamenco/socketwatch'. Therefore to get a diagnosis of a system/application/device, events on these topics must be published and of course there should be a state machine corresponding to a device in oder to be diagnosed. Another issue is that the Flamenco/SW should be subscribing to the same Event manager as the one that publishes events, in order to make use of the Network manager and Trust manager functionalities.

The following steps need to be taken:

- Start Tomcat.
- Check that the Event manager is running (for the moment it is using EventManager_CNET), otherwise start the Event manager.
- Start Network manager
- Change the build file of Flamenco/SW. Only this tag in the build file: <jvmarg value="-Dprotege.dir=c:/protege/3"/> need to be changed to the Protege installation directory. After this start the diagnosis manager with ant build. Alternatively, Flamenco/SW can be started by running as Java application by click on class
- Start one of the test clients. In this tutorial the flowmeter client is used: Build it with the ant build file in order to create a jar file, called Flowmeter.jar. Copy the Flowmeter.jar to Resource manager lib directory, and then change the config.ini under the lib\configuration as follows:

```
osgi.bundles = \
../lib/org.eclipse.equinox.log_1.0.1.R32x_v20060717.jar@2:start, \
../lib/org.eclipse.equinox.common_3.2.0.v20060603.jar@2:start, \
../lib/org.eclipse.osgi.services_3.1.100.v20060601.jar@2:start, \
../lib/javax.servlet_2.4.0.v200706061611.jar@3:start, \
../lib/org.eclipse.equinox.http_1.0.2.R32x_v20061218.jar@3:start, \
../lib/Flowmeter.jar@4:start
```

  The last line is used to start the flowmeter test client. The client is started by simply running it as a java application. It can be seen that the client is sending measurements, and when the event manager publishes the state changes, the diagnosis manager will conduct a diagnosis based on the changed states, and publish it.

- The thermometer scenario can be used by the thermometer client. The config.ini needs to be changed and change Flowmeter.jar to Thermometer.jar (the jar name built from thermometer client).

One thing to note is that in order to test multiple times using ant build file coming with Flamenco/SW, Java process may need to be killed with task manager (this problem can be particularly experienced in windows Vista/XP), remember to leave the one for Tomcat5.5, which is usually using around 45-50M memory.

Development

There may be two kinds of developers who can utilize the Flamenco/SW diagnosis manager SDK, knowledge developer, who is responsible for the development of rules and the addition of diagnosis cases based on the existing ontologies, and Java application developer who make use of the rules and ontologies for realising the diagnosis.

For knowledge developers, and most probably they are the developers who need use the SW diagnosis manager SDK:

To use the SW for your own development, the simplest case is to add a device to an existing system. Please use the ontologies as the starting point.

The first thing needed is to add this device instance to the Device ontology, and then add this device instance to the HydraSystem concept in the Device ontology, which only needs to add related diagnosis rule and the device state machine. For example the steps for adding a flow meter to the Pig system in agriculture domain are:

1. Add the flowmeter device to the Pig system concept in the Device ontology, as shown in the following figure.

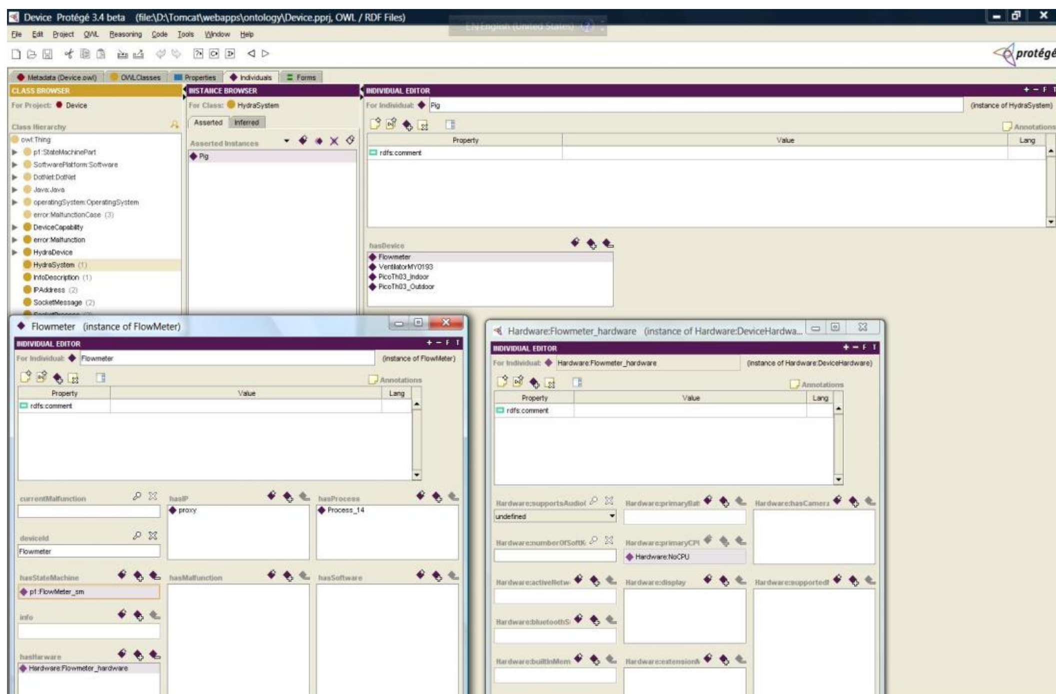

Figure 45: Device Protégé

2. Add the flowmeter state machine instance to the StateMachine ontology. It is called "Flowmeter_sm" in our case, if it does not exist.
3. Add the flowmeter state machine instance to the hasStateMachine property of the "Flowmeter" device.

4. Add flowmeter diagnosis rule to the DeviceRule ontology, for example, one rule to diagnosis flowmeter is:

   device:FlowMeter(?device) ?
   device:hasStateMachine(?device, ?statemachine) ?
   statemachine:hasStates(?statemachine, ?state) ?
   statemachine:doActivity(?state, ?action) ?
   statemachine:actionResult(?action, ?result) ?
   statemachine:historicalResult1(?action, ?result1) ?
   statemachine:historicalResult2(?action, ?result2) ?
   statemachine:historicalResult3(?action, ?result3) ?
   swrlb:add(?temp, ?result1, ?result2, ?result3) ?
   swrlb:divide(?average, ?temp, 3) ?
   swrlb:subtract(?diff, ?result, ?average) ?
   swrlb:abs(?absdiff, ?diff) ?
   swrlb:greaterThan(?absdiff, 6.0)
   ? sqwrl:select(?device, ?statemachine, ?state, ?action, ?average, ?result, ?diff)

The adding of rules can be facilitated by using the SWRL tab in the Protege tool as shown in the following figure:

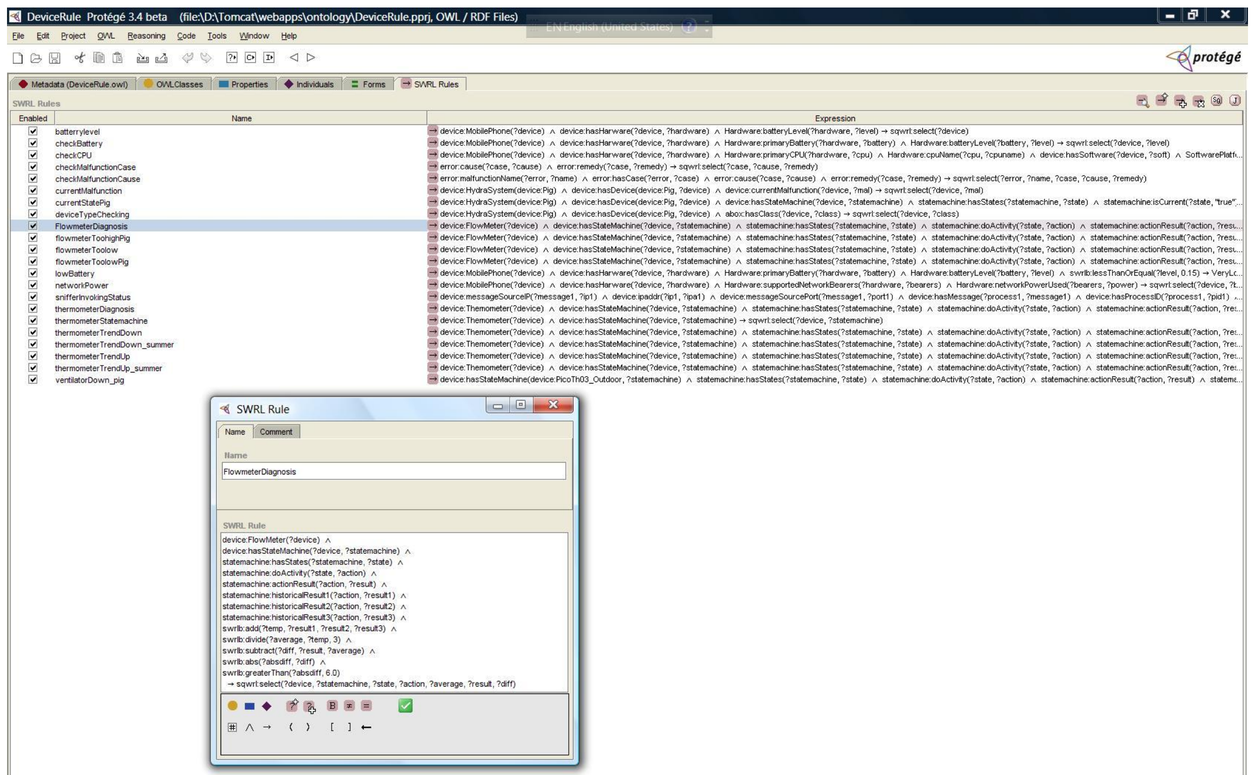

Figure 46: Device Rules Protégé

5. Add diagnosis case to the Malfunction ontology. For example, the "flowToohigh" instance can be added to the "DeviceError" concept, with the "pipeBroken" as the case for the "hasCase" property by clicking the "Add new resource" button, and then fill the "pipeBroken" by adding "cause" as "pipe broken" and "remedy" as "replace pipe".

Java application developer:

Suppose the rules added by the knowledge developer are only related to one device. Then there is no need to do anything as the APIs can handle the diagnosis cases.

In the case that a developer needs to process a rule, then a key class to use is RuleProcessing in package com.eu.hydra.flamenco.ruleprocessing. It can be used like this:

```
RuleProcessing rp=new
RuleProcessing("http://localhost:9999/ontology/DeviceRule.owl");
HashSet<String> set=a.getAllSWRLInferred(); // get all inferred
information, and can get inferred
                                             // individual or property
separately using
                                             //
getSWRLInferredIndividual(), getSWRLInferredProperty().

rp.checkNormalTwoColumnRule("deviceTypeChecking"); // execute rule called
"deviceTypeChecking"
```

There are different methods for processing different types of rules: checkSingleColumnRule() which is used to process a rule returns only one column result but may have multiple rows. Similarly there are other rules processing methods.

As there may be many rules, but different rules are used for different purpose, therefore, a separate rule group can be built and executed as needed. The rule group feature can be used like this:

```
RuleGroupProcessing a=new
RuleGroupProcessing("http://localhost:9999/ontology/DeviceRule.owl");
a.processRuleGroup("pig"); //create a rule group called "pig"
HashSet<String> set=a.processRuleGroup("pig"); //This will execute all
rules whose name contains 'Pig'
HashSet<String> set1=a.processRuleGroup("pig", "battery", "and"); //This
will execute all rules whose name contains 'Pig' and 'battery'.
HashSet<String> set2=a.processRuleGroup("pig", "battery", "or"); //This
will execute all rules whose name contains 'Pig' or 'battery.
```

Now the rule grouping feature can be used to diagnosis as followed:

```
DiagnosisInitializingData.getDiagnosisInitializingDataInstance();
DiagnosisInitiation pig=DiagnosisInitiation.getPigRuleInstance();
 //prepare for infered result parsing as a observer to InferredResult
InferredResultParsing
parser=InferredResultParsing.getInferredResultParsingInstance();
InferredResult  result=InferredResult.getInferredResultInstance();
result.addObserver(parser);
pig.Diagnosis("pig");
pig.Diagnosis("ventilator");
pig.Diagnosis("flowmeter");
```

Planning in Flamenco

Flamenco is adopting a 3Layered self-management approach in which there is layer responsible for planning. The planning layer is using Genetic Algorithms (GAs) to find optimised solutions for a problem, based on the JMetal GA framework. Currently the planning layer supports three GAs, NSGA-II, FastGPA or MOCell.

Usage

The planning layer is part of the Flamenco, but can be used separately also. There are some existing applications (e.g. self-adaption which reads component QoS properties from file, and self-protection which reads QoS properties for security protocols from security ontologies). Here the self-adaption example is used to illustrate how to run the planning layer. From the tests/evaluations, it was concluded that NSGA-II is the best GA for self-management problem, in which usually, the NSGA-II genetic algorithm will find a number of optimised solutions, and its corresponding variables.

Development

The overall architecture of the planning layer is shown in the following figure:

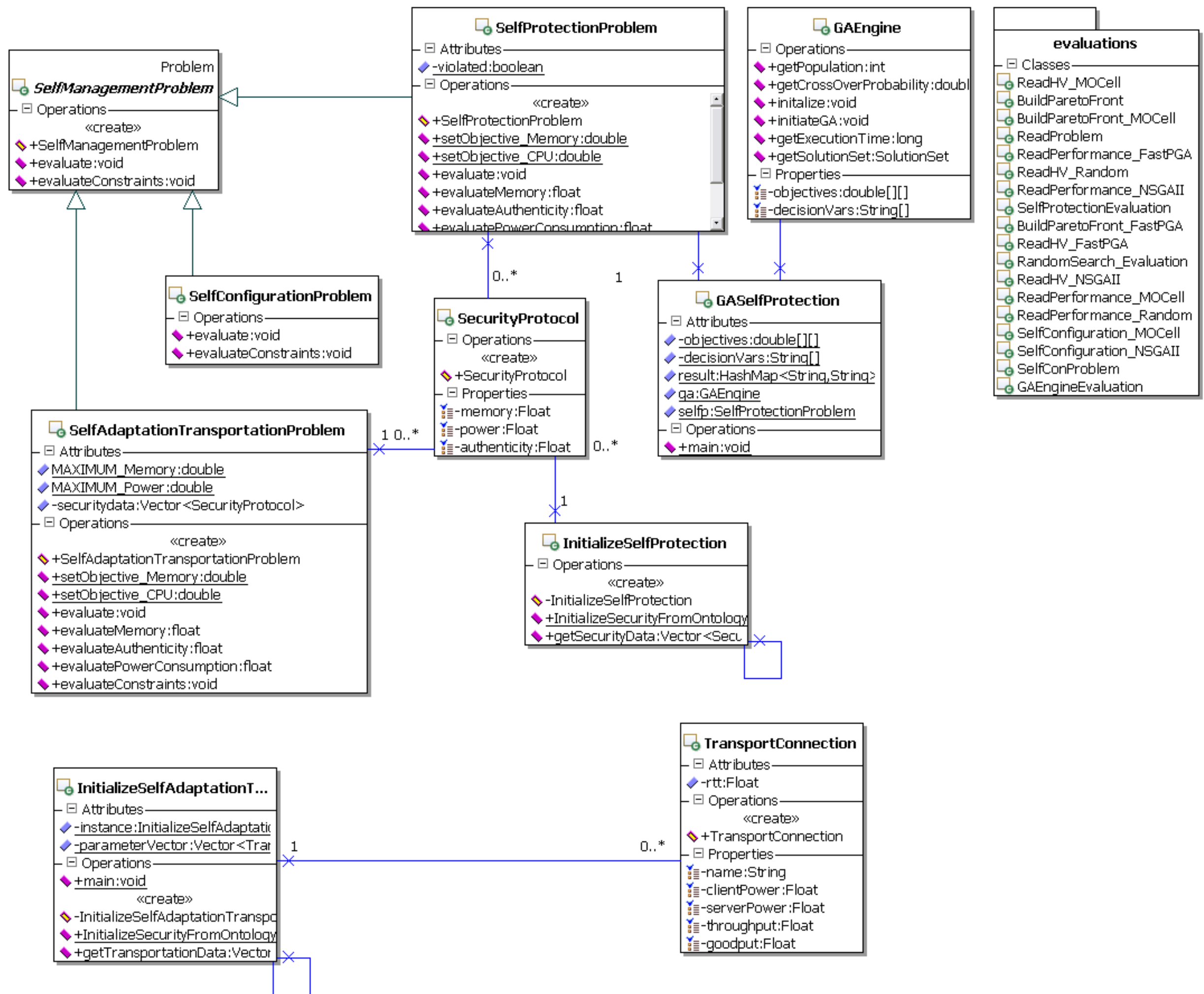

Figure 47: Flamenco Planning Layer

Chromosome encoding and fitness evaluations

The representation of chromosome in this case is using integer (starting from 0). That is to say, an integer vector V =[V 1; V 2; :::V i; :::; V n] (where n is the number of decision variables, and in our case, it is 10) is used to represent a solution. V i is a natural number, acts as a pointer to the sequence of the concrete implementation of the ith services. For example, a chromosome [0,1,3,3,2,0,1,1,2,3] represents that a solution chooses the first implementation of service number 1, chooses the second implementation of service number 2, chooses the fourth implementation of service number 3, and so on. In this case, it chooses AxisSENM, AxisMECM, LimboMEDM, and so on. Based on the chosen components, the GAs then decide its fitness using the objective equation, and will evaluate whether the constraints are meet at the same time.

Define optimisation problem

A number of steps are needed to abstract an optimization problem:

1. Define the problem class (e.g. SelfConfigurationProblem), which should extend the SelfManagement Problem, which extends JMetal:Problem interface.
2. Define the methods for evaluating fitness of a solution, which is defined also in SelfConfiguration Problem class.
3. Define the methods for evaluating constrains of the problem in the SelfConfigurationProblem class.
4. Define another class TestSelfConfig, which is the entry point for initiating the GAEngine, choosing either NSGA-II, FastGPA or MOCell, to operate on the defined problem, and then the solutions and their corresponding values for decision variables (i.e., the number of a concrete components) can be obtained.

6.4 Device Discovery Manager

The Device Discovery Manager is used to detect devices in the Hdyra network and collect information about its functionalities and how this functionalities can be addressed by application and other devices. It also makes use of descriptions of devices and the Ontology Framework.

DDK Class library for .NET
A part of the Hydra DDK is a .net based class library available for device developers and device manufacturers. The DDK documentation is divided into 4 parts: Device Service Managers, Discovery Managers, Hydra Device Manager, Device Device Managers.

6.3.4.1 Device Service Managers
This documents the different device service managers that exist and can be used to create a Hydra Device for interfacing with for instance Bluetooth devices.

6.3.4.2 Hydra Device Manager
A Device Service Manager and a Device Device Manager is needed to create a Hydra Device. The Hydra Device Manager is the base class for all Device Managers in Hydra. A device developer can inherited from this class for creating their own device manager.

6.3.4.3 Device Device Manager
This is the documentation of a number of already existing device managers that a developer can choose from or make use of to derive own functionality:

WeatherSensorDevice::AirPressureDevice
BasicPhoneDevice::BasicPhoneDevice
SwitchDevice::BasicSwitchDevice

Hydra::BlindDevice
BlueToothDevice::BlueToothDevice
SwitchDevice::EnhancedSwitchDevice
SwitchDevice::ExternalSwitchDevice
Hydra::GPSDevice
Hydra::MediaRendererDevice
Hydra::MedicalDevice
WeatherSensorDevice::RainSensorDevice
RFIDTagDevice::RFIDTagDevice
RemoteUPnPDevice::RemoteUPnPDevice
RemoteWSDevice::RemoteWSDevice
RFIDTagDevice::RFIDTagDevice
SmartPhoneDevice::SmartPhoneDevice
WeatherSensorDevice::ThermometerDevice
WeatherSensorDevice::WeatherSensorDevice
WeatherSensorDevice::WindmeterDevice
Hydra::ZigBeeCoordinatorDevice
Hydra::ZigBeeEndDevice

6.3.4.4 Discovery Managers

A device developer creating his own Hydra Device might also need to provide a specific Hydra Discovery Manager to allow the device to be discovered in a Hydra network. This part of the documentation describes the existing discovery managers in Hydra which can be used directly by a device developer or specialised for his own device type.

BluetoothDiscoveryManager
DiscoveryManager::DiscoveryManager
ExternalBlindDiscoveryManager
ExternalDiscoveryManager
GPSDiscoveryManager
RFIDDiscoveryManager
SerialPortDiscoveryManager

TellstickDiscoveryManager
UPnPDiscoveryManager
WSDiscoveryManager
ZigBeeDiscoveryManager

6.5 Hydra-Enabling a Device

This is a brief step-by-step description on how to Hydra-enabling a device using the .Net framework and Visual Studio by Microsoft. (see chapter 8 for additional information about Hydra and .Net and chapter 10 for links to the Hydra websites)

Step 1

Create Visual Studio Project

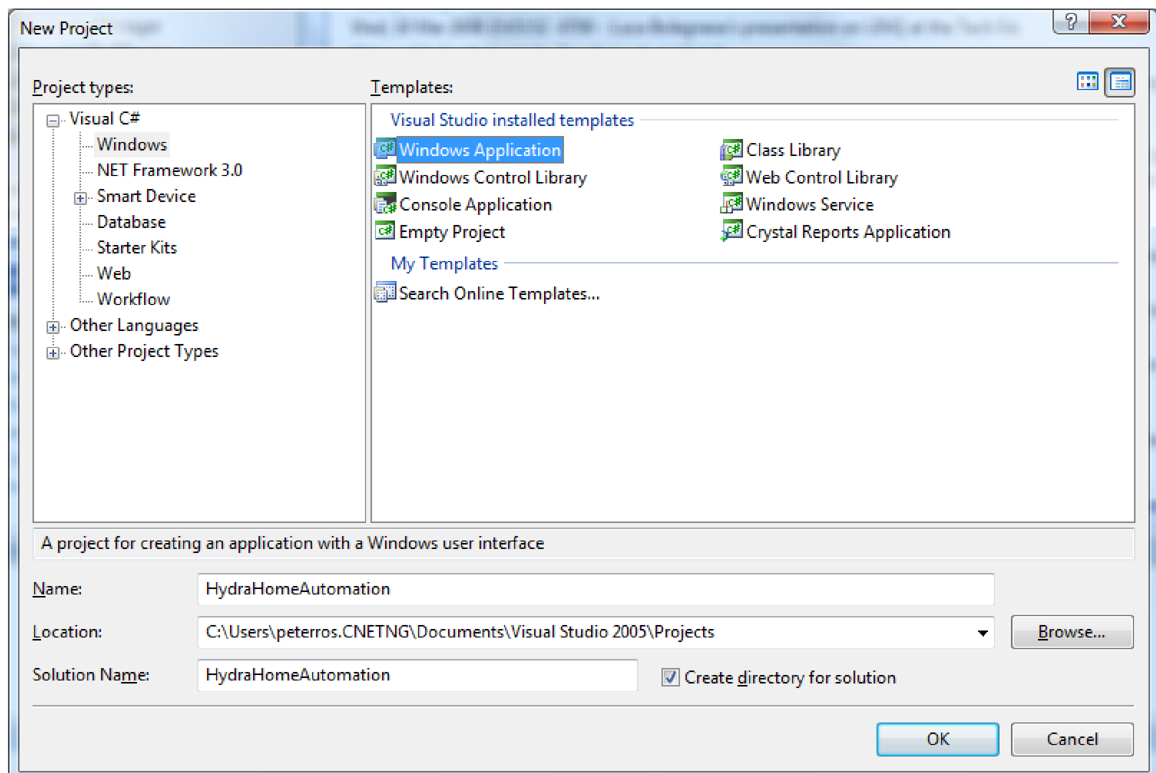

Figure 48: Create Visual Project

Step 2

Open C#-code file by clicking the ”Solution Explorer Tab” and then ”Right-click program.cs” and selecting ”View Code”

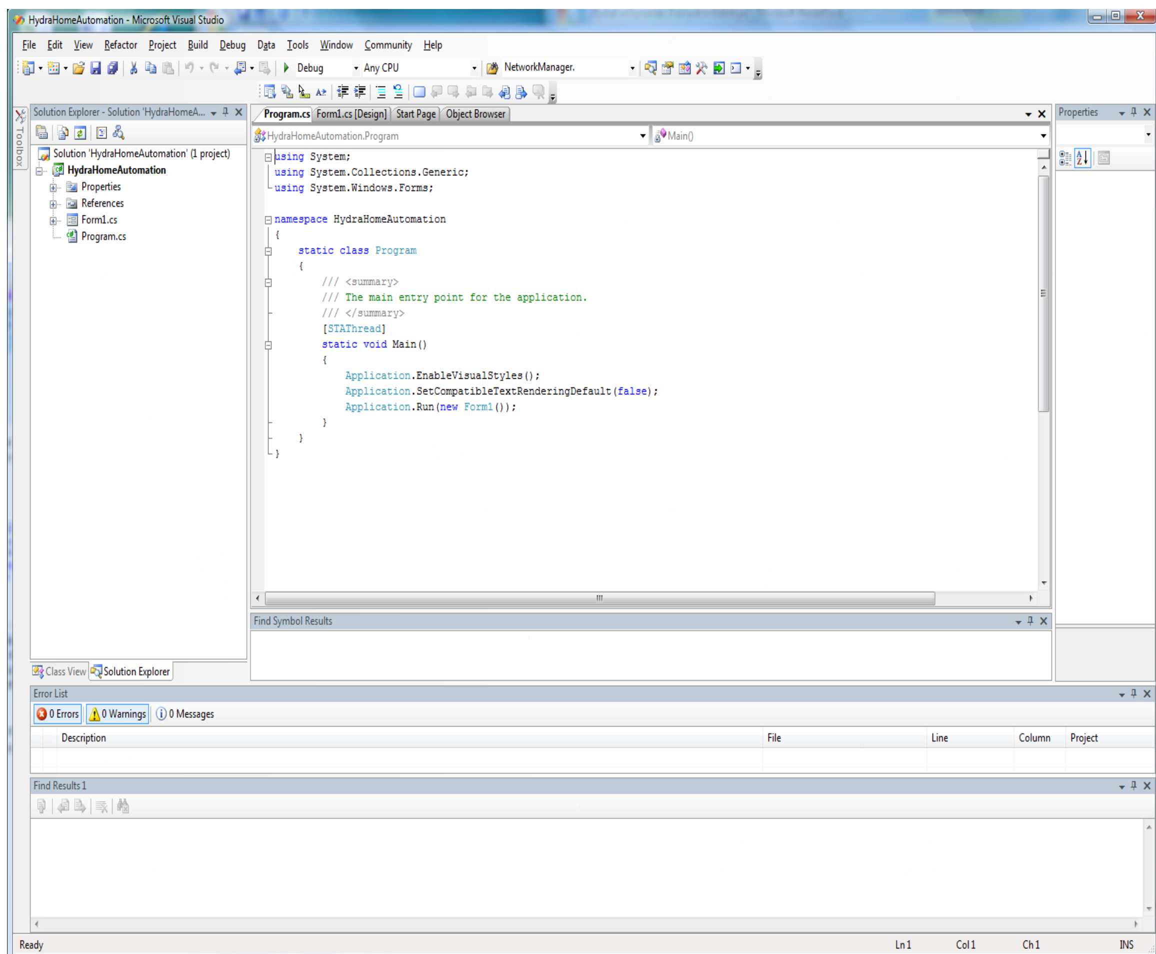

Figure 49: Visual Studio – editing file

Step 3

Add Web References for a Device

Right-click ”References and select :

”Add Service Reference”/”Advanced”/”Add Web Reference”

 to get to the form.

URLs to be used for testing

DiscoBall: http://212.214.80.161:8080/3/BasicSwitchWS

Fan: http://212.214.80.161:8080/4/BasicSwitchWS

Light: http://212.214.80.161:8080/2/EnhancedSwitchWS

Thermometer: http://212.214.80.161:8080/ThermometerWS

Windmeter: http://212.214.80.161:8080/WindmeterWS

Treno: http://212.214.80.161:8080/7/BasicSwitchWS

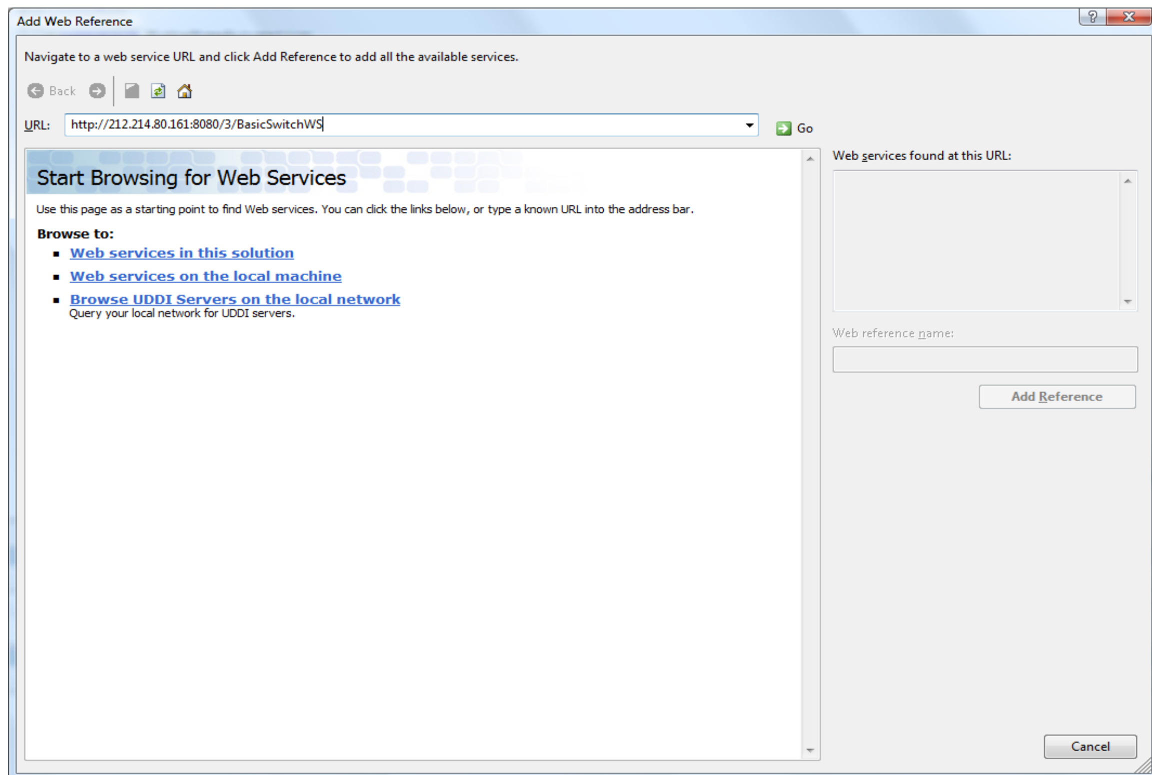

Figure 50: Visual Studio (WebServices)

Step 4

Overview of solution approach

- Connecting to NM to get a HID for the Application Device Manager
- Use the AppDevMgr to find HIDs for devices
- Create WS clients for the devices
- Create Event Listener
- Register Event Listener with Event Manager
- Inside the Event Listener implement the logic to turn on and off devices depending on events.
- Test and Debug.

Step 5

Use NM status page to view NM contents

Connect to a Network Manager:

```
NetworkManager.NetworkManagerApplicationService nm = new
NetworkManager.NetworkManagerApplicationService();
```

Step 6

Find the HID for an ApplicationDeviceManager

Use the HID to create an endpoint for a SOAPTunnel;

```
ApplicationDeviceManager.ApplicationDeviceManager myAppDevMgr = new
torinotest.ApplicationDeviceManager.ApplicationDeviceManager();
string AppDevMgrHID =
nm.getHIDsbyDescriptionAsString("ApplicationDeviceManager:BLONDIE:StaticWS
");
myAppDevMgr.Url = "http://10.38.101.30:8082/SOAPTunneling/0/" +
AppDevMgrHID + "/0/hola";
```

Step 7

Use graphical DAC browser to find WSDL-file for device or endpoint.

Import and create WS clients in your program for device.

Use Application Device Manager to find the HIDs for the device (example PetersLight).

```
string discohid=myAppDevMgr.GetHID("","PetersLight");
```

Step 8

Test your interface with the device

```
Light. BasicSwitchWS db = new Light BasicSwitchWS();
db.url="http://10.38.101.30:8082/SOAPTunneling/0/" + lightHID+ "/0/hola”
db.TurnOn();
```

Step 9

- Create Event Listener
- Create Event Manager interface
- Create Network Manager Interface
- Create a HID for your event listener
- Subscribe to the event types you want to listen to

```
EventManagerService em = new EventManagerService();
NetworkManagerApplicationService nm = new
NetworkManagerApplicationService();
string myhid = nm.createHIDwDesc("PhidgetEventStack", addresse);

em.subscribeWithHID("phidgetsensor/79285/ValueChanged", myhid);
```

Step 10

- Start listening to events
- Processing incoming events, be sure to check the status of the device before turning on.

```
PhidgetEventHandler.notifyResponse
EventSubscriber.notify(PhidgetEventHandler.notify request)
{...
      else if (request.@event[1].value == "Light_sensor")
                 {
                     Light. BasicSwitchWS db = new Light. BasicSwitchWS ();
                     bool on = (db.GetSwitchStatus().ToLower() == "on");
                     if (System.Convert.ToInt32(request.@event[2].value) <
200)
                     {
                         if (!on)
                             db.TurnOn();
                     }
                     else if(on)
                     {
                                 db.TurnOff();
                     }
...}
```

Step 11

- Build Application

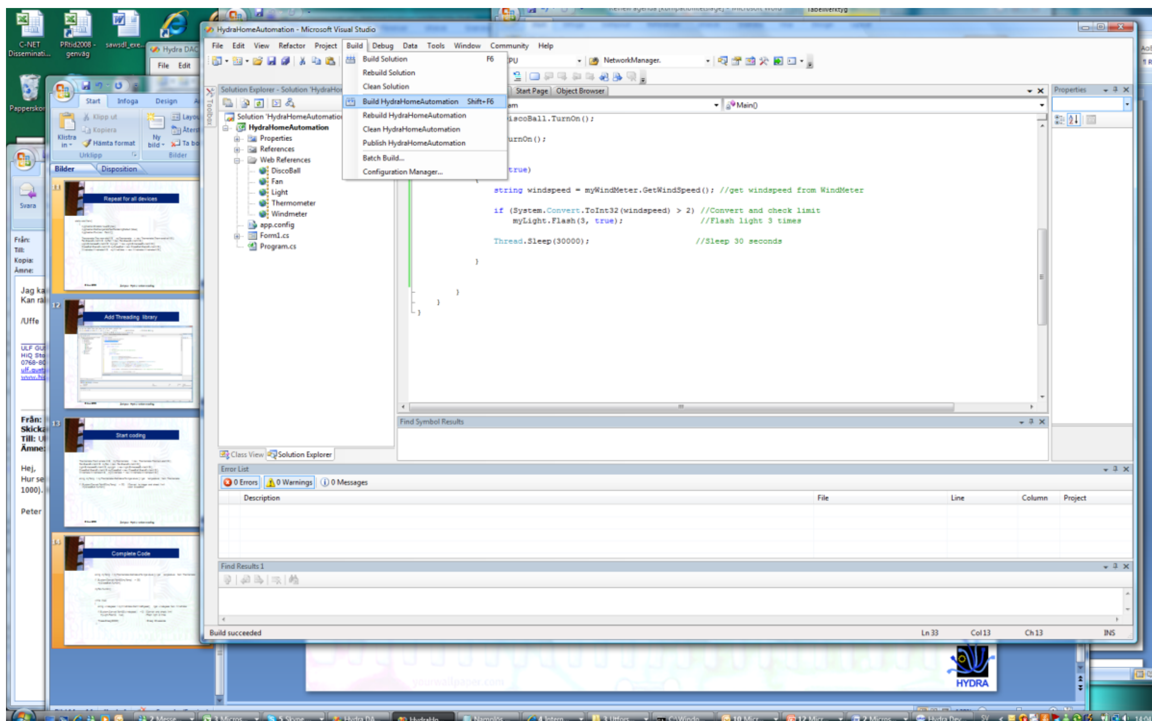

Figure 51: Build application

Step 13

If needed for debugging: add a break point and start the execution

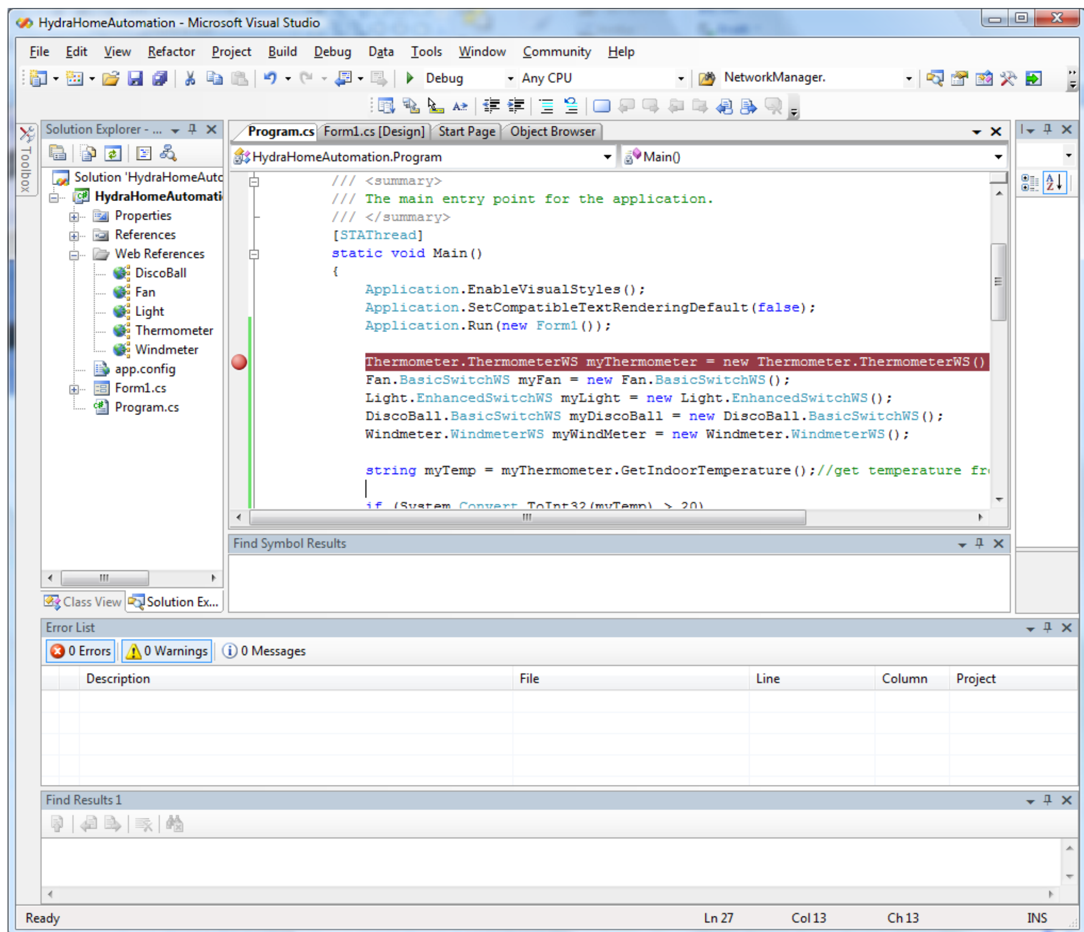

Figure 52: Adding breakpoint

Hint: Close the Form that pops up when application starts, to allow execution to reach the breakpoint.

7. Integrated Development Environment – Java

This section provides tutorials on how to use each component of the Java (Eclipse) IDE, including a general introduction to the whole IDE itself (where to find views etc).

7.1 Network Manager IDE

7.1.1 IDE connection

Using the NetworkManager for connecting the Hydra IDE to the middleware instance provides the benefit of being able to access every Hydra device that is present in the global Hydra P2P network – even those which were not accessible over plain TCP connections, for example because they are located behind firewall restricting access to TCP sockets from the outside. There are also no special security concerns about this connection as the communication is protected by the standard Inside Hydra security mechanisms and accesses to the managers are controlled by the policy framework.

However, when accessing a Hydra device over the NetworkManager, a developer can only make use of the functionality which the managers provide to the Hydra network. In many cases this is not sufficient, especially when it comes to configuring the security or internal settings of the managers. For example, a common task for a developer would be to modify access control policies. As the developer's own access to the policy framework would be controlled by the policies, modifying the very policy set that grants him access runs the risk of unintentionally locking out the developer from Hydra devices and the policy administration service itself. So, an alternative way of accessing managers directly at OSGi layer is required in some cases.

For this purpose we provide the possibility to connect the Hydra IDE to the middleware over R-OSGi connections. After starting the IDE, application developers can choose which type of connection they prefer using the eclipse dialog box under Window ! Preferences ! Hydra Middleware ! Connection. In either case, they have to provide the address of the middleware instance to which a connection should be established – either as a HID when using the NetworkManager or as a URL when using R-OSGi.

7.1.2 Remote connection

The remote connection feature enables the Hydra IDE to develop software for Hydra middleware without having a Hydra installation running locally while connecting to a remote working Hydra installation. This allows the developer to run Hydra user applications on a device without having Hydra middleware installed or running on it.

In order to configure the connection to the Hydra middleware in Eclipse, the developer has to select in the file menu Window -> Preferences and from there select Hydra Middleware -> Connection page (as seen in Figure 53 below).

Figure 53: Hydra Middleware Connection configuration page

Once the remote connection has been configured, the remote connection can be established and discarded by pushing the remarked button of the Hydra IDE toolbar as seen in Figure 54.

Figure 54: Remote connection button

The remote connection only works if both HydraMiddlewareAPI and HydraMiddlewareClients bundles are running in the system. When connecting remotely, a set of functions provided by the GlobalHydra IDEUtils class included at the package com.eu.hydra.main.global of the Hydra IDE bundle are used. This class provides the following functions to ease the remote connection, irrespective of the way used to establish the connection:

- connectToRemoteHydra(): this function connects to a remote Hydra instance. The configuration preferences are taken from the preferences page, as already seen.

- getRemoteOSGiService(String serviceName): this function provides an object which will be the instance of the required service. A developer can use this function in order to get an instance of the service for using it. The returned object must be cast in order to be compliant with the suitable interface.

The Hydra IDE provides to means for remotely connect to a Hydra installation: R-OSGi and the Hydra NetworkManager.

7.1.3 Hydra Status and Configuration views

Another important feature of the Hydra IDE is the possibility of controlling the status and the configuration of the Hydra Middleware (the Hydra managers) within the same Hydra IDE. A set of views have been deployed in order to provide information about the Network and Event manager status, so as to be a way of configuring the different managers adapted to the common configuration system.

In particular, and referring to the Network Manager, the Network Manager UI bundle (from the SVN at the /trunk/ide/eclipse folder) provides these views. This bundle must be run along with the Hydra IDE bundle.

Then, when deploying these views, the developer has to go to Window -> Show view -> Other… in the menu and then select Hydra Status and Configuration. Afterwards, the developer has access to the view.

The Network Manager Status view provides information about the Network Managers and other HID instances running in the Hydra network. It provides the same information as the Network Manager section of the HydraStatus page accessible via the web browser, providing an HID, description, IP and endpoint of all located hydra devices. The developer can select between showing the Network Managers, the local HIDs (local Network Manager included) or the remote HIDs (remote Network Managers included).

The view also provides a dynamic search system by typing on a text box, along with the possibility of copying the information from the data table by right-clicking over the information of interest.
A screenshot of the Network Manager Status view can be seen in Figure 55 below.

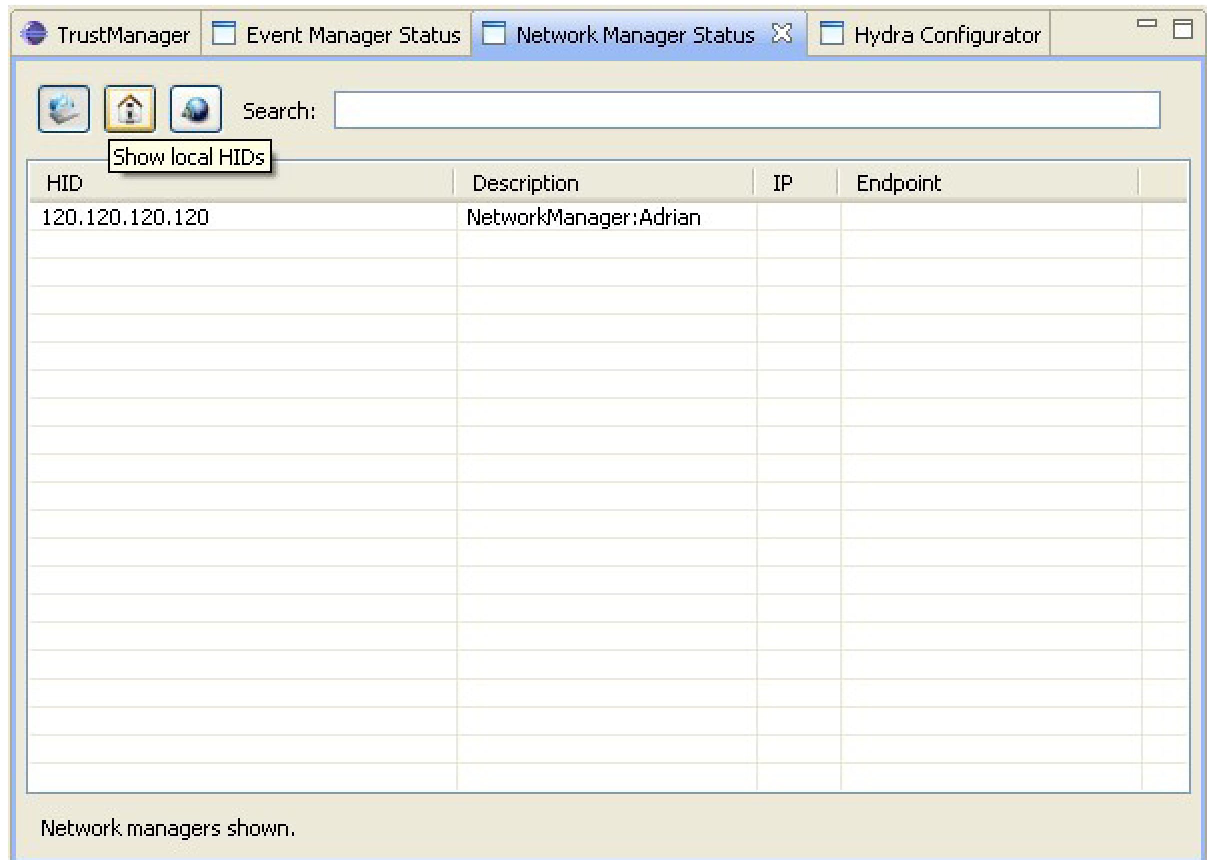

Figure 55: Network manager status views

Then, the Event Manager Status view provides information about all subscriptions managed by the local Event Manager in a similar way as is done in the Event Manager section of the HydraStatus page. It provides information about the topic, endpoint, date and counter of all subscribed events on a table, in a similar way as the Network Manager Status view does.

It also provides the same dynamic search system and the same data copying system as the Network Manager Status view does. A screenshot of this view can be seen in Figure 56 below.

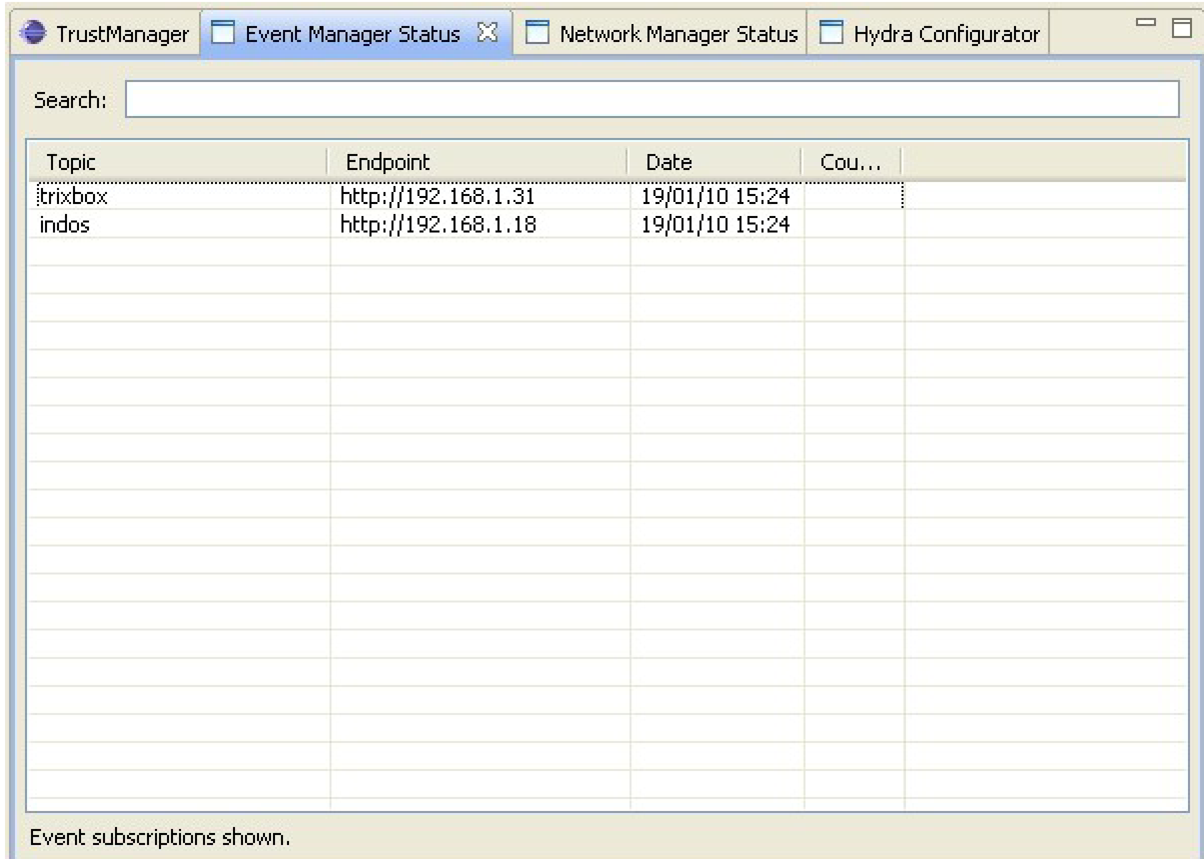

Figure 56: Event Manager Status view

The Hydra Configurator view provides a way of configuring the different Hydra managers previously adapted to the common configuration system. Using this view it is possible to modify the configuration of a manager in a dynamic way, being the case that these modifications are effective from the moment they are sent to the system.

The Hydra Configurator view interface consists of a set of deployable bars, one used to apply all the changes performed during the configuration process, the others devoted to the different managers adapted to the common configuration system. Once deployed, the list of configurable properties and options is shown, being possible to modify their values, written inside textboxes. A screenshot of the Hydra Configurator view can be seen in the following Figure.

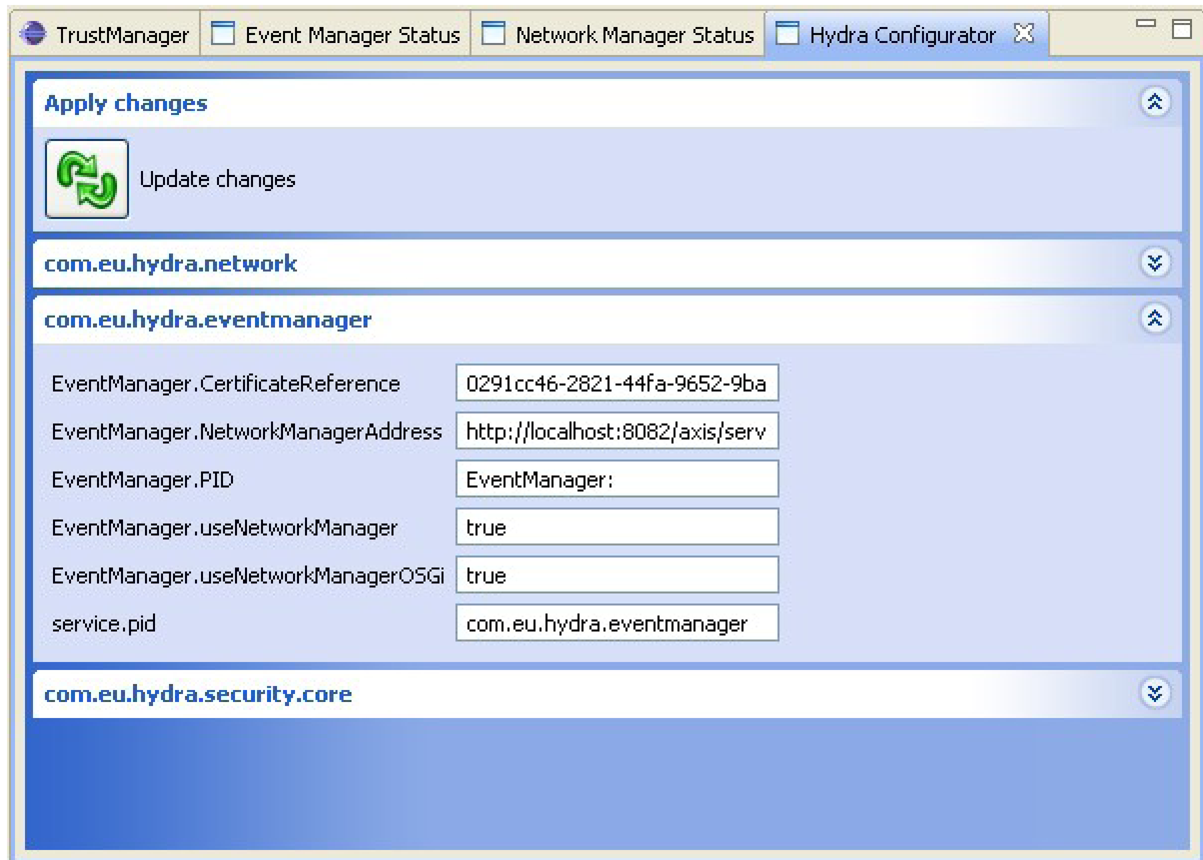

Figure 57: Hydra Configurator view

7.2 Trust Manager IDE

The Hydra TrustManager is responsible for evaluating the trust value of certificates, as they are sent by Hydra devices upon establishment of a communication session, for example. The trust value reflects how much the TrustManager (or the developer who configured it) relies on the authenticity of the cryptographic key contained in the certificate. A trust value of zero means that there is no evidence that a certain cryptographic key does actually belong to the device claimed in the certificate's attributes. As a consequence, one cannot be sure with whom communication is taking place – if such an untrusted key is used, communication might be protected against eavesdropping but by no means can it be assured that one is not communicating with the attacker himself. So, certificates with a trust level under a certain threshold (which can be configured by the developer) must be considered as insecure.

The TrustManager GUI is a user interface to the TrustManager which is integrated into the general Hydra IDE, based on the Eclipse RCP framework. It connects to the TrustManager and provides an interface for controlling some of its features. By opening the TrustManager preferences dialog, a developer is able to configure the connection from the TrustManager GUI to the actual manager. After the connection settings have been made, the developer can open the TrustManager perspective, thereby opening a list of supported trust models on the left of the screen and an editor for the respective trust model on the right. As the TrustManager provides a plug-in mechanism and loads any supported trust model at run time, the trust model list on the left is populated the first time the perspective is opened. Each trust model comes with its own administration service (for which access rules can be seperately configured) and with thus also with its own user interface. At the moment, user interfaces have been implemented for the (trivial) Null trust model and the most-used X509v3 model.

At first, after opening the Trustmanager perspective, developers can view the list of trust certificates, i.e. the list of all certificates which represent a Certification Authority or which are explicitly marked as "trusted". For each of these certificates, details about its content can be retrieved, including the cryptographic public key and the list of attributes which have been created using the NetworkManager's createCryptoHID()-method (c.f.Figure 58). If a certificate is not trust edanymore it can be deleted by using the context menu in the certificate list. The next tab of this editor enables the uploading of certificates which should be marked as trusted. The TrustManager accepts certificates in standard PKCS#12 format (i.e. .cer files), displays their content and adds the respective certificate to the list of "root of trusts", i.e. the certificate itself and all public keys which have been signed with that certificate are considered to be trusted from now on (c.f. Figure 59). The third tab of the editor can be used by developers in order to test which result the TrustManager would generate for a certain certificate. By uploading a certificate file – again in PKCS#12 format – into the TrustManager and clicking on the Validate button, the validation process is started and the trust value for the uploaded certificate is evaluated. In the screen shot in Figure 60 a trust value of 1.0 is returned, which means that the tested certificate is fully trusted and Hydra devices using keys signed with this certificate are accepted for communication.

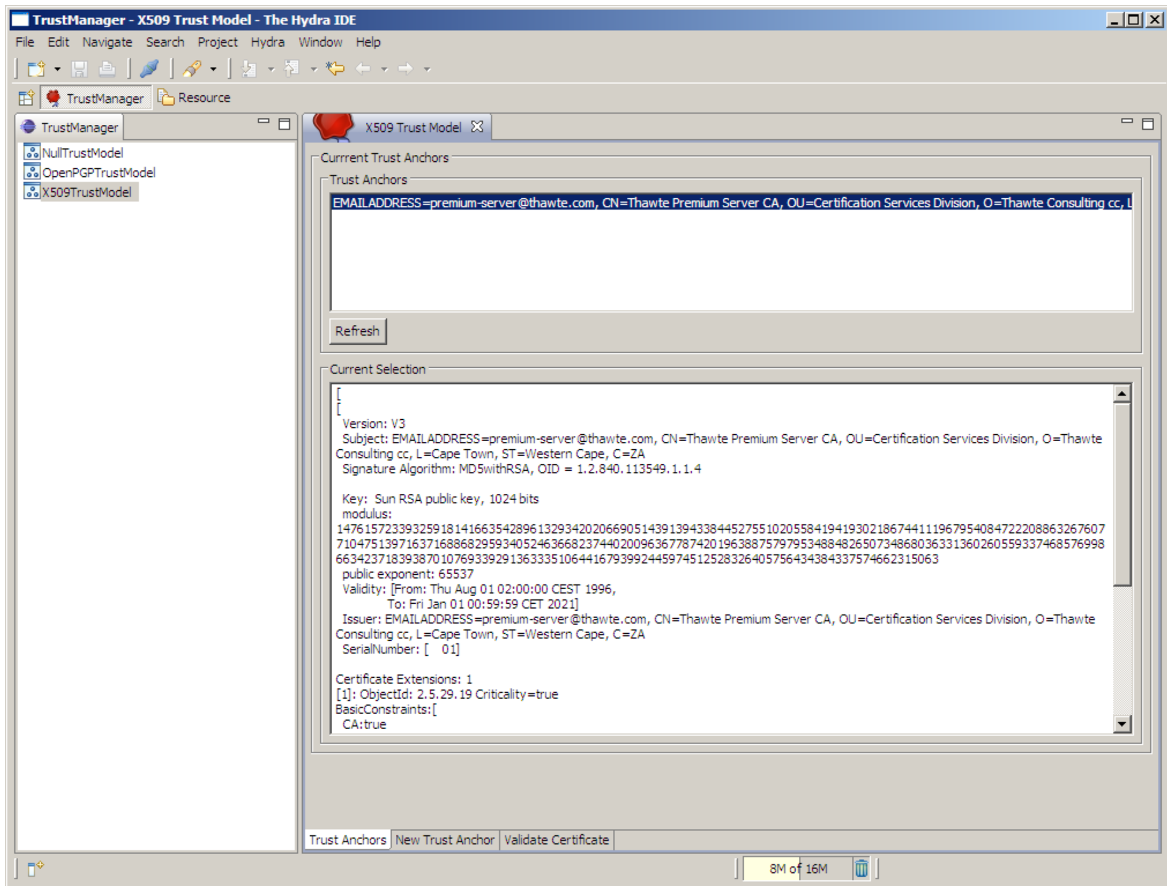

Figure 58: TrustManager GUI showing the details of a X509v3 certificate

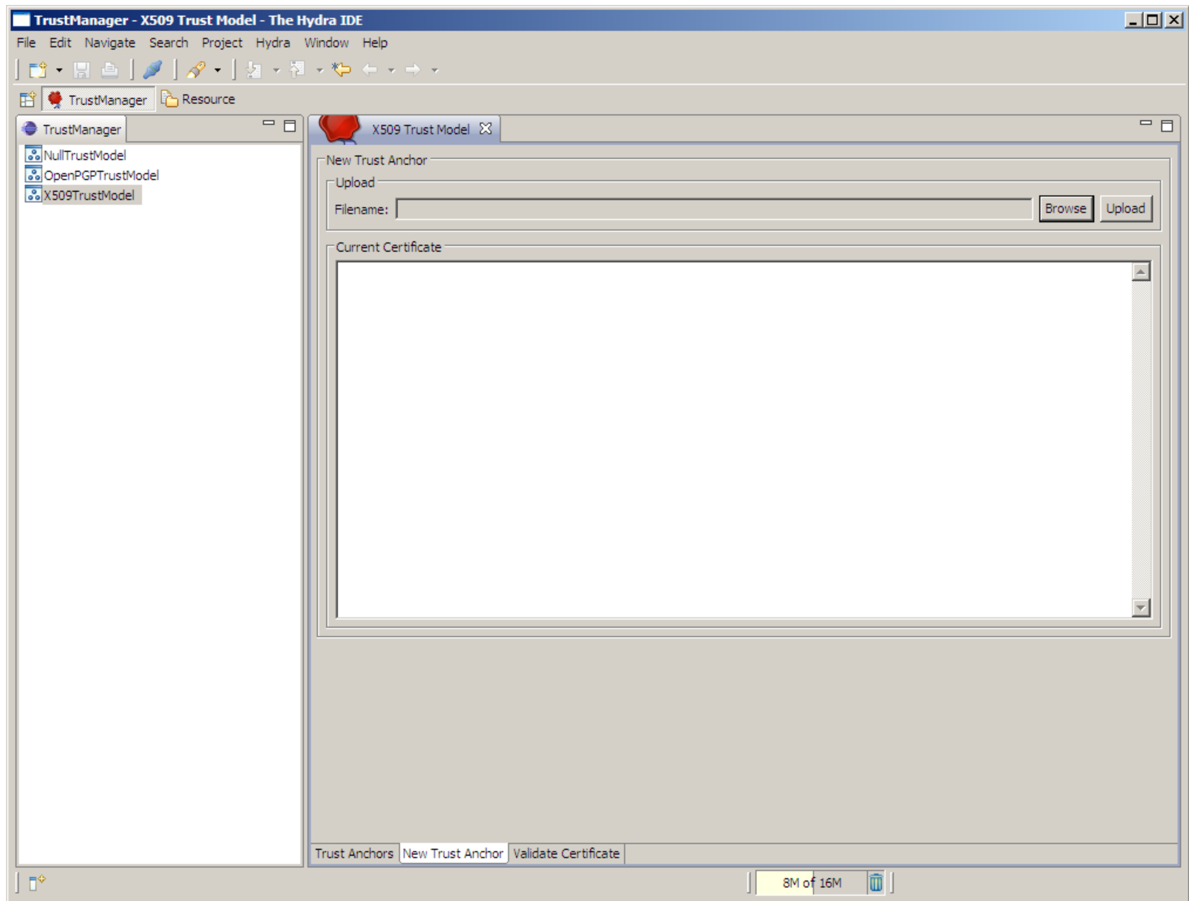

Figure 59: Adding a new trust root to using the TrustManager GUI

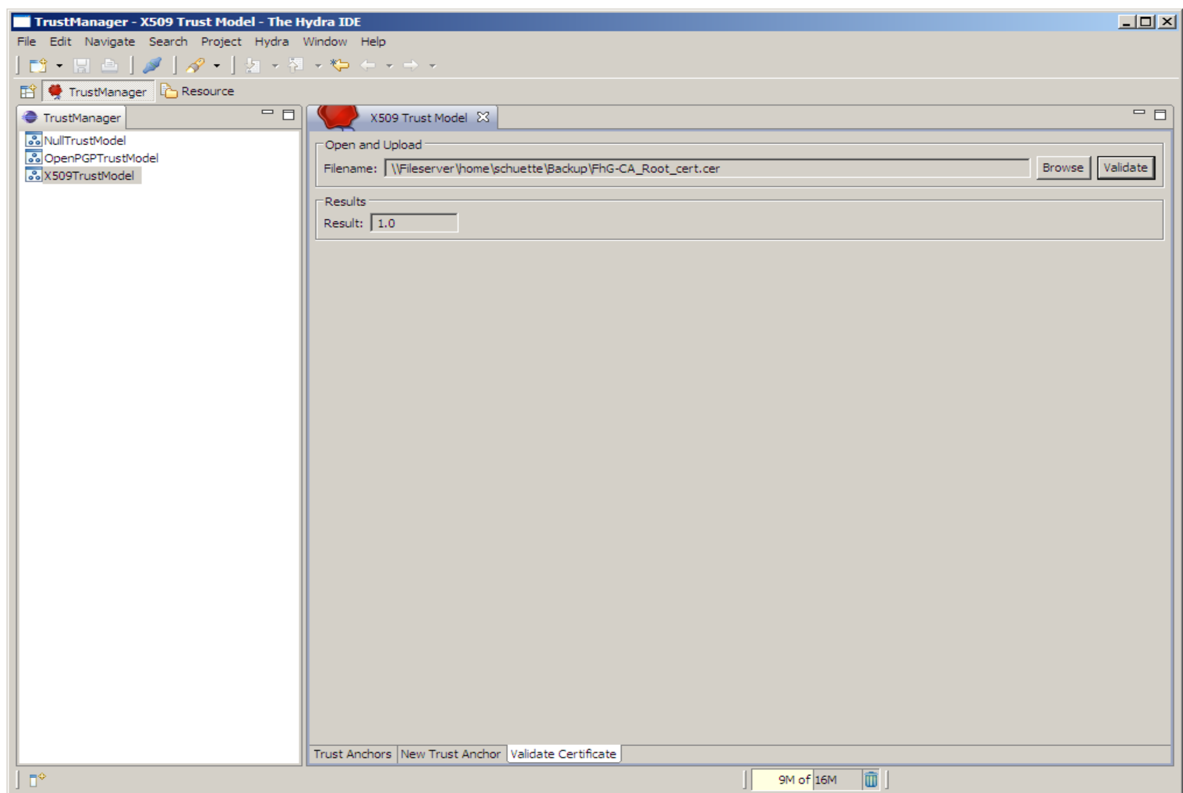

Figure 60: Validating a certificate using the TrustManager GUI (Trust Manager IDE)

7.3 Crypto Manager IDE

Crypto Manager GUI

The CryptoManager is a stand-alone manager providing various cryptographic operations such as encryption, key management and handling of digital signatures. It is basically used in two ways: On the one hand, the CryptoManager is automatically used by internal part of the Hydra middleware – such as the security modules in the NetworkManager. On the other hand, the CryptoManager can be used by application developers as a tool for applying cryptographic operations and managing keys. The idea of the CryptoManager is to abstract away cryptographic keys and algorithms. Developers using the CryptoManager only need to specify an "identifier" along with the message format. The CryptoManager will then take care of selecting appropriate keys and algorithms. So, the CryptoManager facilitates writing secure applications by encapsulating complex cryptographic operations in an easy-to-use interface.

The GUI for the CryptoManager supports application developers to manage keys stored inside the CryptoManager. The basic view shows a list of all stored keys providing details such as the corresponding identifier and the key type (see Figure 61). Basic operations are key deletion and list refresh (buttons on top of the list). A double-click on one of the RSA certificates opens a message box which presents the certificate details.

Apart from managing existing keys the CryptoManager GUI makes it possible to invoke the generation of new keys. The two buttons "Generate Symmetric Key" and "Generate Certificate" trigger the CryptoManager connected to the IDE to start a key generation. The "Generate Symmetric Key" button starts key generation immediately and presents the identifier of the new AES key as result.
The "Generate Certificate" button opens a dialog which makes it possible to specify certificate attributes and as optional a HID (see Figure 62). After finishing the wizard, the CryptoManager generates a new RSA certificate and the wizard presents a message box with the identifier of the new certificate.

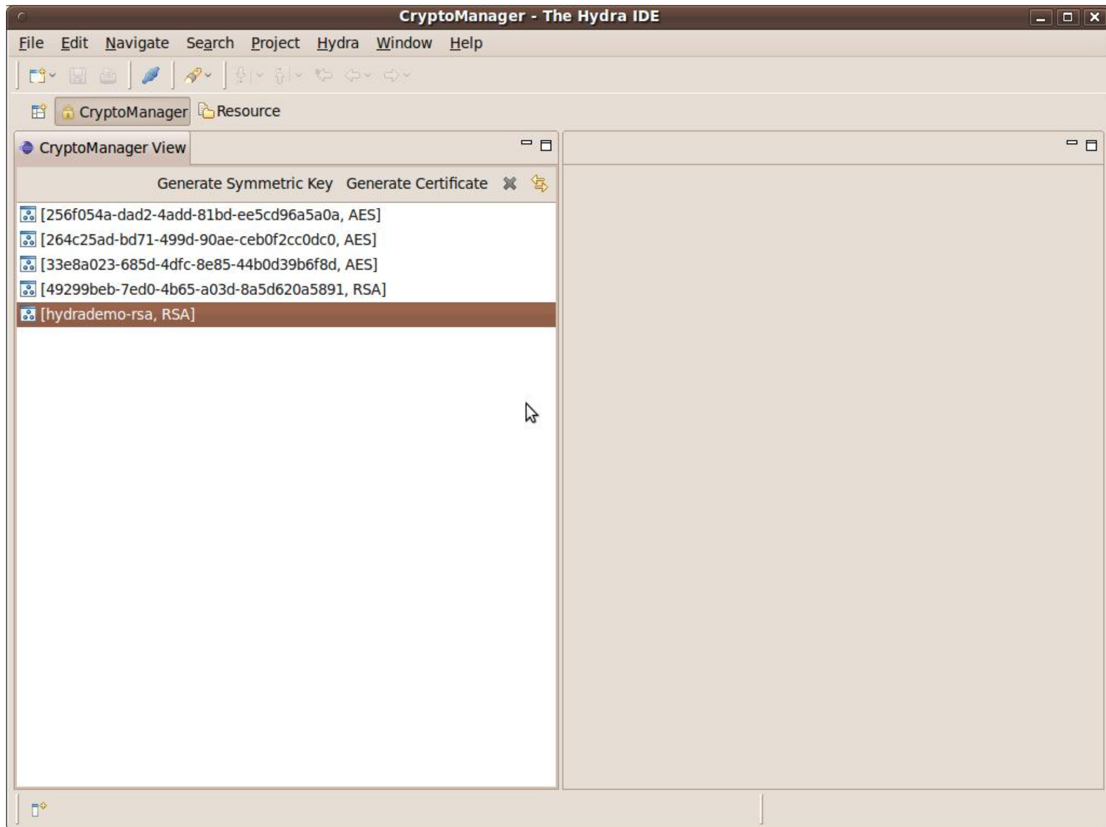

Figure 61: CryptoManager View

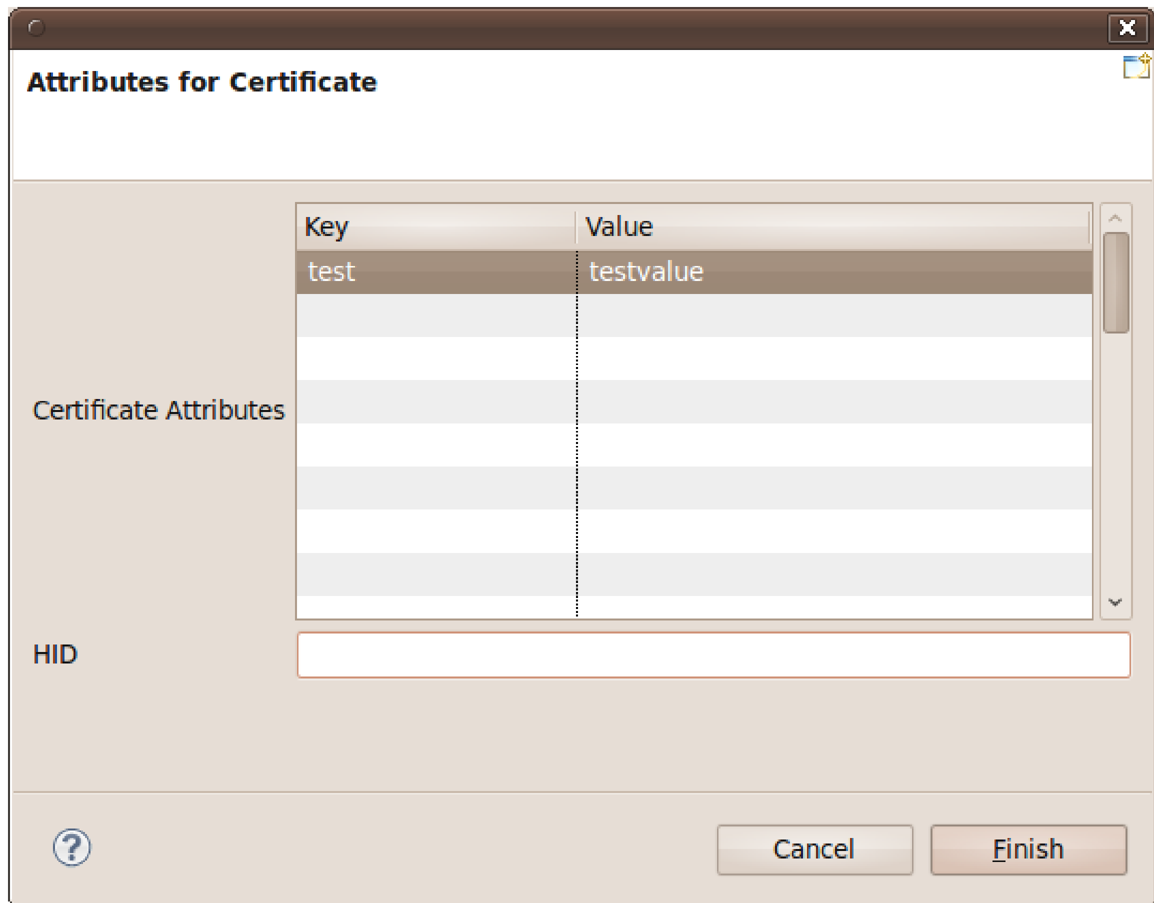

Figure 62: Certificate generation wizard

7.4 Context Manager IDE

The Context Manager IDE components provide the functionality for creating Context Specification objects, that can be sent to the Context Manager, as described in the SDK section for the Context Manager 5.7.1. It also provides views for runtime management of Context Managers, for retrieval of installed context specifications etc. As well as Context Specifications, the IDE also supports the creation of named Context Queries that can be stored in the Context Manager, and called by name at runtime.

7.4.1 Context Specifications

As described in D12.8, Context Specifications can represent three different types of context - Application, Device and Semantic, where Semantic contexts can also be broken down into different types - Location, Environment and Entity. When creating a new Context Specification, the exact type can be specified, along with the name of the context - the *contextId*. New contexts can be created in the same manner as adding any other object in a project:

New -> Other -> Hydra -> Context Specification

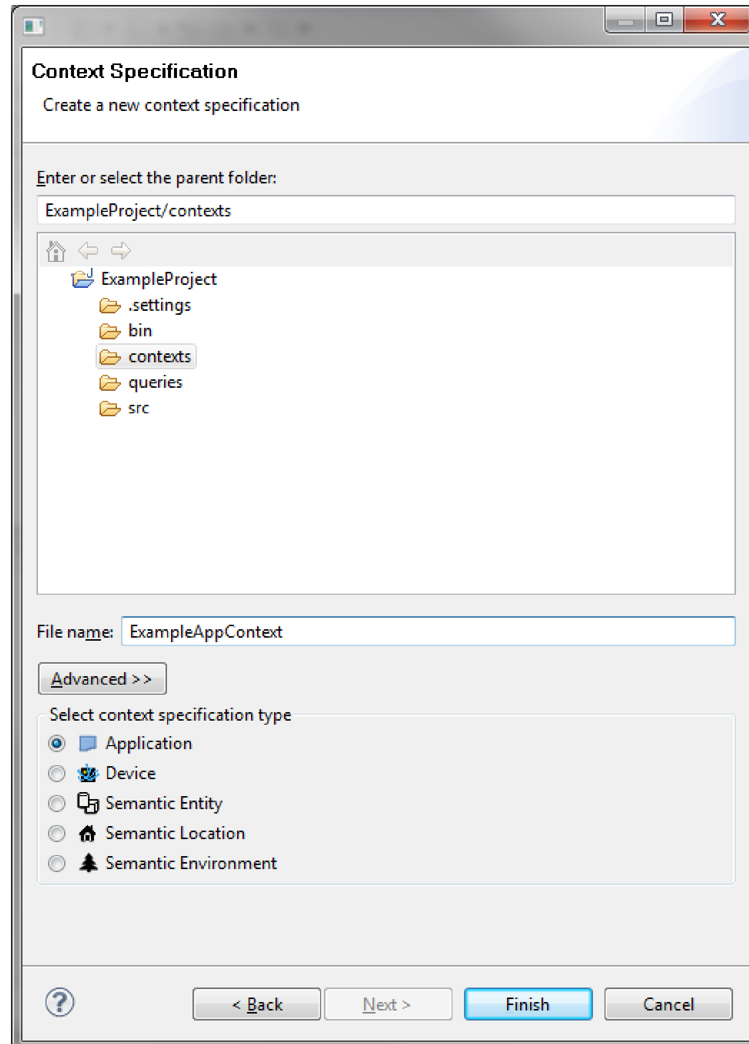

Figure 63: Create Context Specification Wizard

This throws up a wizard, as shown in Figure 63 above, generating the empty context specification in the workspace location specified. Double-clicking on the .ctx (Context Specification) file in the workspace, opens up the file in the Context IDE Editor, in which the Context Specification can be edited. As discussed in the SDK section, Context Specifications, and Context Query Sets are persisted as XML, but transferred as objects using the methods described. The IDE editor provides the ability to edit the three core facets of a Context Specification, these being:

- Context Definition
  - Defines the properties of the context, as well as any data members
  - Present in ALL context types
- Data Subscriptions
  - Subscriptions for data from a data source, forwarded to the Data Acquisition Component
  - Only in *Device* contexts
- Context Rules
  - A set of rules, types and functions that define the reasoning performed by the context, in addition to the context-sensitive actions
  - Present only in *Application* and *Device* contexts - not *Semantic*

Each of these is represented by a different tab in the Context Editor, in addition to a *Source* tab, that shows the (un-editable) XML source for the Context Specification, as it is stored locally.

Figure 64: Context Definition page

The Definition page, shown in Figure 64 above, demonstrates the interface for defining the properties and members of a context. Defined properties are no necessarily static, as they may be updated based on reasoning performed by rules - for example, the "location" property of the mobile phone above may be updated if some other rules detect its location as being elsewhere, or that it is updated directly.

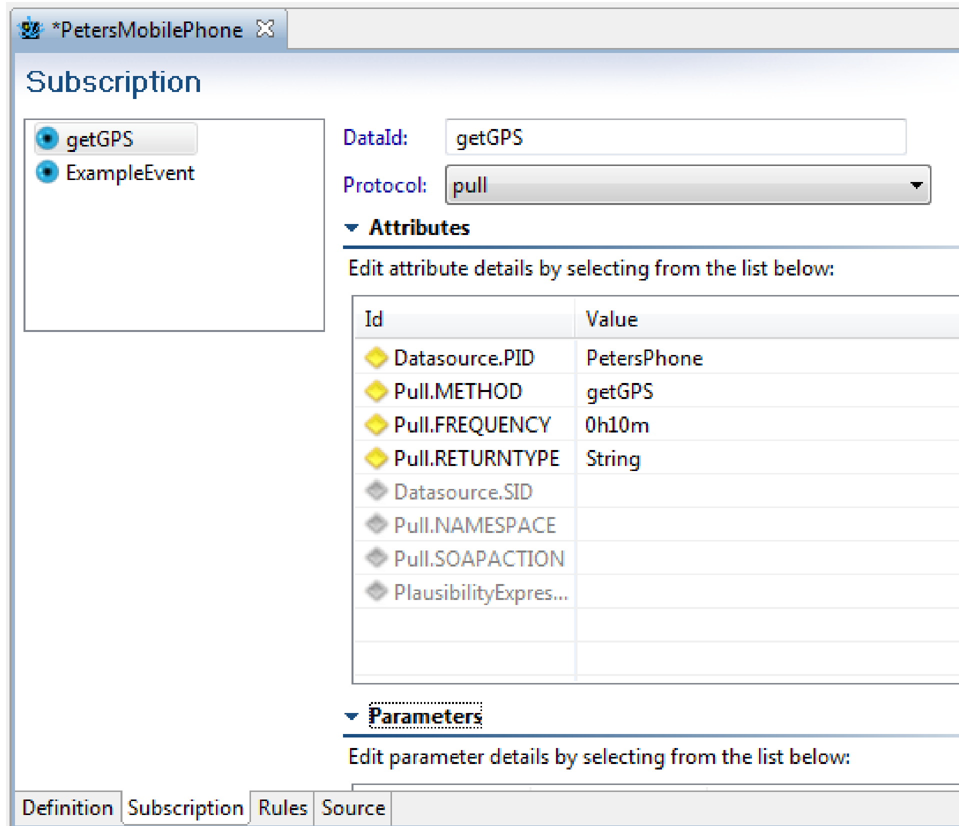

Figure 65: Data Subscription page

Figure 65, above, shows the Data Subscription page, that specifies a set of subscriptions to retrieve the desired data made exposed by the data source - through either the PUSH or PULL protocols, as handled by the Data Acquisition Component. The example shown demonstrates a PULL subscription, featuring the attributes recognised by the PULL protocol, with the non-mandatory being greyed-out.

The functionalities of context-reasoning and interpretation, in order to perform context-sensitive actions, is enabled by the set of context rules provided with the *Device* and *Application* contexts. The Context Manager uses the DROOLS Rule Engine [4], to process these rules as well as to maintain the modelled context from data acquired from data sources - *Context Consumers*. Context Rules in a Context Specification contain several different parts that can be specified. These are:

- Imports
- Types
- Functions
- Rules

Context Rules provide the ability to declare inline types and functions that may be used in Rules for a specific purpose. Any libraries referenced in these Rules / Functions / Types must be specified with their fully qualified name (package name & class name), in the Imports, such that the Rule Engine can recognise what they represent. Declared Types allow for Contexts to store certain data for internal purposes, such as recording historical data. Declared Functions allow for Java-coded functions to be defined and declared inline, that can be used in the rules themselves. Rules are the individual rules entered into the Rule Engine. As previously mentioned, these are Drools formatted rules - the core of which are “When” clauses and “Then” actions, along with additional Drools rule attributes that can be associated with a rule, to alter how it is handled by the Rule Engine.

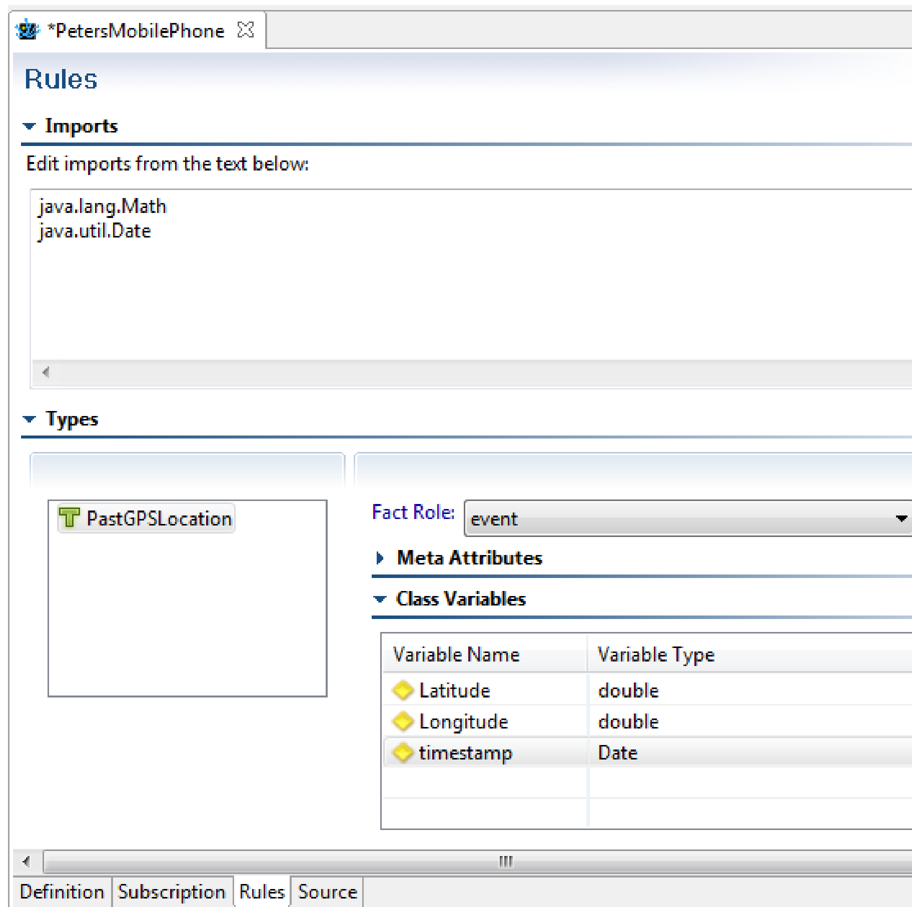

Figure 66: Context Rules - Imports and Types

Figure 66, above, shows part of the Context IDE interface for Context Rules, through which classes can be imported (to use in types, functions or rules), as well as for the creation of types within a context. In the example, the *timestamp* variable of the *PastGPSLocation* type, uses the *Date* type as referenced in the imports. The fact role defines how the Rule Engine handles the type - the role of *event*, as opposed to *fact*, declares that the type should be handled as an event, meaning that the instance gets remembered temporally, such that it can be reasoned over as such.

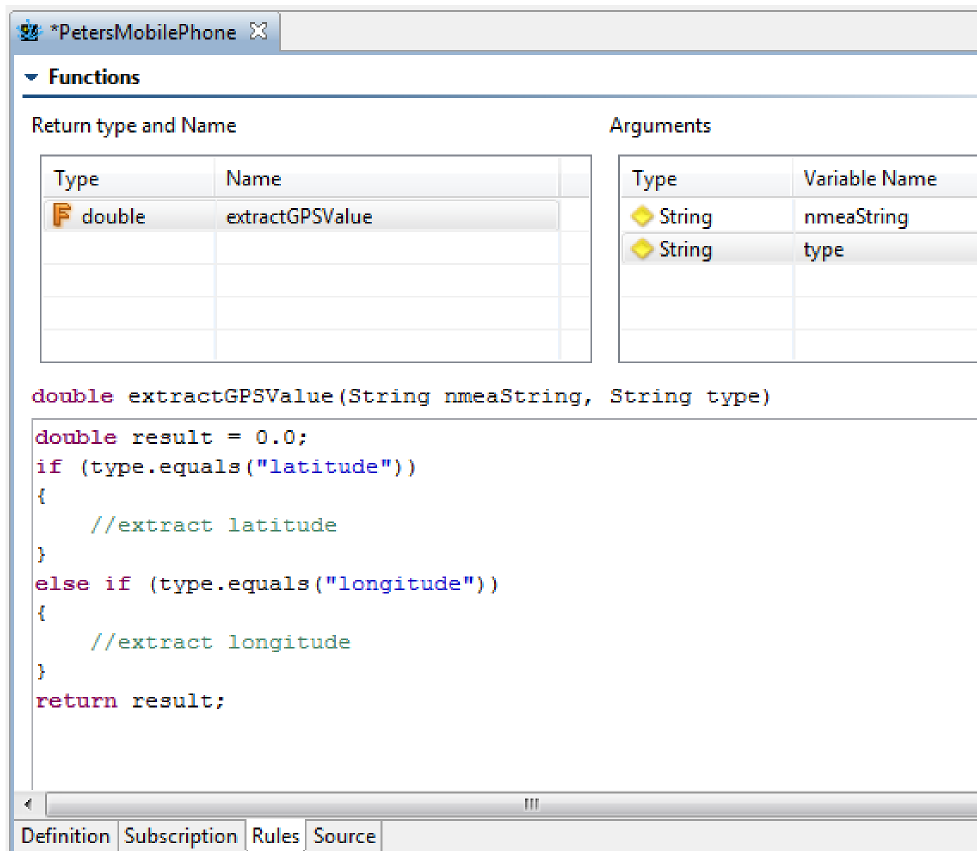

Figure 67: Context Rules - Functions

Figure 67 above shows the Context IDE interface for creating Declared Functions in a Context Specification. As previously stated, these are essentially Java-code functions, that can then be used in Context Rules. The example function, *extractGPSValue*, could be used to extract the desired GPS positional value from an acquired NMEA formatted string.

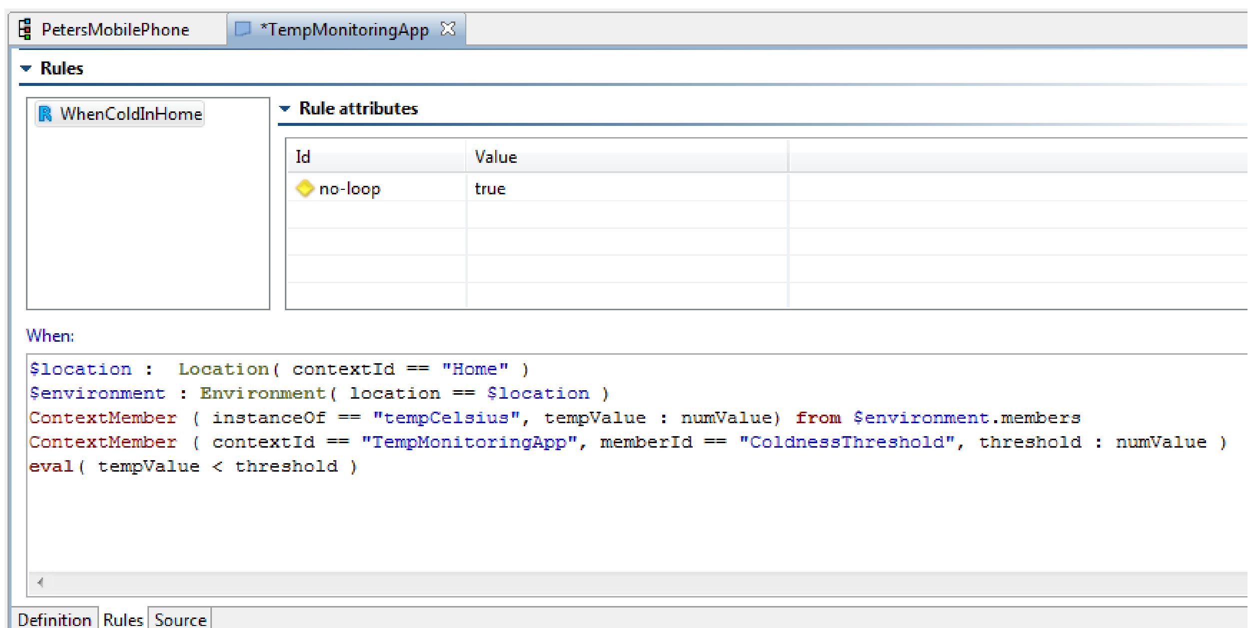

Figure 68: Context Rules - Rules LHS

Figure 68 above shows the IDE interface for creating the LHS (When) of Context Rules. These are DROOLS DRL 'When' conditional statements, using the same syntax as described in the DROOLS

documentation [4], short example below. Also shown, is the area for specifying rule attributes such as *no-loop* and *salience*. The exact usage of these can also be found in DROOLS documentation, and upcoming Hydra training materials. The featured rule will fire when the sensed temperature (in Celsius) in the "Home" location, is below that of the threshold defined by the "ColdnessThreshold" member of the "TempMonitoringApp" Application context, of which this rule is a part.

Very basic Drools rule:

```
rule "Primitive support"
when
    $c : Cheese( $price : price )
then
    $c.setPrice( $price * 2 )
end
```

The firing of the LHS of the rule, results in the RHS actions, or the "Then" side, being triggered. The Context Manager features several pre-defined actions, including:

- Fire External Event (to Event Manager)
- Fire Internal Event
- Call Web Service
- Store Context (to Storage Manager)

In addition to the predefined actions, the developer may also write their own "Then" code, allowing for the full flexibility in authoring rules.

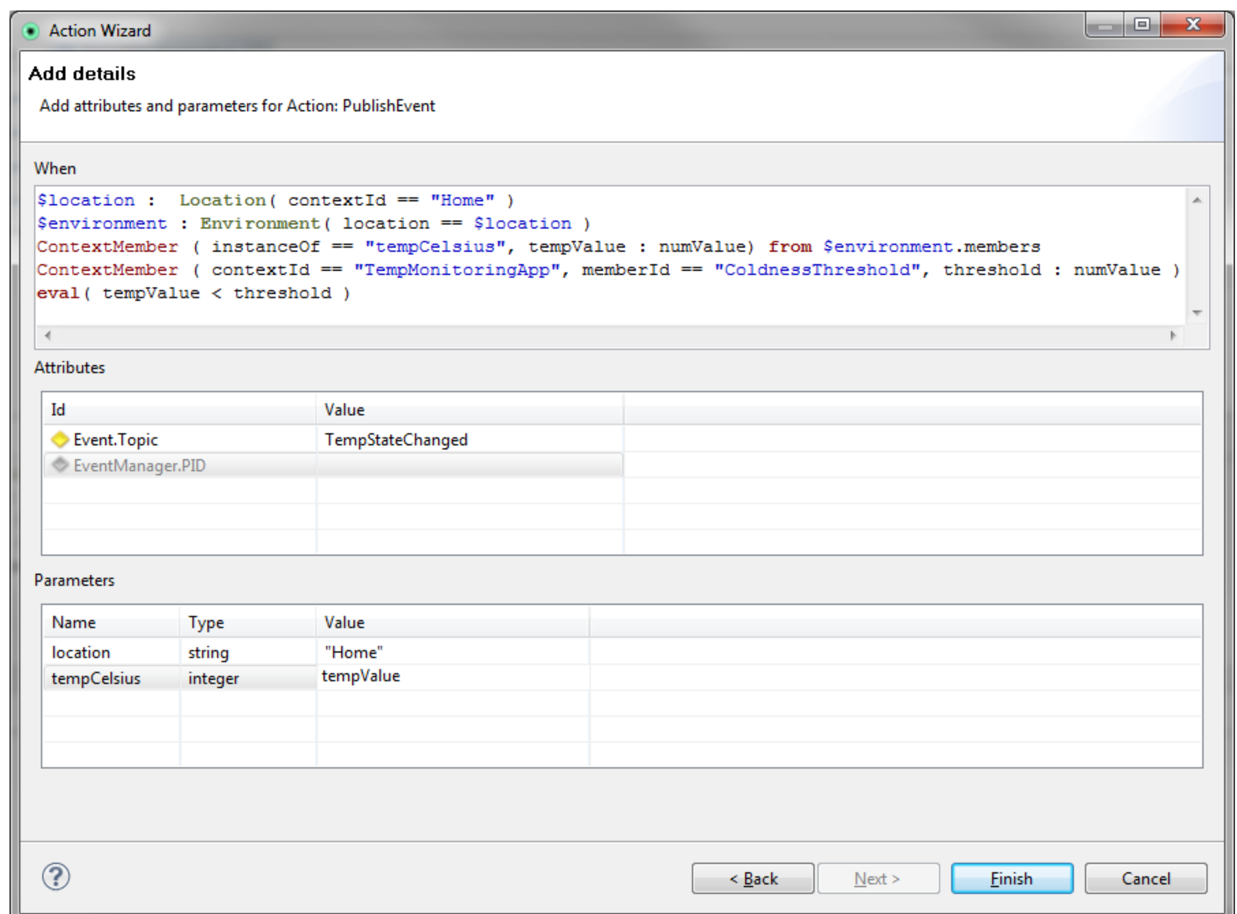

Figure 69: Publish Event to Event Manager Wizard

Predefined actions are added, throwing wizards, such as shown in Figure 69 (after the *PublishEvent* action has been chosen). Here, the recognised attributes of the action (both mandatory and non-mandatory) are displayed, as with the Subscriptions. In the figure, the *Event.Topic* and *EventManager.PID* attributes are provided, though the latter is not mandatory, as the Context Manager can also be configured with a default Event Manager to publish events (as discussed in the SDK section).

Additionally, a set of Parameters can be set, to specify the additional key-value data sent with the Event, when it is published to the Event Manager. This can either be static values (see the *location* parameter), or dynamic, using the values of the variables (from the LHS) provided - (the *tempCelcius* parameter with the value of *tempValue*.

7.4.2 Context Queries

Context Queries can be configured in the Context IDE, and persisted locally as XML, as with Context Specifications. They can be created in the same way also, as follows:

New -> Other -> Hydra -> Query Set

The Context Query Set editor contains two pages - one for any declared functions and imports (utilising the same interfaces as the Context Specification editor). The other page is for the definition of a set of queries, specifying the name, arguments, query code and output.

*ExampleQuerySet

Queries

Query Name

getDevicesOfTypeInLocation

Arguments

Type	Variable Name
String	type
String	locationId

```
getDevicesOfTypeInLocation (String type, String locationId)
```

```
$location : Location ( contextId == locationId )
$device : Device ( location == $location )
ContextProperty ( propertyId == "deviceType", value == type ) from $device.properties
```

Output

Variable	Type
$device	Device

QuerySet Queries

Figure 70: Context Queries

Context Queries, as shown in Figure 70, are pre-configured queries in the Context Manager that return a set of data as specified by the query. These are referred to by a unique identifier (the Query Name), and can take a set of arguments to the query, such that the actual results of the named query may be dynamic, but the logic used in the query to retrieve the results remains static.

These queries are stored in the rule engine, as Drools queries, and are essentially just the *When* (LHS) part of the rule, specifying constraints for collecting a set of data.

The example query, given in Figure 70, returns the encoded context state (as XML) of all Device contexts of the type specified by the *type* argument, and with location matching that is specified by the *locationId* argument.

7.5 Obligation Framework IDE

7.5.1 Obligation GUI

The Hydra middleware includes an Obligation Policy Framework which allows the execution of actions upon the occurrence of certain events. Although it appears to be similar to the Context Management Framework, both frameworks serve different purposes and are based on different technologies: The Context Management Framework primarily works on a static fact base which is built up from various data sources and integrated with the Hydra ontologies. The Obligation Framework makes use of Complex Event Processing to recognise certain situations in highly dynamic data streams such as sensor events.

In contrast to the Context Management Framework, it is not aimed at executing general-purpose actions but rather at setting (security-relevant) configurations based on the current situation. As an example, we will show how the Obligation Framework is used to set situation-specific access control rights in the Hydra middleware.

The Obligation GUI is a user interface that is integrated into the Hydra IDE and allows developers to control the Obligation Policy Framework in numerous ways. In this chapter, we describe the features of the Obligation GUI and illustrate its use by a usage case in which "situation-aware" policies are realised.

The Obligation GUI deals with the concepts Event, Action and Situation. An event is the definition of a complex event pattern describing the occurrence of a critical incidence upon which a certain action is required. The event pattern is described using the EPL1 language and refers to information that is sent on logical event channels. An event channel represents a stream of events of a certain type, such as temperature, humidity, network events, user interactions or events that have been retrieved over the Hydra EventManager. Which event channels are available at run time depends on the kind of plugins attached to the Obligation Framework and may thus vary. Every event type has a property value which can be used to retrieve the content data of the event. In addition, event types can provide further specific properties, further describing the event. For example, a temperature event could provide the properties isFahrenheit and isCelcius while a network event could be described by properties such as sourceIPaddress, packetSize, etc.

The developer can use the EPL syntax to define complex patterns of such events and then specify an action that should be executed upon the occurrence of that event pattern. Actions are defined by plugins in the Obligation Framework and can serve various purposes. In the use case example below we will use actions only for logging the occurrence of an event, however, by adding further plugins it would be possible to realise any other kind of action such as increasing the level of communication security in untrusted environments, for example.

A Situation is defined by a starting and a stopping event. Once the starting event is detected, the situation becomes active and stays it until the stopping event has been detected. The Obligation GUI allows binding access control policies to such situations so that it becomes easy for Hydra developers to specify which access rights should be applied in a certain situation.

In order to achieve this, a developer can make use of the five extensions to the eclipse IDE provided by the Obligation GUI (cf. Figure 71):

- A Message Console
- An EventListener View
- A Situation View
- An *EventListener Editor*
- A *Situation Editor*

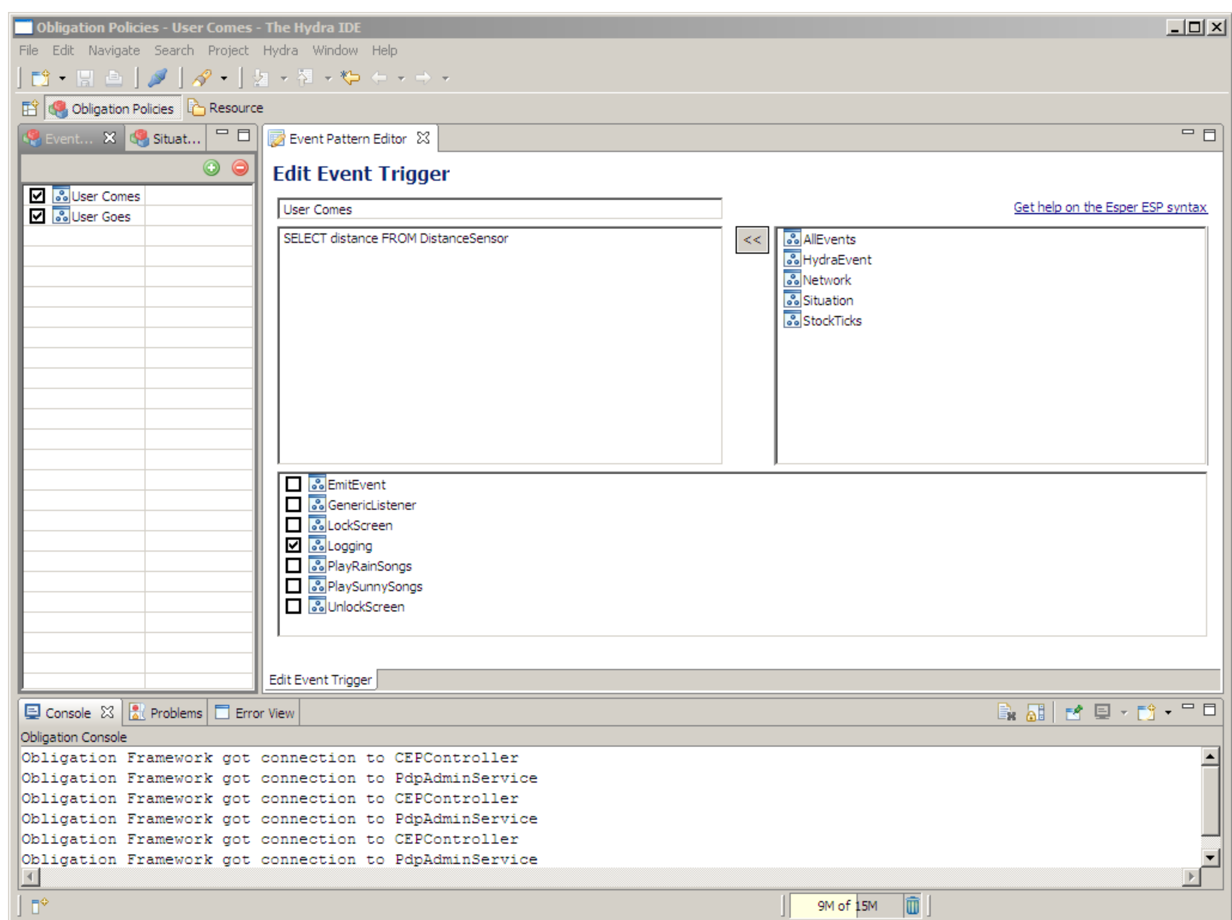

Figure 71: The Obligation GUI perspective. EventListener view on the left, message console on the bottom and event editor in the middle.

We now illustrate how a developer can use these extensions to define two complex event patterns, create a situation from them and then assign different access control policies to that situation, thereby creating a "situation-aware" policy:

At first, the developer would open the "Obligation GUI" perspective in Eclipse. At the bottom of the IDE screen he would then see the Message Console which simply displays information about the Obligation GUI, such as the status of the connection between IDE and the Obligation Framework. On the left is the EventListener View which lists the names of all complex event patterns and makes it possible to activate or deactivate each of them. If no event patterns have been specified before,

this list will be empty and the developer would click on the green "+" button in order to create a new one.

The new event pattern will be opened in the EventListener Editor in the middle of the screen. Using this screen the developer would now define the complex event pattern that indicates a change in the system status which is to be reacted on. Let us assume the developer would like to react on the (complex) event of a user leaving the home. This complex event would consist of a number of more detailed events which he would describe using the EPL syntax in the respective text field. For example, the absence of a user could be described (in a simplified manner) by the following rules:

- motion has been detected
- door has been opened, then closed
- no motion detected for about 2 minutes

The developer would now specify these rules in EPL, whereas the IDE supports him by displaying the list of available event channels from which he could choose. In our example, the EPL query could look as follows:

```
SELECT 'userabsent' FROM
PATTERN [everyMotion (location. inside = true)
□> Door ( action='open')
□> Door ( action=' close ')
□> (NOT Motion (location .inside=true)
WHERE timer : within (2 minutes))]
```

If this event pattern is detected, the Obligation Framework will execute the action which has been specified in the checkbox list below. We assume the developer only wants to log this event, so he just checks the *Logging* action.

Note that – depending on the specified event pattern – high volumes of events can be triggered and defining time-consuming actions to them may lead to significant performance losses in the middleware. For example the event pattern "SELECT * FROM AllEvents" will generate a huge amount of events in a very short time and any other action than simply logging these will result in dramatic computation overhead in the middleware (not in the IDE, of course). Developers are advised to limit the output rate of events by adding "*output every 10 seconds*" to their rule, for example.

Now that the developer has finished defining this event pattern, he simply clicks on "Save" and the event pattern in stored in the Obligation Framework in the Hydra middleware instance to which he is connected. By activating the checkbox besides the new entry in the EventListener view, the event pattern becomes activated (i.e. the event engine starts listening to events and triggers the specified action, if necessary).

After having defined an event for users leaving the home, the developer would define another event for the user arriving at his or her home, analogous to the previous event. Now two event patterns would appear in the EventListener view: *User leaves* and *User arrives*. From these events the developer would now define a new situation *User away*. For this, he would change to the Situation View and click on the green button to create a new situation. The new situation would be created and opened in the Situation Editor (Figure 72). In this editor, the developer would then at first name the situation "User Away", select "User left" as starting event and "User arrives" as the stopping event. On the right the list of all available access control policies in the Hydra middleware instance is displayed and the developer would be able to select those policies he wants to be active in case nobody is at home – for example he could activate policies denying access to all devices except the door lock service.

After the situation has been saved, the Obligation Framework in the Hydra middleware listens to the defined event patterns, detects the specified situation and enables or disables the access control

policies as set by the developer. These mechanisms now run solely in the middleware instance and the IDE can be disconnected without interrupting the functioning of the Obligation Framework.

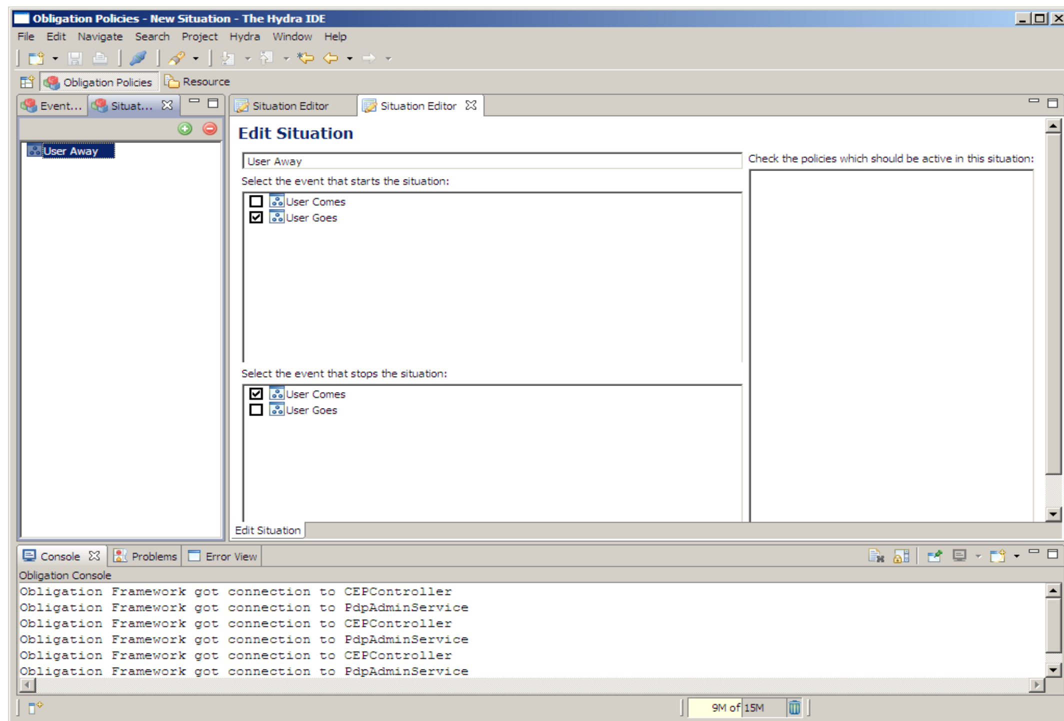

Figure 72: Situation editor (empty list of policies on the right)

7.6 Access Control Policy Framework IDE

The Hydra Policy IDE essentially corresponds to the Policy Administration Point in the XACML processing model. The Policy IDE serves as both an end-user and developer-user interface to guide and assist them in writing XACML 1.x policies, represented as XML documents, using a customised schema-backed editor (in Eclipse) to provide content-assistance and runtime validation of policies being authored. It also provides the ability to manage a user's PDPs on the Hydra network, with a PDP browser that shows existing policies published to a PDP, and their status, allowing these to be edited / deleted, and have new policies published to them.

The Policy Management aspect of the Policy IDE brings the ability to interface with existing PDPs on the Hydra network, to manage the policies they have published to them, and their activity status. The Policy IDE also provides the means to create local policy projects, for working on a set of policies locally, without initially publishing them to a PDP. These policies can then be deployed to a given PDP as desired. Figure 73 shows the Policy Management Dashboard, on the right-hand side of the Eclipse IDE. This provides two functionalities - to either create a new policy, or to manage existing policies.

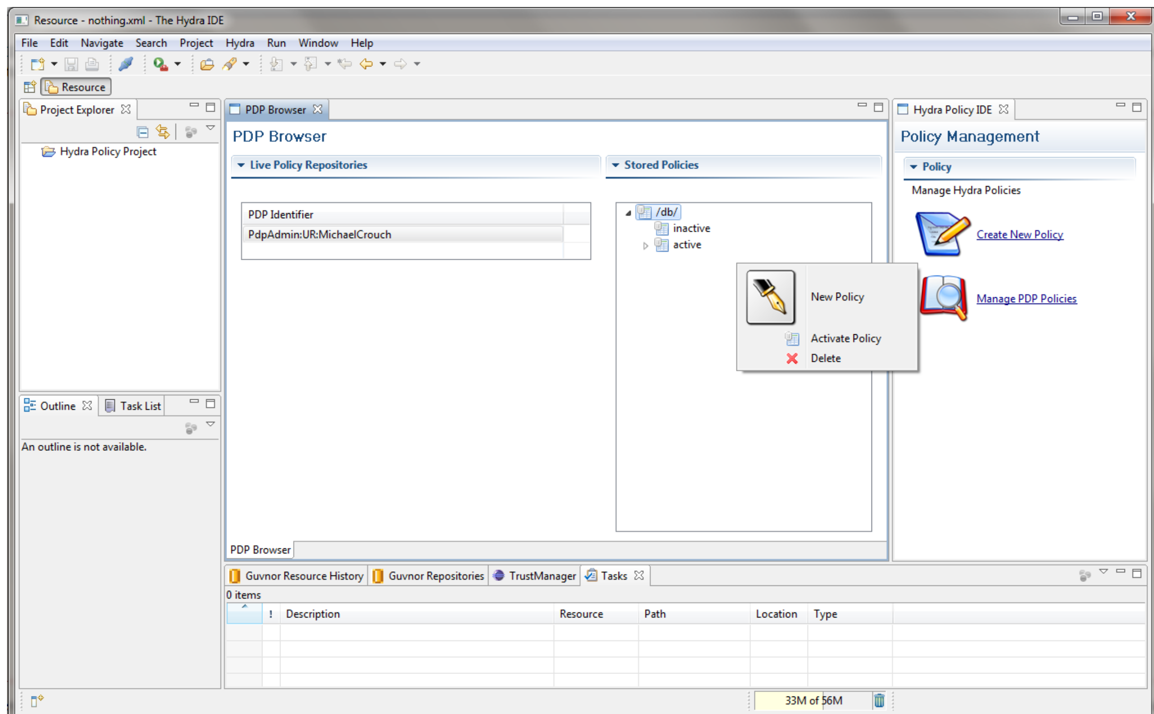

Figure 73: The Policy IDE Dashboard

Choosing the second option, opens up the PDP Browser, that lists the various PDPs on the network, which is shown in the central panel of the Eclipse IDE. Selecting a PDP here retrieves the list of policies that are stored in that PDP's Policy Repository, shown in the Stored Policies section of the PDP Browser listed by their PolicyId along with details of their activity status - either Active or Inactive. Through this view, policies can be double-clicked, which downloads them from the PDP, and opens them up in an XACML Policy Editor tab. Additionally, in this view, new policies can be created and deleted, using the Context Menu shown in Figure 73, which is activated with a right mouse-button click.

Selecting the "Create New Policy" option, from the Policy Management Dashboard (Figure 73), creates a new policy with a base template XACML policy stub initialised for the user, as shown in Figure 74.

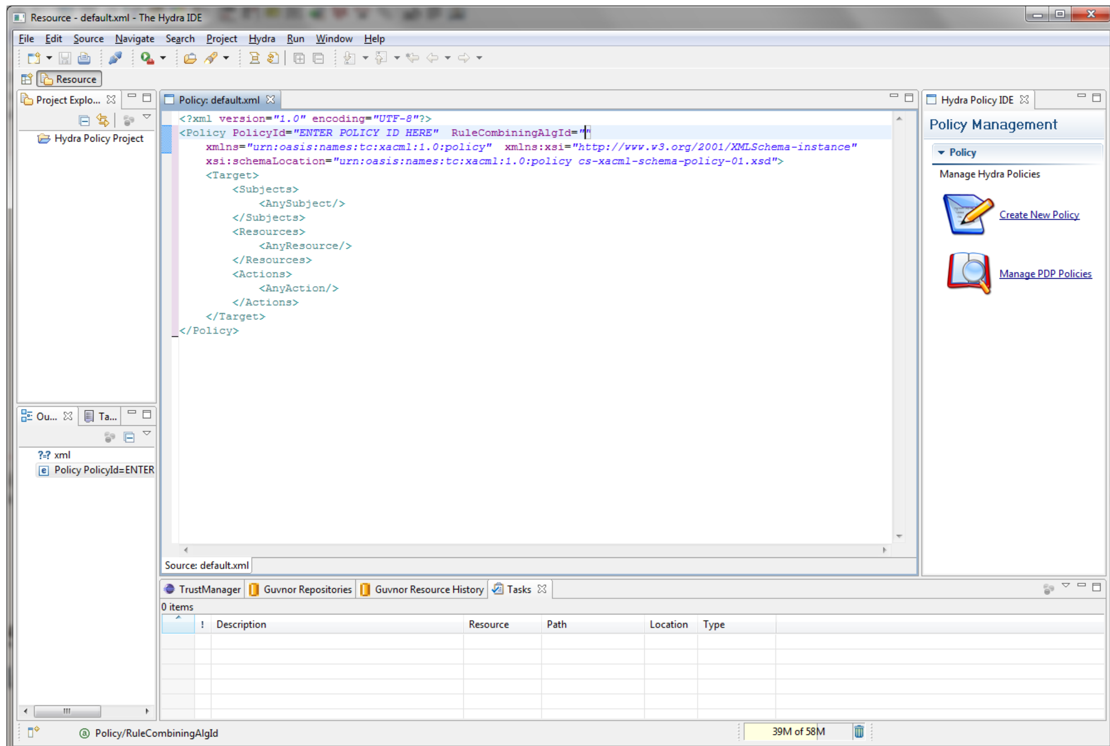

Figure 74: Creating a new Policy

The Policy IDE allows for multiple different types of templates to be created to support a developer-user in authoring policies for specific purposes. This template includes the root <Policy> element, along with the key XACML 1.x-defined attributes of the policy - PolicyId and RuleCombiningAlgId. PolicyId is the identifier of the policy, that is used to refer to the Policy externally, for retrieval and management purposes, while RuleCombiningAlgId specifies the algorithm to use for combining the different rules of the policy - for example, the algorithm 'permit-overrides' designates that if any rule defined in the policy returns the decision PERMIT, than that overrides any the result of any other rules, and is returned as the final decision. In addition, the template shown has some extra attributes defined in the root element, designating the XACML schema used to define the policy.

The Policy IDE uses a Content Assist Processor to provide a number of functionalities to the Policy IDE, to assist in the creation and editing of XACML 1.x Policies. Content Assist is activated by pressing the Ctrl and Space buttons together, which is the standard convention for requesting content assist in the Eclipse development environment, which then lists a selection of targeted proposals relevant to the point at which assistance is requested. Selection of a proposal can either insert a pair of XML tags, with required attributes specified, or insert attribute or element values. The functionalities include:

- XACML Attribute Assistance
- XACML Structure Assistance

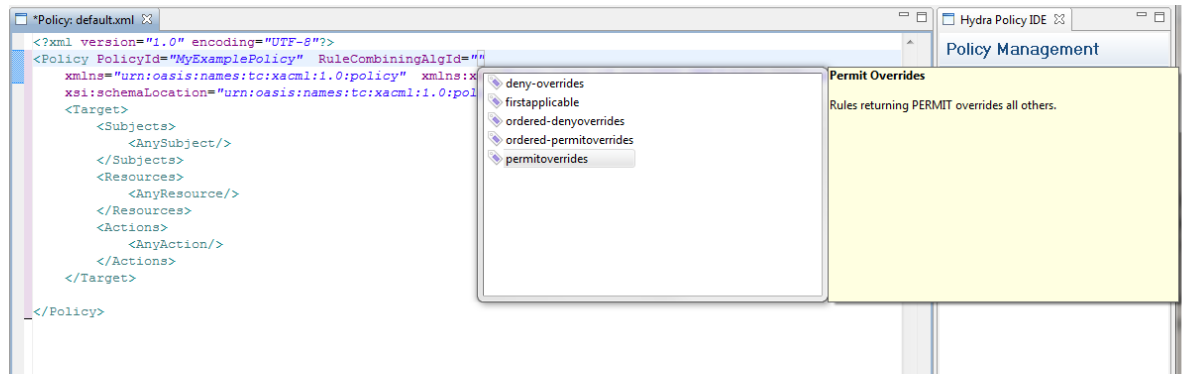

Figure 75: Content Assist for selecting Rule Combining Algorithm

Figure 75 demonstrates the Context Assist proposals for the RuleCombiningAlgId attribute of the Policy element, with each proposal being accompanied by a brief description as to its purpose.

These proposals are retrieved from a local database of algorithms that are configured with the PDP. Mostly, these are core XACML-defined algorithms, functions and attributes, but the Policy IDE can also be informed of additional, non-standard functions and attributes, that are unique to the Hydra environment, providing the ability to use components of the Hydra Middleware as input for making access control decisions. As described previously, these additional functionalities are provided to the PDPs through implemented PIPs, that register the attributes and functions of IDs (that they can resolve) with the PDP, such that the ability to do so is then possible.

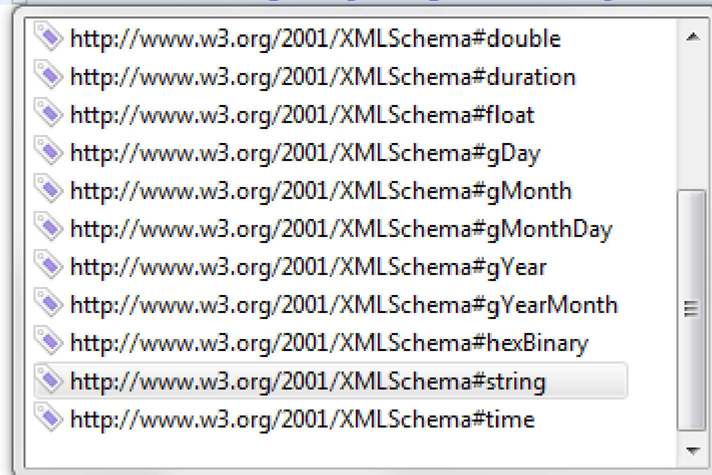

Figure 76: Content Assist selecting Data Type

Figure 76 shows another example of using the Content Assist to set the value of an XACML attribute - this time the DataType attribute of the ResourceAttributeDesignator. The proposals given are taken from the core XMLSchema, which defines the notations for referencing data types. Again, custom data types can be created and supplied to the PDP through the PIP extension, which can then be referenced in policies.

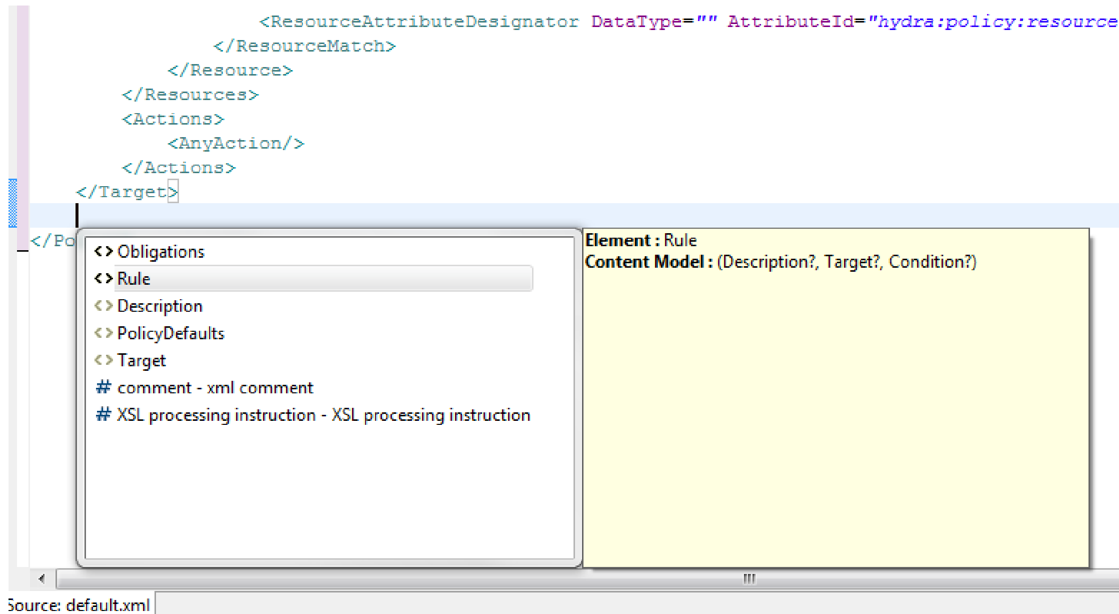

Figure 77: Content Assist in adding new root Policy XACML Elements

Figure 77 demonstrates using the Content Assist to add new elements to the policy. XACML 1.x policies have three core elements - Target, Rule and Obligations. Target specifies the context of the policy - the particular subject, resource or action it governs. In the example, the Target element defines that the policy is valid for any requests with a Resource attribute with and id of "hydra:policy:resource:pid", having a value of "MyExampleResource.PID". This id represents the persistent id that is attached to the CryptoHID created to register the requesting service on the network. Rule elements each specify a set of conditions that must evaluate to true for the defined Effect of the Rule to be returned. Obligations specify a set of obligations that are returned to the PEP depending on the decision returned. Figure 77 shows a request for content assistance underneath the Target element. It gives two proposals, the Obligations and Rule elements, while also showing proposals that are possible, but not required.

When adding new elements to the document, through Content Assist, any required attributes specified in the XACML 1.x schema are also added, some with default values - if they are also defined by the schema. For example, when adding a Rule element, the attributes RuleId and Effect are automatically added. The possible values of the Effect attribute are defined in the XACML 1.x schema as being only either Permit or Deny - anything else would cause an error (Figure 78), and as such the value Permit is entered by default.

Figure 78: XACML policy with an invalid attribute value

XACML policies authored using the Policy IDE are subject to validation by the XACML 1.x schema, with any errors being reported with their location being underlined, and a tooltip that pops up with a description of the error. This is demonstrated in Figure 78 with an invalid attribute value for the Effect attribute. Typically, the validation errors will be due to structuring errors or missing elements. To ensure schema validation, the schema being used must be specified in the root node of the policy. This schema, which is automatically added into the root node of the policy on creation, is internal to the Policy IDE, so only needs to be referred to by name in order for the system to use it.

```
<?xml version="1.0" encoding="UTF-8"?>
<Policy PolicyId="MyExamplePolicy"  RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:permitoverrides"
    xmlns="urn:oasis:names:tc:xacml:1.0:policy"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="urn:oasis:names:tc:xacml:1.0:policy cs-xacml-schema-policy-01.xsd">
    <Target>
        <Subjects>
            <AnySubject/>
        </Subjects>
        <Resources>
cvc-complex-type.2.4.b: The content of element 'Resources' is not complete. One of '{"urn:oasis:names:tc:xacml:
1.0:policy":Resource, "urn:oasis:names:tc:xacml:1.0:policy":AnyResource}' is expected.
                                                                                    Press 'F2' for focus
            <AnyAction/>
        </Actions>
    </Target>
    <Rule Effect="Permit" RuleId="IsMyExampleSubject">
        <Condition FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-one-and-only">
                <SubjectAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string" AttributeId="hydra:policy:subject:pid"/>
            </Apply>
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">MyExampleSubject.PID</AttributeValue>
        </Condition>
    </Rule>
    <Rule Effect="Deny" RuleId="CatchAllDeny"/>
```

Figure 79: XACML Schema Validation reporting errors

Figure 79 demonstrates the schema validation reporting errors with the XACML structure, as the elements inside the Resources node of the Target element have been deleted. It reports that the policy is not complete, and suggests what is missing.

7.7 Device Application Catalogue IDE

The Device Application Catalogue provides the integration of the DAC Browser in the Hydra IDE. Using this view it is possible to connect to a local or to a remote instance of the DAC, retrieve and show in real-time the Hydra device in the local area or in a remote network, and also use the DAC discovery capabilities to create new Hydra Application.

As shown in Figure 80 the description of the device and the possibility to connect to the Wizard that helps in the creation of a new Hydra Application can be seen. The Wizard is designed with a Master/Details pattern, so selecting the device on the left, you can show the details on the right.

Right clicking on one or more selected devices, or by using the first icon on the top right the new Hydra OSGi Plugin wizard menu using the selected devices can be accessed. On the upper right, the icon with the arrow work as a start/stop button for start the DAC application in background depending on its configuration. The configuration of the DAC view passes through the DAC configuration preference page, showed in Figure 81. Here the nature of the DAC can be specified, if it's local or remote. If it is local, the path of the executable must be specified, otherwise if it's a remote DAC the URL of the remote DAC application needs to be provided. Any change on the preference page will fire changes on the model and in the other plugins that are using the DAC data.

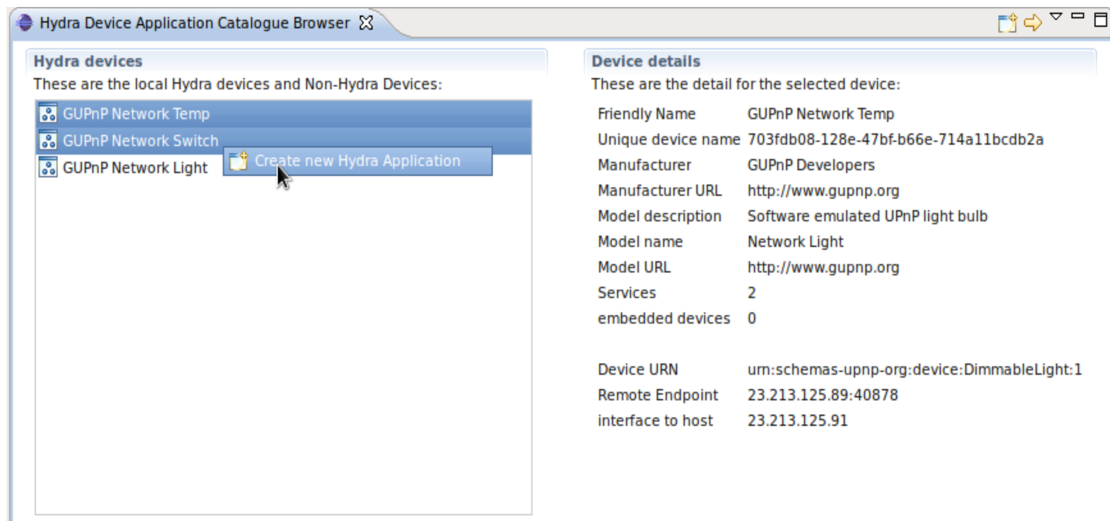

Figure 80: Device Application Catalogue View

The DAC bundle is composed of two main plugins: one charged with UI visualisation (Preference page and DAC View) and the other, called model, charged of connection and data exchange to the DAC, set the callback processes, and retrieves Hydra device information from the Device Application Catalogue. The model provides end declares the following interface:

```
public interface IDacModel {
public void startDac();
public void stopDac();
public void setDacURL(String url,boolean local, String localPath);
public ObservableList getAllGateways();
public ObservableList getHydraDevice(String gateway);
public ObservableList getUnresolvedDevices(String gateway); }
```

One relevant thing developed in the DAC model bundle is that the used collection to store and maintain the .Net DAC retrieved information is an ObservableList, which provides automatic firing change event to any other bundle registered to this observable. Using Eclipse data-binding, any plug-in that creates a dependence on this model within its views or form, is informed in real-time of any changes from the DAC (discovered a new device, removed a device, changes on a device description etc). This bundle is persistent and must be included in the Hydra IDE environment to work correctly.

Figure 81 presents a screenshot of the DAC configuration preference page as part of the Device Application Catalogue IDE.

Figure 81: DAC configuration preference page

7.8 Installing Limbo in IDE

After making the Limbo tool accessible from Eclipse (by registering the corresponding OSGi bundle) the Limbo wizard supporting the user in the Eclipse GUI controlling Limbo must be installed.

The wizard is realized as an Eclipse plugin. It is stored in the plugin directory of Eclipse using the following commands:

•LimboWizard → Export → Plug-In Development → Deployable plug-ins and fragments → Directory → local Eclipse plug-in directory.

After restarting Eclipse, the Limbo wizard can be activated by pressing the corresponding button which is indicated by the red line in the subsequent figure:

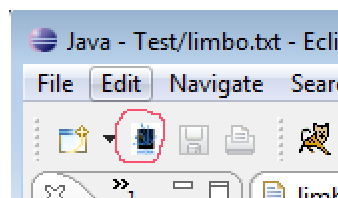

Figure 82: Limbo wizard selection in Eclipse

Using limbo in the IDE:

The Limbo wizard starting page is shown in the next figure:

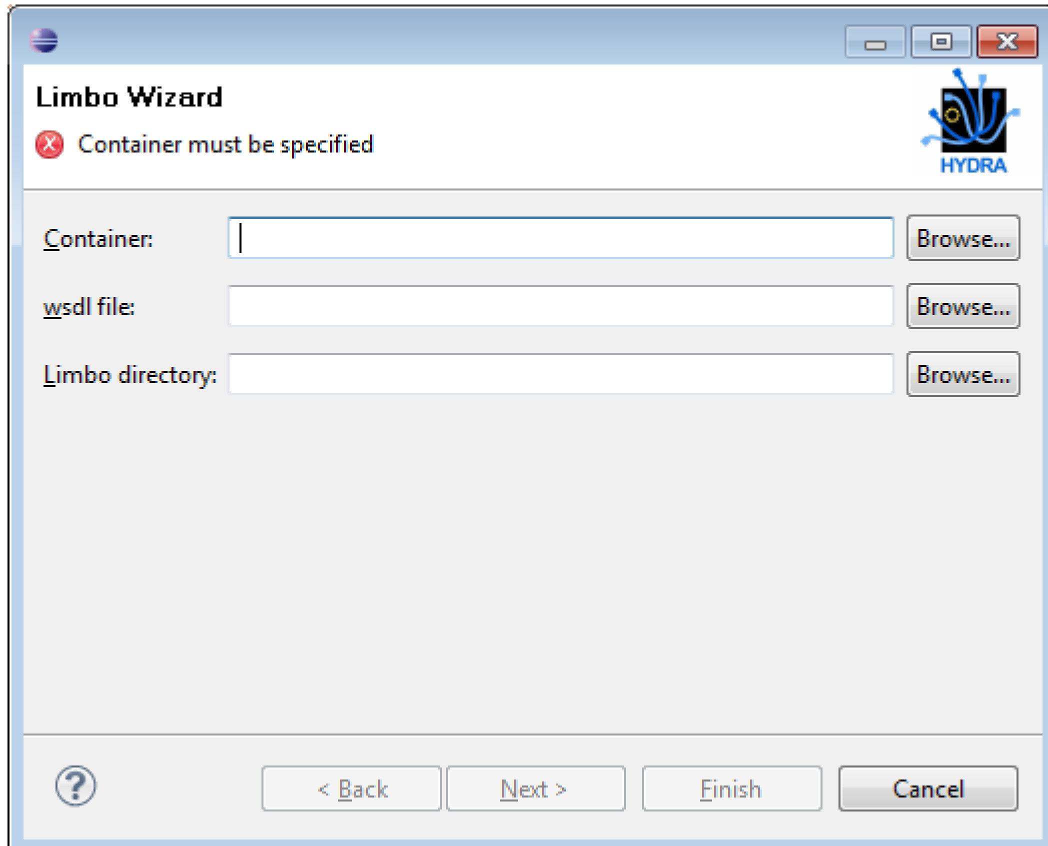

Figure 83: Limbo wizard starting point

The entry fields have the following meaning:

- Container: the name of the Eclipse project processed
- wsdl file: the path to the WSDL file
- Limbo directory: the directory of the limbo tool

The next figure shows an example:

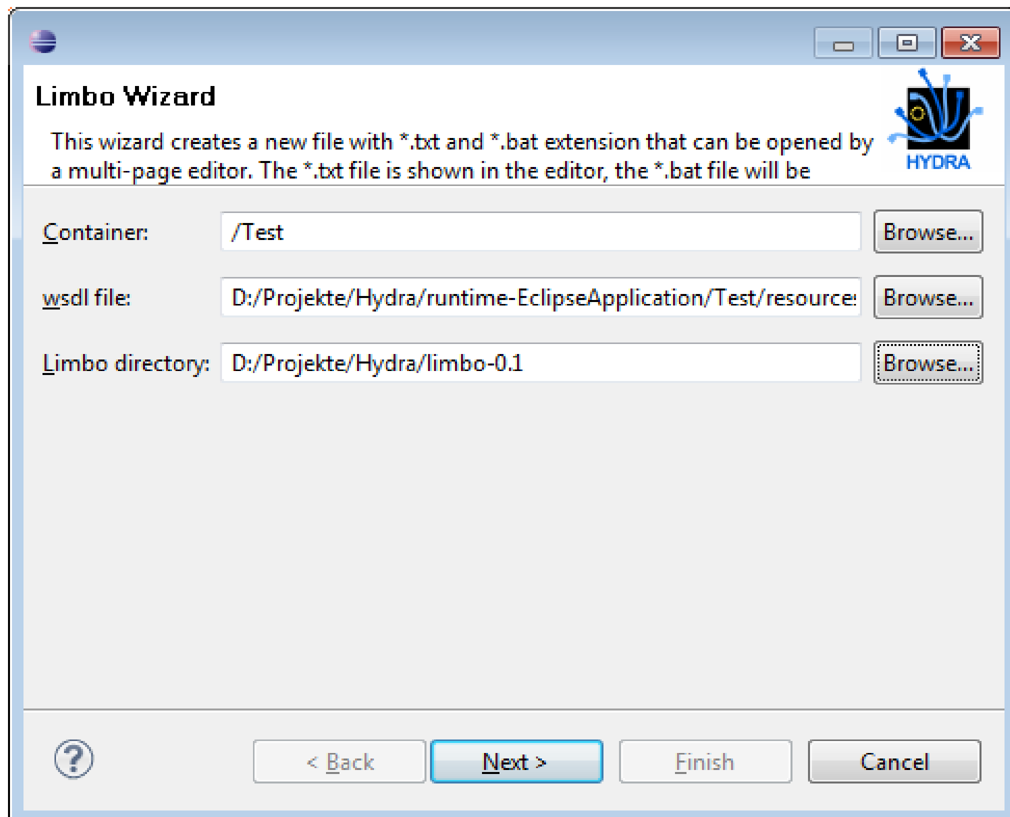

Figure 84: Limbo wizard options

The following appears after *Next* is pressed:

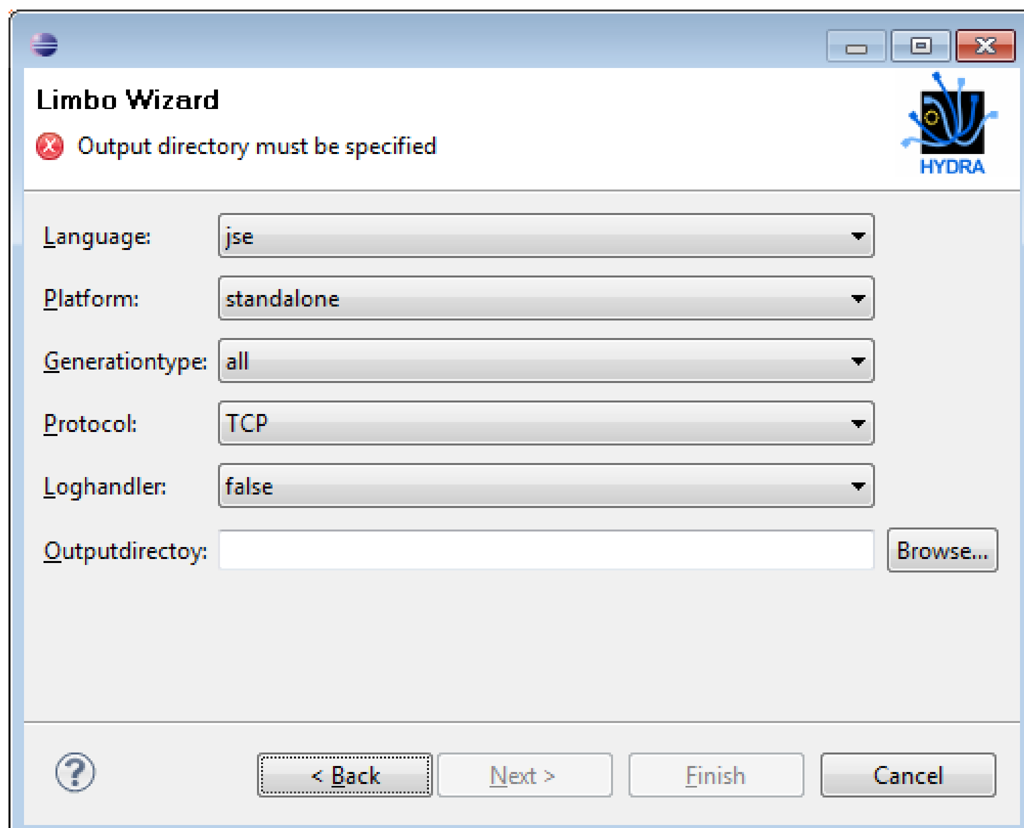

Figure 85: Limbo wizard, selecting output

The entry fields have the following meaning:
• Language: JSE or JME
• Platform: for specifying the target platform: standalone or osgi
• Generation type: server, client or both
• Protocol: TCP, UDP, or BT
• Loghandler: activating logging
• Output directory: the output directory of the generated files. This defaults to the container directory.

The next figure shows an example. Note the Finish button is enabled when the data has been entered.

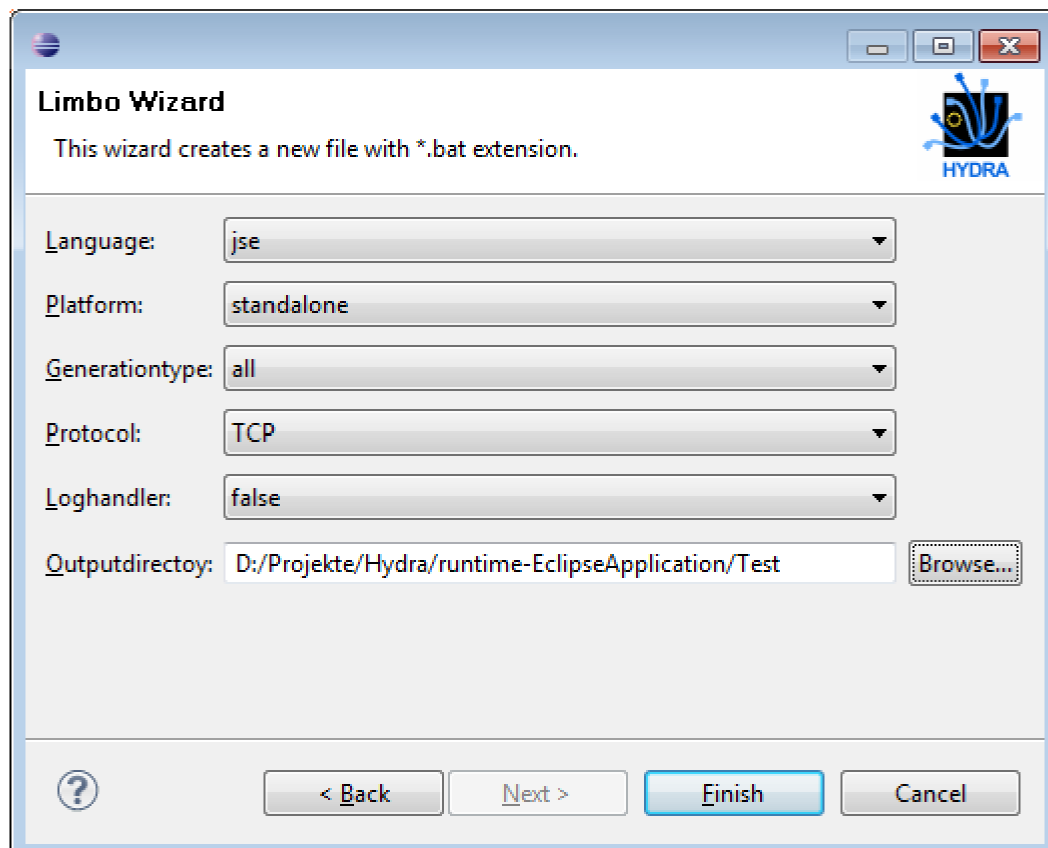

Figure 86: Limbo wizard, output directory

After pressing the *Finish* button the file *limbo.txt* is generated and subsequently executed. Here comes the content of this file:

```
@echo off
REM limbodirectory=D:/Projekte/Hydra/limbo-0.1
REM language=jse
REM platform=standalone
REM generationtype=all
REM protocol=TCP
REM loghandler=false
REM outputdirectory=D:/Projekte/Hydra/runtime-EclipseApplication/Test
REM wsdlfilename=D:/Projekte/Hydra/runtime-
EclipseApplication/Test/resources/th03r.wsdl
cd D:/Projekte/Hydra/limbo-0.1
java -jar D:/Projekte/Hydra/limbo-0.1/limbo.jar -limbo.language jse -
limbo.platform standalone -limbo.generationtype all -limbo.protocol TCP -
limbo.loghandler false -limbo.outputdirectory D:/Projekte/Hydra/runtime-
```

```
EclipseApplication/Test D:/Projekte/Hydra/runtime-
EclipseApplication/Test/resources/th03r.wsdl
```

Postprocessing

Limbo generates quite a lot of source files which have to be processed further. Subsequent to the source 2 code generation, the generated files can be modified. As this is outside the scope of limbo, this is not detailed here, but the reader should know this can be done by the tools available in Eclipse.

8. Integrated Development Environment - .Net

This section provides tutorials on how to use each component of the Hydra .NET IDE, including a general introduction to the whole IDE itself.

8.1 Creating a Basic Hydra Application

In this chapter the basic steps to create a Hydra application in a .Net environment are described and how the Hydra SDK is integrated into the Visual Studio development environment is shown.

8.2 Creating a Hydra application from a template

In Visual Studio selecting "New Project" as seen below

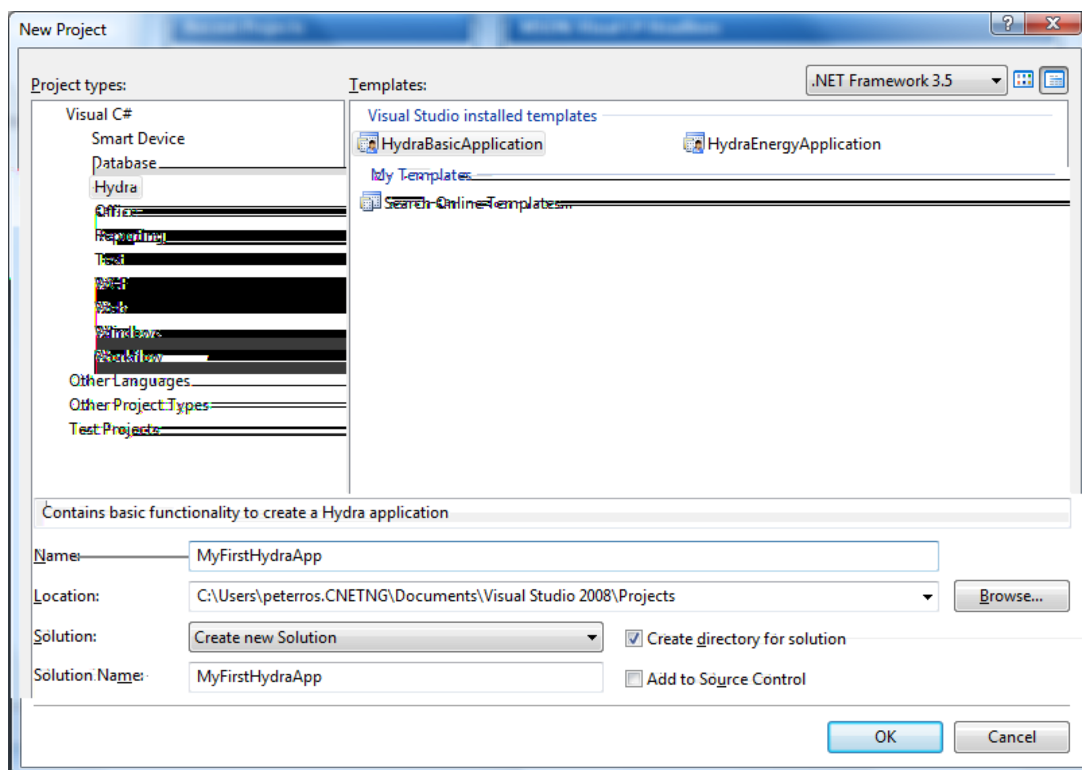

Figure 87: Template view in Visual Studio

Under the Visual C# menu the Hydra category appears. Selecting the type of Hydra application that should be develop, e.g. a basic Hydra Application.

Once the type of application is selected click OK - Hydra will create the necessary project files:

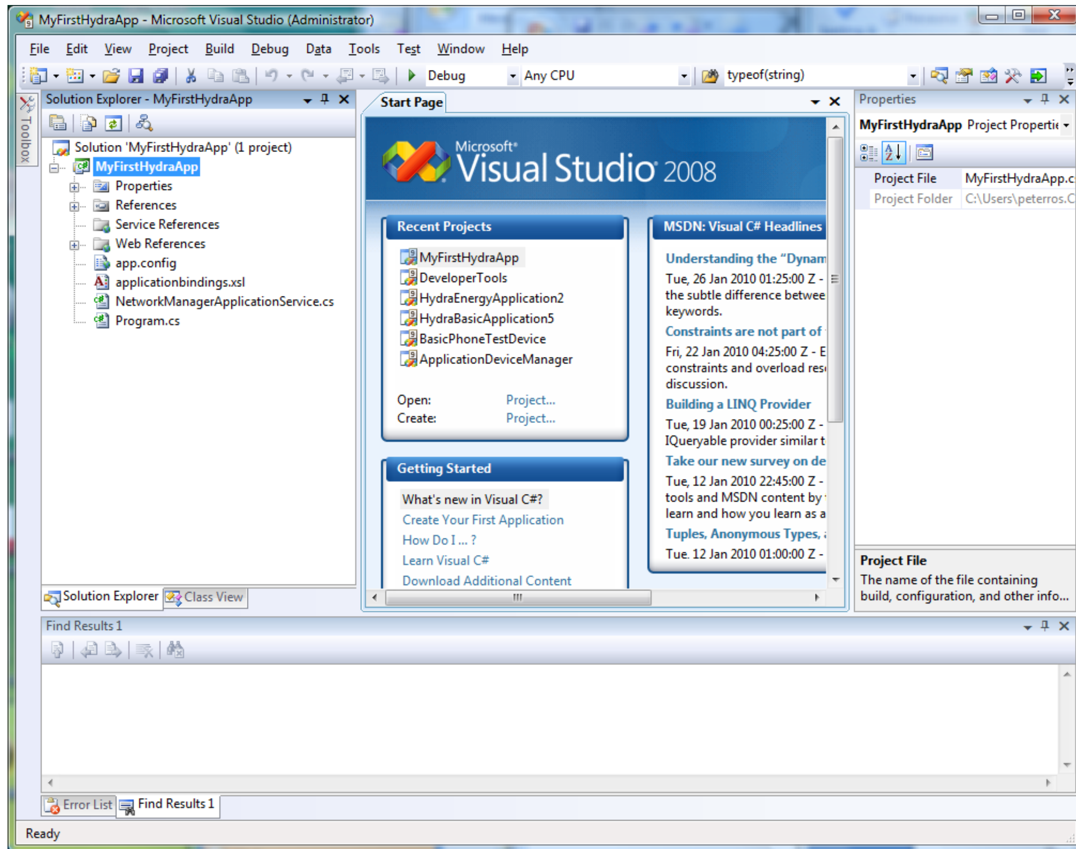

Figure 88: Auto generated files for Basic Hydra Application

The following files and references are automatically created:

- The main program file named "program.cs"
- A rule file for binding your devices to identifiers (PIDs). This file is called applicationbindings.xsl
- A Web Reference to the Application Device Manager
- A Web Reference to the Network Manager (in file networkmanagerapplicaitonservice.cs)
- A Web Reference for creating WS clients for accessing basic Hydra devices

8.2.1 Initiating the Network Manager

The first step in any Hydra application is to initiate the Network Manager in order to be able to communicate with other Hydra Managers and devices. This is done in the method

SetUpNetworkManager:

```
void SetUpNetworkManager(string url)
 {
     m_networkmanager = new NetworkManagerApplicationService();
     m_networkmanager.Url = url;

     System.Net.ServicePointManager.Expect100Continue = false;
   }
```

8.2.2 Initiating the Application Device Manager

The next step is to initiate the Application Device Manager. There are three things you need to do to initiate the Application Device Manager:

- Retrieve the Hydra ID for the Application Device Manager from the Network Manager
- Use the HID to create an endpoint URL for the Application Device Manager
- Load the device bindings into the Application Device Manager (if there is no binding provided a bindings file the Application Device Manager, will use default ways of making bindings instead).

```
void SetUpApplicationDeviceManager(string gateway, string endpoint, string appname)
 {
  m_applicationdevicemanager = new ApplicationDeviceManager.ApplicationDeviceManager();


     //Use NM to find HID for Application Device Manager
string AppDevMgrHID = 
m_networkmanager.getHIDsbyDescriptionAsString("ApplicationDeviceManager:" + gateway + 
":StaticWS");


string[] DacHIDs = AppDevMgrHID.Split(' ');


AppDevMgrHID = DacHIDs[0].Trim();


  m_applicationdevicemanager.Url = endpoint + "/SOAPTunneling/0/" + AppDevMgrHID + 
"/0/hola";



try
{
 //check if bindingfile is correct xml before sending to application device manager
  XmlDocument myDoc = new XmlDocument();
  string bindingrules = "";
  myDoc.Load("applicationbindings.xsl");

  bindingrules = myDoc.OuterXml;
   m_applicationdevicemanager.AddApplicationBinding(appname, bindingrules);
 }
 catch (Exception e)
 {

 }

}
```

At this point a connection to a NetworkManager is established and the DAC is initiated.

8.2.3 Working with devices

Once the Network Manager and Application Device Manager are initiated, working with devices can be started. A Hydra Device can be used in an application by creating a Web Service client for it. The web reference ‘HydraDevice’ should be used.

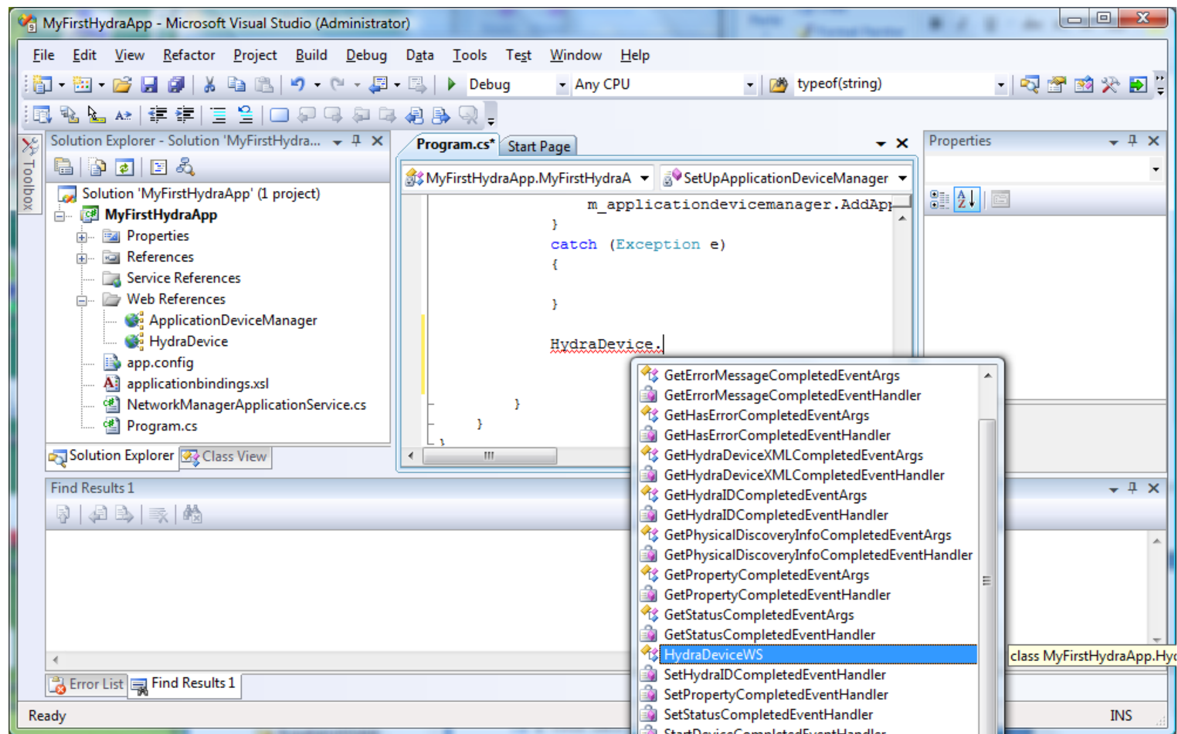

Figure 89: Creating WS clients for device

A Hydra identifier is needed to create an endpoint URL for the device (assuming the base URL in the variable "endpoint" is used) which is assigned to the identifier "PetersPhone" in the application bindings file:

```
HydraDevice.HydraDeviceWS myHydraDevice = new HydraDevice.HydraDeviceWS();
string myhid=m_applicationdevicemanager.GetHID("", "PetersPhone");

if (myhid != "")
  myHydraDevice.Url = endpoint + "/SOAPTunneling/0/" + myhid + "/0/hola";
```

Once you have established a URL for the device you can now start consuming its Hydra Services. This example only works with devices at a generic level, as Hydra Device and therefore only have access to meta data services like "GetDeviceXml":

```
string myXml = myHydraDevice.GetHydraDeviceXML();
```

8.2.4 Applications Bindings

The application bindings file (*applicationbindings.xsl*) is used to assign persistent and context dependent identifiers to devices.

The bindings are expressed as a set of xslt rules over the Hydra Device XML.

```
<binding>
<xsl:template match="upnp:device">
……….
<xsl:if test="upnp:deviceType='urn:schemas-upnp-org:hydradevice:basicswitchdevice:1' or ….
```

```
<xsl:if test="upnp:friendlyName='DiscoBall'">
   <hydraUDN>DiscoBall</hydraUDN>
   <locationdata>
     <building>CNet Office</building>
           <room>Main</room>
           <position>Table</position>
   </locationdata>
</xsl:if>

<xsl:if test="upnp:friendlyName='PetersLight' and hydra:gateway='DELL1'">
   <hydraUDN>DemoLight</hydraUDN>
   <locationdata>
     <building>CNet Office</building>
          <room>Main</room>
          <position>Table</position>
   </locationdata>
</xsl:if>


<xsl:if test="hydra:gateway='Casa Domotica'">
    <hydraUDN><xsl:value-of select="upnp:friendlyName"/></hydraUDN>
   <locationdata>
          <building>Casa Domotica</building>
          <room><xsl:value-of select="upnp:friendlyName"/></room>
   </locationdata>

</xsl:if>
……….
</binding>
```

The *hydraUDN* is the Hydra Unique Device Name, which can be derived from any of the properties in the Device XML, though normally it is set to the upnp:friendlyName. The binding combines the hydraUDN with possible location (context) data, into a PID (Preferred Identifier) for the device. Applying the binding rules to the Device XML results in the specific binding being added to the DAC where it can be used by the application code.

The developer can define the application bindings by updating the bindings XML file (an associated XML schema supports the editing). In any case the SDK also provides a default binding of devices, based on the upnp:friendlyName and without context data.

8.3 Creating an Advanced Hydra Application

One of the most common uses of the Hydra middleware is to use it for monitoring and controlling the energy consumption of physical devices.

8.3.1 Initiate Application

To create an Energy Application you follow the same steps as before:

- Select New Project
- Select HydraEnergyApplication template

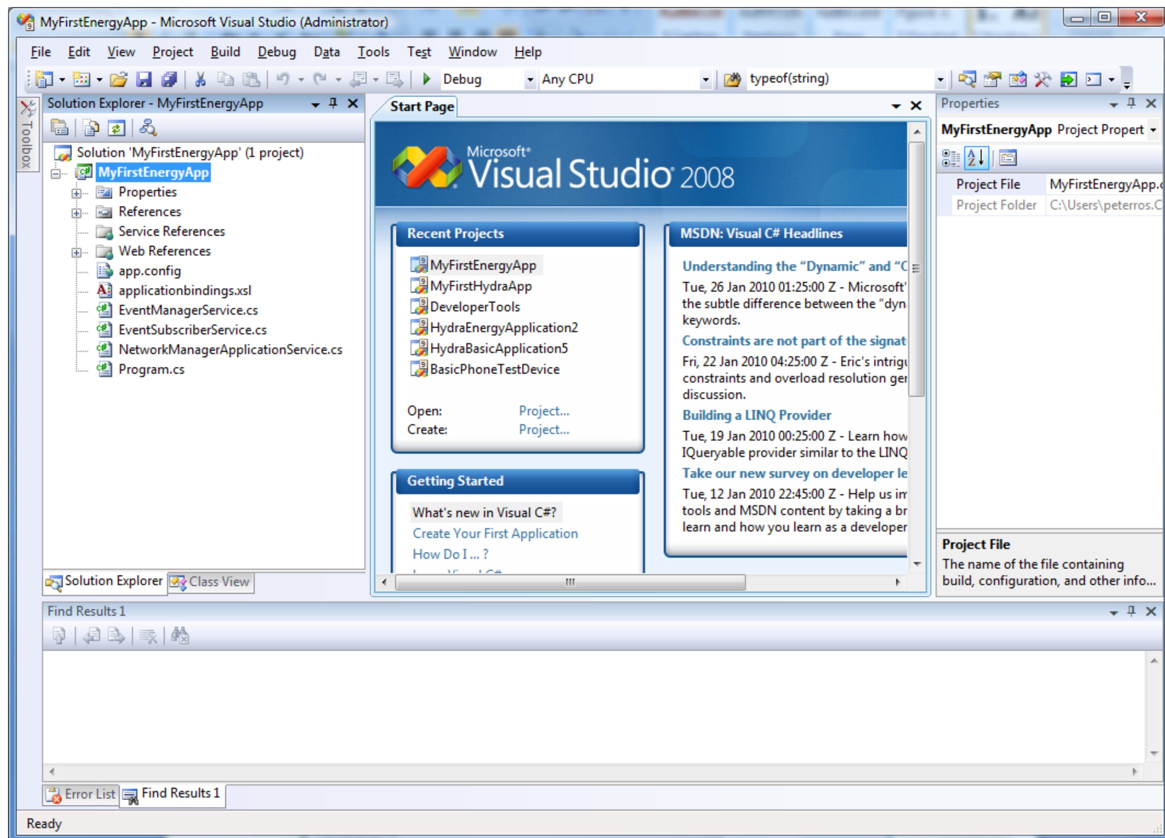

Figure 90: Energy Application Template view

The following files and references are automatically created:

- The main program file named "program.cs"
- A rule file for binding your devices to identifiers. This file is called applicationbindings.xsl
- A Web Reference to the Application Device Manager
- A Web Reference to the Network Manager (in file networkmanagerapplicaitonservice.cs)
- A Web Reference to the Event Manager (in file eventmanagerservice.cs)
- A Web Reference for creating WS clients for accessing basic Hydra devices.
- A Web Reference for creating WS clients for accessing Basic Switch and Enhanced Switch devices.
- Web Reference for accessing the Energy services of Hydra devices.

8.3.2 Searching and finding for devices

Next step is to access and control some energy consuming devices. Under the "Web References" menu two new references to "BasicSwitchDevice" and "EnhancedSwitchDevice" are shown.

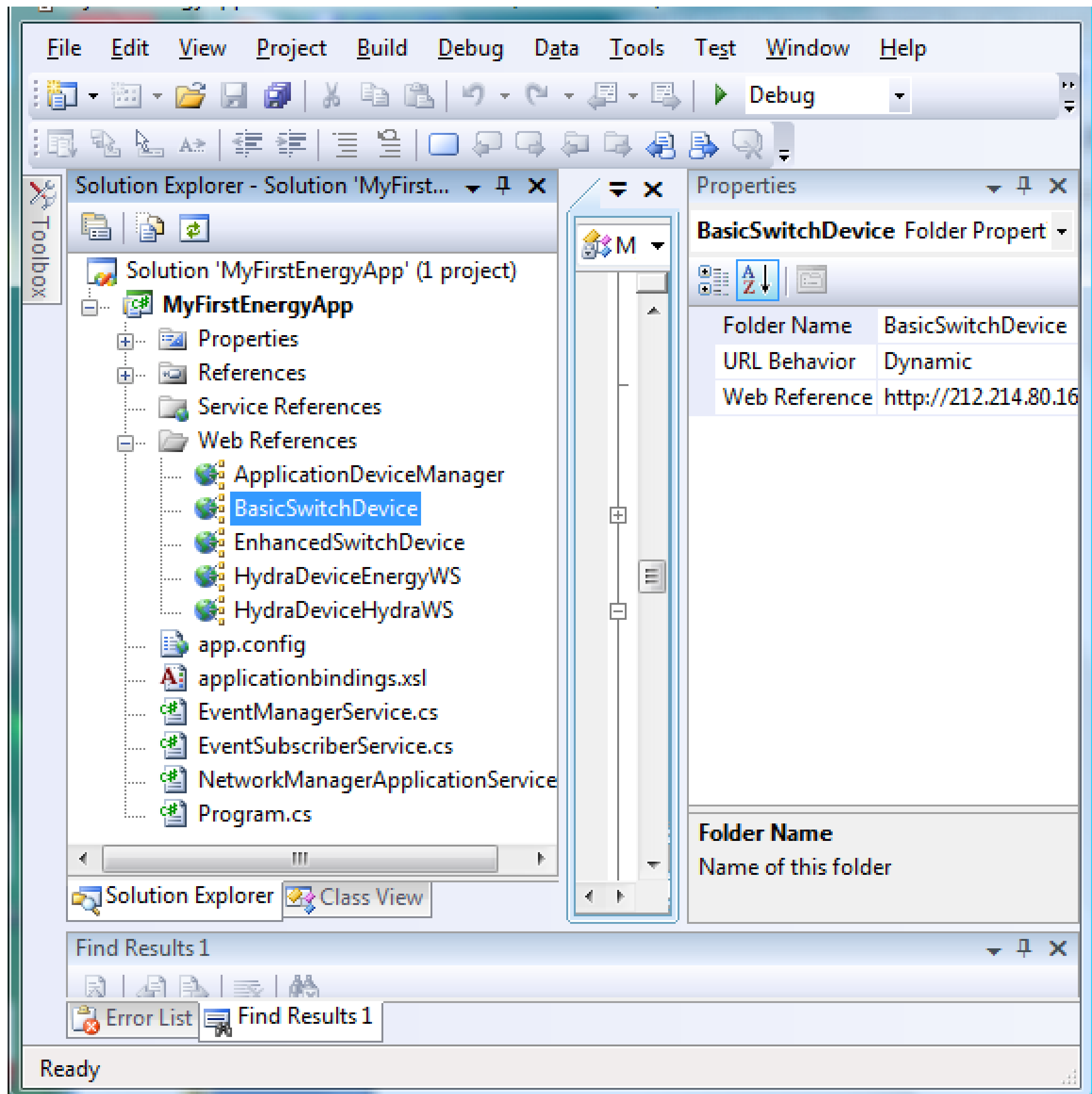

Figure 91: Selecting web references to devices

These can be used to control a particular device, but finding the device is required. Setup the Network and Application Device Manager is necessary as described in previous section (the code is already in the program.cs file).

Once this is done, querying the Application Device Manager can be used to find the devices. The knowledge of the Hydra Device XML structure and the standard XML query language "Xpath" is needed.

The following XPath statement will match each device that is of type "basicswitchdevice".

```
".//*[name()='deviceType' and .='urn:schemas-upnp-org:hydradevice:basicswitchdevice:1']"
```

If you use this statement as input to the method GetHydraURLsFromXPath of the Application Device Manager you will get a list of all discovered and active devices in a Hydra Network. Since a device might expose several web services you need to specify which one you are interested in. In this case it is "hydraidStaticWS". This method is a shortcut compared with retrieving the HID and composing the URL as it was described in the previous example.

```
public void TurnOnAllSwitchDevices()
```

```
{
string basicswitches =
m_applicationdevicemanager.GetHydraURLsFromXpath(".//*[name()='deviceType' and
.='urn:schemas-upnp-org:hydradevice:basicswitchdevice:1']", "hydraidStaticWS", "");
```

8.3.3 Invoking Device Services

The next step is to start the controlling of the devices. This code shows an example of how to turn on all switches:

```
public void TurnOnAllSwitchDevices()
{
 string basicswitches =
m_applicationdevicemanager.GetHydraURLsFromXpath(".//*[name()='deviceType' and
.='urn:schemas-upnp-org:hydradevice:basicswitchdevice:1']", "hydraidStaticWS", "");
 char[] splitchar = new char[1];

 splitchar[0] = ',';
  string[] switches = basicswitches.Split(splitchar);

  foreach (string switchurl in switches)
    {
     BasicSwitchDevice.BasicSwitchWS mySwitch = new BasicSwitchDevice.BasicSwitchWS();

     mySwitch.Url = switchurl;

     mySwitch.TurnOn();
    }
}
```

This example shows how to use the EnergyWS web service to calculate the current total effect for all running devices:

```
public int GetTotalCurrentEffect()
 {
     int returnvalue = 0;
                    string basicswitches =
        m_applicationdevicemanager.GetHydraURLsFromXpath(".//*[name()='deviceType' and
        .='urn:schemas-upnp-org:hydradevice:basicswitchdevice:1']", "hydraidEnergyWS", "");

     char[] splitchar = new char[1];

     splitchar[0] = ',';
     string[] switches = basicswitches.Split(splitchar);

     foreach (string switchurl in switches)
     {
        HydraDeviceEnergyWS.HydraDeviceEnergyWS mySwitch = new
        HydraDeviceEnergyWS.HydraDeviceEnergyWS();

      mySwitch.Url = switchurl;

      string effectstring = mySwitch.GetCurrentEffect();

      if (effectstring != "")
          {
              returnvalue = returnvalue + System.Convert.ToInt32(effectstring);
          }
      }
 }
```

8.4 Understanding the Hydra Device XML

Since all metadata and the state of a device is communicated using an XML structure it is fundamentally important to understand this structure and how it can be used. Below is an example of the Hydra Device XML for a device. The Hydra Device XML is an extension of the UPnP SCPD XML (Service Control Point Document) vocabulary. Elements with the namespace “hydra” are the Hydra-specific extensions.

```
<root xmlns="urn:schemas-upnp-org:device-1-0">
<specVersion>
  <major>1</major>
  <minor>0</minor>
  </specVersion>
 <device>
  <deviceType>urn:schemas-upnp-org:hydradevice:enhanchedswitchdevice:1</deviceType>

  <hydraidDynamicWS xmlns="hydra">0.0.0.6189708676876140718</hydraidDynamicWS>
  <energywsendpoint xmlns="hydra">http://212.214.80.144:8080/hydradevice/8619ff3a-af98-44a9-85da-
      7f5f18f7e562/energy</energywsendpoint>
  <hydraidStaticWS xmlns="hydra">0.0.0.6592261886889156134</hydraidStaticWS>
  <discoveryinfo
      xmlns="hydra"><tellstickdevice><name>PetersLight2</name><vendor>Nexa</vendor><deviceid>2</
      deviceid></tellstickdevice></discoveryinfo>
  <hydraidUPnPService_urn_schemas-upnp-org_memoryservice_1
      xmlns="hydra">0.0.0.4695383175879738995</hydraidUPnPService_urn_schemas-upnp-
      org_memoryservice_1>
  <networkmanager
      xmlns="hydra">http://localhost:8082/services/NetworkManagerApplication</networkmanager>
  <hydraUDN xmlns="hydra">PetersLight2</hydraUDN>
  <standbytime xmlns="hydra">60</standbytime>
  <status xmlns="hydra">web service initiated</status>
  <hydraidStaticWSDescription xmlns="hydra">PetersLight2:StaticWS</hydraidStaticWSDescription>
  <hydraidUPnPService_urn_schemas-upnp-org_locationservice_1
      xmlns="hydra">0.0.0.8817877591614169464</hydraidUPnPService_urn_schemas-upnp-
      org_locationservice_1>
  <hydraidUPnPService_urn_schemas-upnp-org_energyservice_1
      xmlns="hydra">0.0.0.410334127518851262</hydraidUPnPService_urn_schemas-upnp-
      org_energyservice_1>
  <hydraWSEndpoint xmlns="hydra">http://212.214.80.144:8080/hydradevice/8619ff3a-af98-44a9-85da-
      7f5f18f7e562</hydraWSEndpoint>
  <UPnPEndpoint xmlns="hydra">http://212.214.80.144:64277/</UPnPEndpoint>
  <hydraidUPnPService_urn_upnp-org_serviceId_switchservice_1
      xmlns="hydra">0.0.0.7715272012937744631</hydraidUPnPService_urn_upnp-
      org_serviceId_switchservice_1>
  <dynamicWSEndpoint xmlns="hydra">http://212.214.80.144:64277/</dynamicWSEndpoint>
  <wsendpoint xmlns="hydra">http://212.214.80.144:8080/0/EnhancedSwitchWS</wsendpoint>
  <hydraidHydraWS xmlns="hydra">0.0.0.713272519360667694</hydraidHydraWS>
  <DACEndpoint xmlns="hydra">http://212.214.80.144:8080/ApplicationDeviceManager</DACEndpoint>
 <hydraidUPnPDescription xmlns="hydra">PetersLight2:UPnP</hydraidUPnPDescription>
  <hydraidHydraWSDescription xmlns="hydra">PetersLight2:HydraWS</hydraidHydraWSDescription>
  <securityinfo xmlns="hydra"><securityInfo xmlns="hydra"><property
      name="tellstick.api.version"><value>2.1</value></property><property
      name="switch.mode"><value>2</value></property><property
      name="EncryptionProtocol"><value>None</value></property></securityInfo></securityinfo>
  <hydraidUPnPService_urn_upnp-org_serviceId_1
      xmlns="hydra">0.0.0.6339391984478104269</hydraidUPnPService_urn_upnp-org_serviceId_1>
  <hydraidEnergyWSDescription xmlns="hydra">PetersLight2:EnergyWS</hydraidEnergyWSDescription>
  <gateway xmlns="hydra">BLONDIE</gateway>
  <hydraidUPnP xmlns="hydra">0.0.0.3263501067198386232</hydraidUPnP>
  <hydraidEnergyWS xmlns="hydra">0.0.0.3952190387415366563</hydraidEnergyWS>
  <friendlyName>PetersLight2</friendlyName>
  <manufacturer>Telldus</manufacturer>
  <manufacturerURL>http://www.telldus.se</manufacturerURL>
```

```
    <modelDescription>Remote switch</modelDescription>
    <modelName>Tellstick</modelName>
    <modelNumber>X1</modelNumber>
    <UDN>uuid:8619ff3a-af98-44a9-85da-7f5f18f7e562</UDN>
</device>
</root>
```

The following element is an example of a standard UPnP element. It specifies the device type:

```
<deviceType>urn:schemas-upnp-org:hydradevice:enhanchedswitchdevice:1</deviceType>
```

This element is an example of a Hydra-specific extension. It specifies the gateway where the device is running:

```
<gateway xmlns="hydra">BLONDIE</gateway>
```

There are a number of methods that allows for searching of devices in the network. These require XPath expressions as parameter. This Xpath expression is evaluated against the Hydra Device XML for each device to decide if the match the search criteria or not.

The various elements can be grouped into categories:

PID

The hydraUDN element represents the PID (Persistent ID) that has been assigned to this particular device.

```
<hydraUDN xmlns="hydra">PetersLight2</hydraUDN>
```

Hydraids

The following elements represents the different Hydra IDs (HID) for different device services. The hydraidStaticWS is the normal HID to be used, while hydraidHydraWS is the HID to access the generic Hydra services of the devices.

Note the element hydraidUPnPService_, for each UPnP service a HID is created with the format hydraidUPnPService_serviceid (where in the service id : has been replaced with _ as in "hydraidUPnPService_urn_schemas-upnp-org_energyservice_1")

```
hydraidStaticWS
hydraidDynamicWS
hydraidHydraWS
hydraidEnergyWS
hydraidUPnPService_
```

Endpoints

The endpoint elements represent the endpoint to the device service. Normally this should not be used. Use the corresponding HID instead.

```
energywsendpoint
wsendpoint
dynamicwsendpoint
UPnPendpoint
```

Other Hydra elements

The DACEndpoint element represents the DAC that has discovered and created the Hydra Device. It "owns" the device

```
<DACEndpoint
xmlns="hydra">http://212.214.80.144:8080/ApplicationDeviceManager</DACEndpoint>
```

The gateway element represents the gateway where the device is running:

```
<gateway xmlns="hydra">BLONDIE</gateway>
```

UPnP elements

The following elements are standard UPnP elements

```
<deviceType>urn:schemas-upnp-org:hydradevice:enhanchedswitchdevice:1</deviceType>
<friendlyName>PetersLight2</friendlyName>
<manufacturer>Telldus</manufacturer>
<manufacturerURL>http://www.telldus.se</manufacturerURL>
<modelDescription>Remote switch</modelDescription>
<modelName>Tellstick</modelName>
<modelNumber>X1</modelNumber>
<UDN>uuid:8619ff3a-af98-44a9-85da-7f5f18f7e562</UDN>
```

8.4.1 Extending the Hydra Device XML

It is possible to extend the Hydra Device XML to incorporate your own meta data and state information. Simply call the method SetProperty in the Hydra WS, then you can add properties to the device which will be available in the Hydra Device XML and can be used as part of your search expressions.

Calling myDevice.SetProperty("myproperty","value1"), will create the following element in your Hydra Device XML:

```
<myproperty xmlns="hydra">value1</myproperty>
```

You can then easily select devices in the network that has myproperty="value1".

For instance the following call with get an Hydra encoded URL to the Energy WS for the all devices that has myproperty="value1".

```
m_applicationdevicemanager.GetHydraURLsFromXpath(".//*[name()='myPropety' and .='value1']" ,
"hydraidEnergyWS", "");
```

8.5 SDK components

The SDK provides a collection of templates, classes and browser tools for development of Hydra applications. The SDK, and the DDK, are intended as integrated components in the Hydra IDE, to be instantiated on two available platforms (.net Visual Studio and Eclipse).

The chosen way of integration is by mean of templates and by embedding selected tools.

8.5.1 Application Project Templates

The project templates for Visual Studio gives the developer a way of creating a Hydra application without having to write the boiler plate code necessary to set up the environment and finding the end points for interacting with Hydra managers.

There are a number of templates available which cater for some typical development scenarios. The basic difference between templates is which managers are directly available and which boiler plate code examples are provided.

8.5.2 HydraBasicApplication

Creates a standard no thrills project that connects to Hydra managers.

Managers

- ApplicationDevice Manager
- Network Manager

Device types

- Hydra Device

8.5.3 HydraEnergyApplication

This template creates a project that contains the code necessary to monitor energy consumption for a number of devices.

Managers:

- ApplicationDevice Manager
- Network Manager
- Event Manager
- Context Manager
- Storage Manager

Device Types

- Basic Switch
- Enhanced Switch

8.5.4 HydraDynamicApplication

This template creates a project that deals with devices at type level instead of binding to individual devices, for instance interacts with all BasicSwitch devices available in a Hydra network. This is useful when writing general applications which determine their devices in run time.

Managers

- ApplicationDevice Manager
- Network Manager

Device types

- None

8.5.5 HydraSensorApplication

This template creates a project that works with sensors using Events and gives the developer the necessary boiler plate code for adding which sensors and events that it should work with.

Managers

- ApplicationDevice Manager
- Network Manager
- Event Manager

Device types

- None

8.6 Tools integration

8.6.1 The DAC browser

The DAC Browser is also an integral component of SDK as part of the IDE as seen in Figure 92.

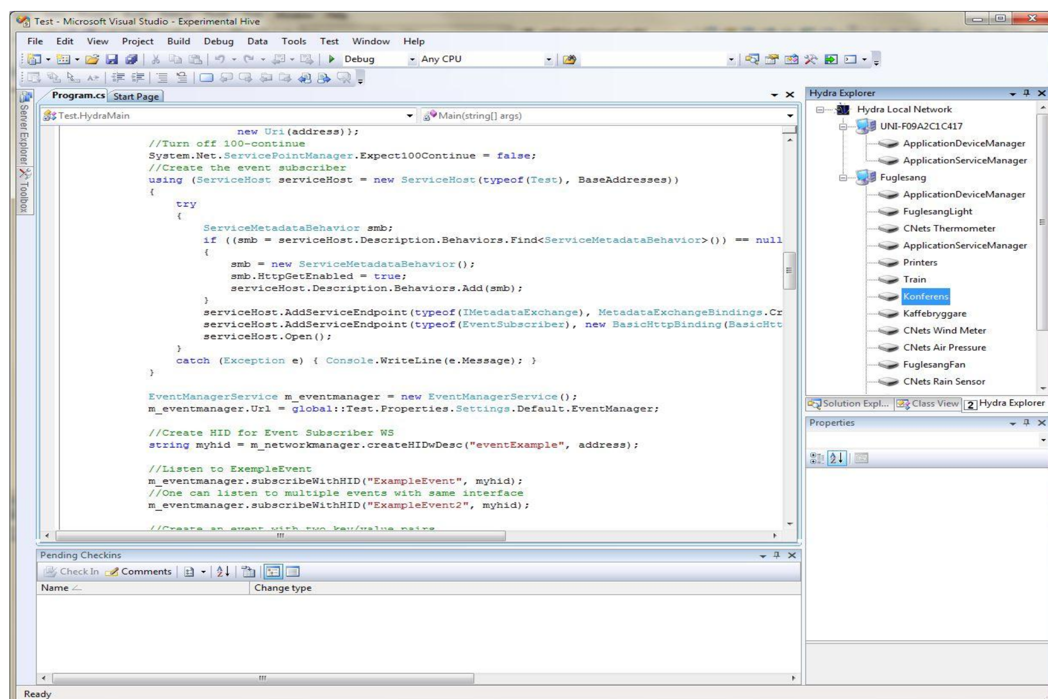

Figure 92: DAC Browser (upper right) in the IDE

It provides the same functions as the stand-alone version and in addition,

- Provides an IDE-view of all devices known to the Hydra Network
- Enables the developer to create proxies by selecting devices

8.6.2 The Device Ontology browser

Similar to the way developers can access the DAC, the Device Ontology is available in the IDE in a seamless way. The Device Ontology tool is integrated the same way the DAC is integrated.

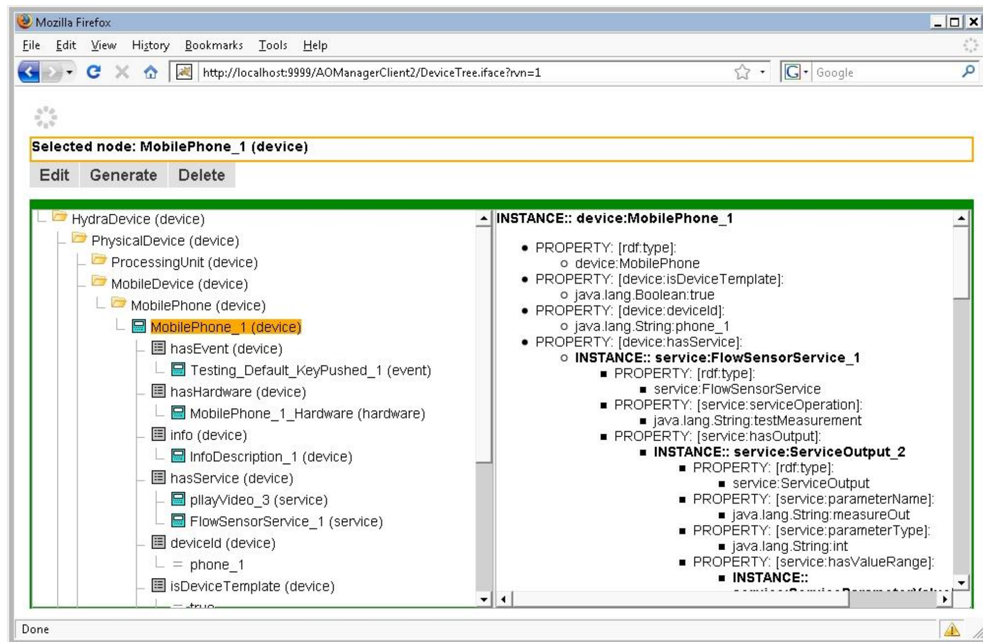

Figure 93: The web-based Device Browser

This means that a subset of the functions from the current web based ontology tools interface is available from within the IDE.

8.7 SDK Class library for .NET

The SDK Class library contains a number of ready-to-use Hydra Devices Types with corresponding web services, which are available to developers.

8.7.1 Using the .Net DDK tools

There are two main tools for creating device code for .Net in Hydra:

- Intel Service Author for UPnP Technologies
- Hydra .Net DDK tool

The example device that we will create in this tutorial is an OBEX device for a smart phone.

8.7.2 Using Intel Service Author for UPnP Technologies

This tool is used for creating the service methods and producing an SCPD that will be used as input for the final code generation.

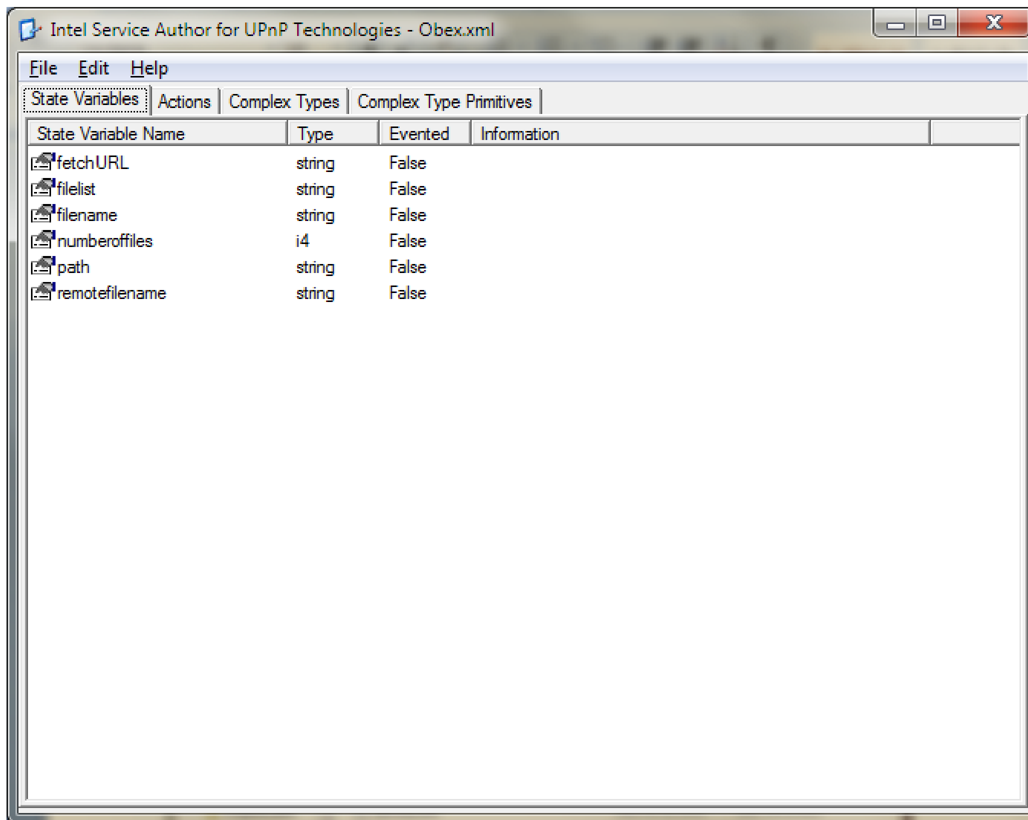

Figure 94: Producing SCPD window

The first step is to define the state variables that will be used by the service. State variables have to be defined for all Input/output parameters used in the service. In this case there are a number of state variables defined with their respective types.

The next step is to define actions, i.e. the methods that this service should support. This is carried out in the “Actions” tab.

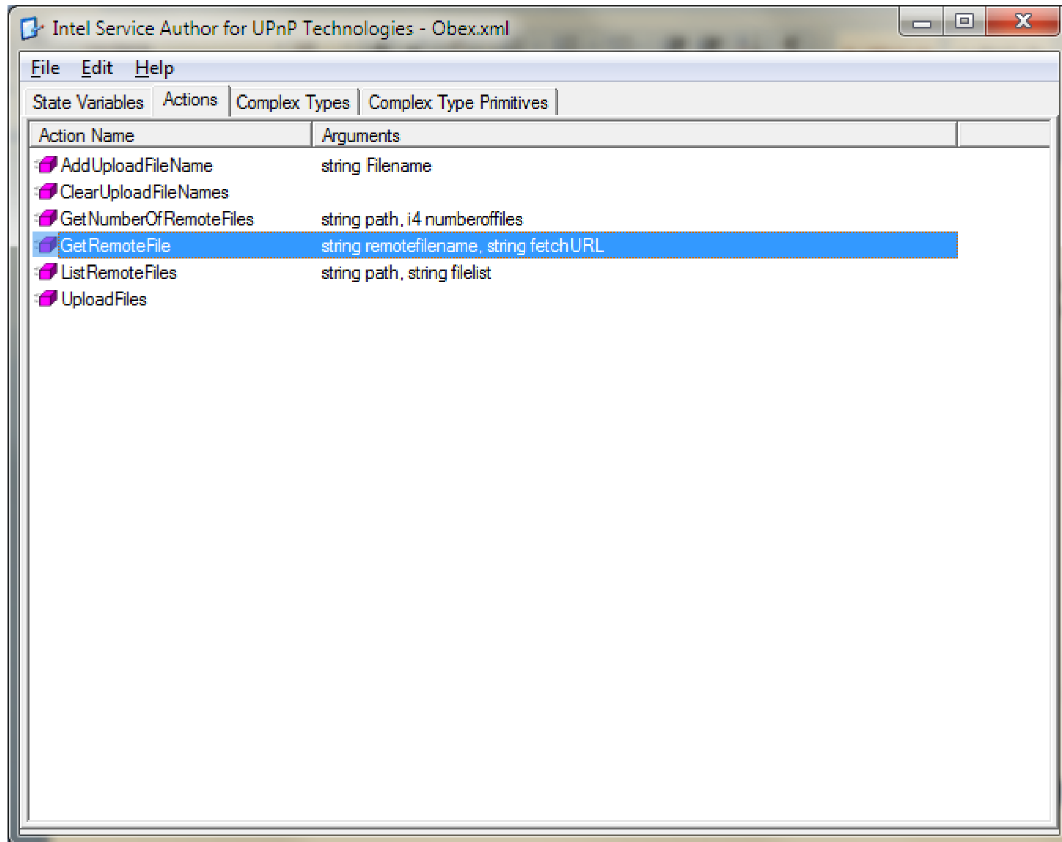

Figure 95: Action tab

There are a number of methods defined with their corresponding arguments. The methods are added using the "Action Editor" which allows for adding arguments and defining in which direction it is used.

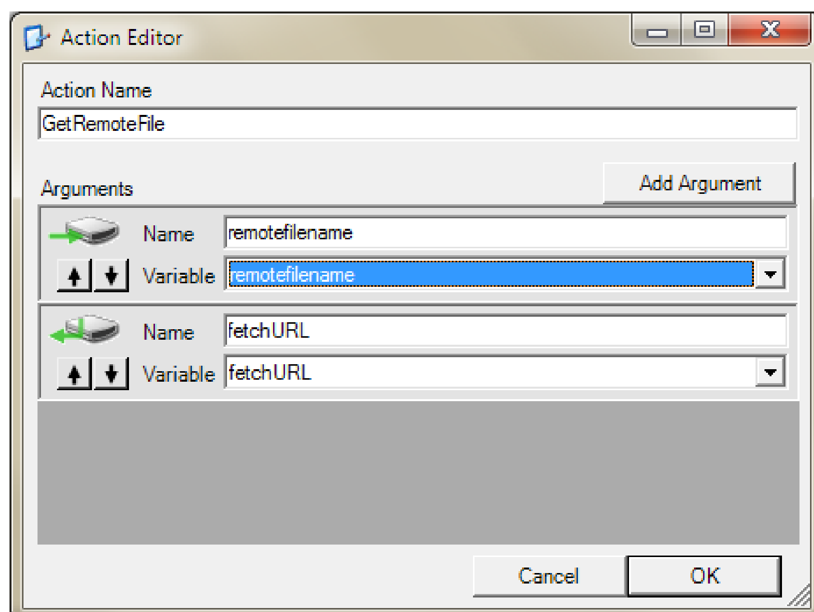

Figure 96: Action Editor

Save the SCPD to file when finished for later processing in the Hydra .net DDK tool.

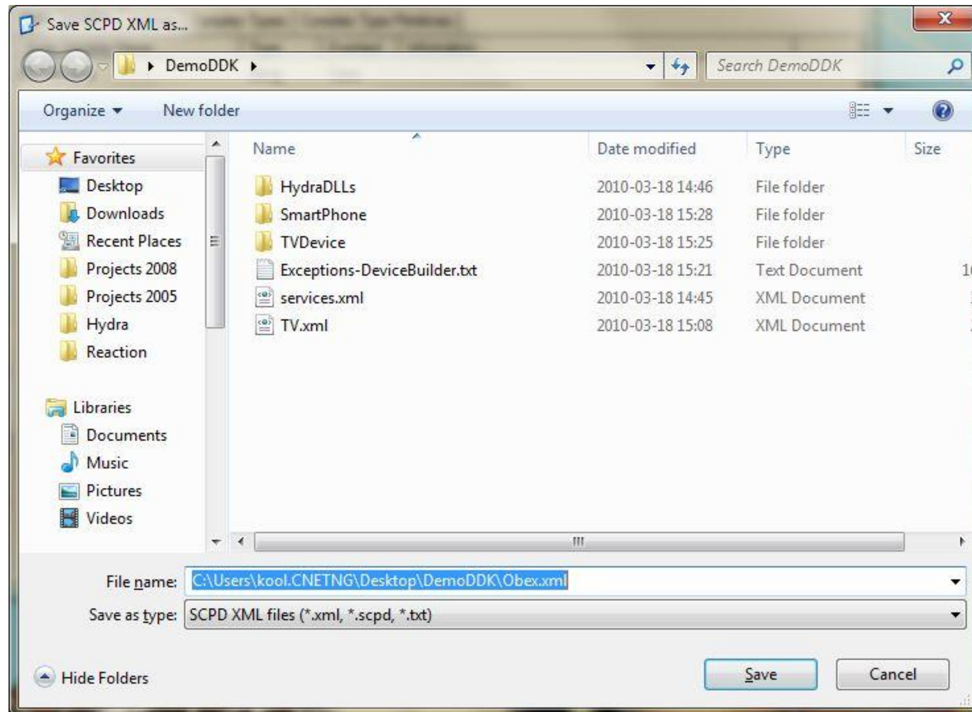

Figure 97: Save file window

8.7.3 Using Hydra .Net DDK tool

The actual code generation is carried out in the Hydra .Net DDK tool. It is also where the actual configuration of device type and other settings are done.

The first step is to "Add Device" by right clicking in the tools left pane.

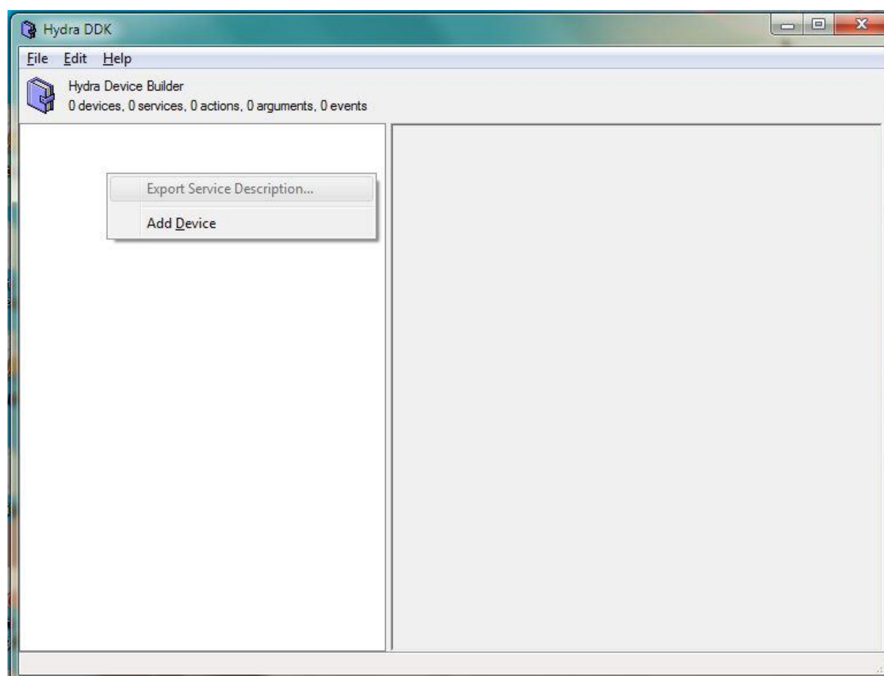

Figure 98: Add new device window

The next step is to edit the meta data for the device, i.e., device name, type, description etc.

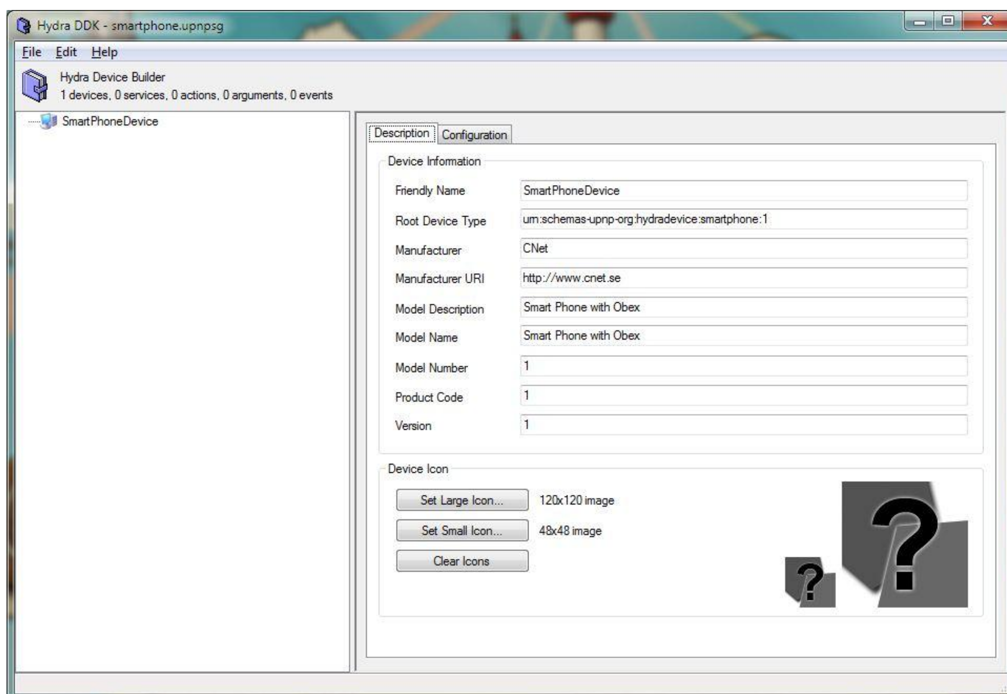

Figure 99: Window for setting name and other properties

Then one adds the service created in the previous section by right clicking on the device in the left pane.

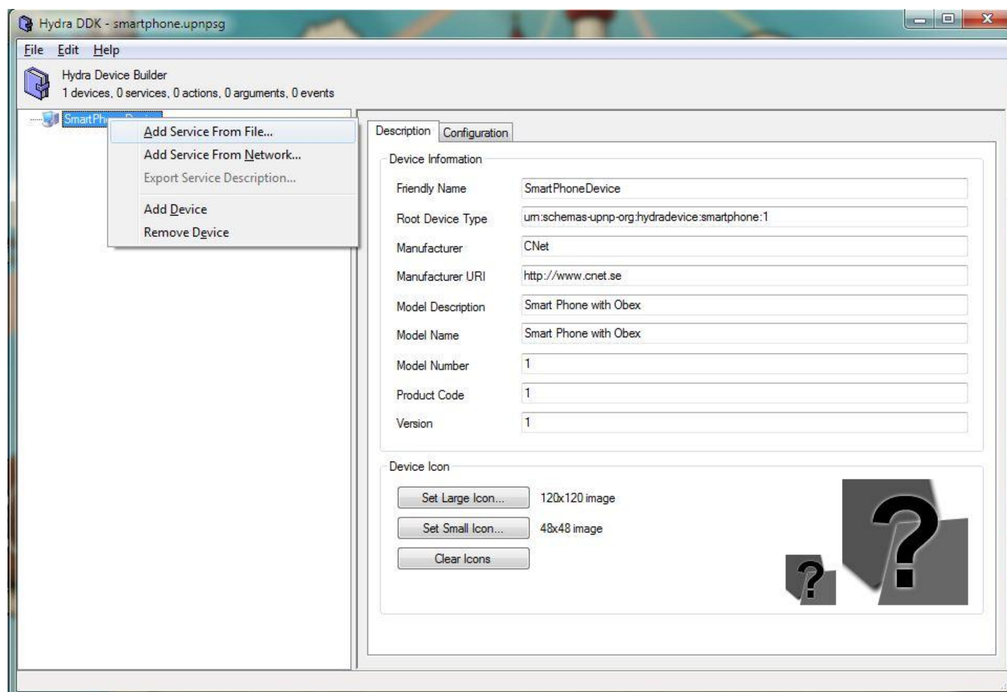

Figure 100: Adding service window

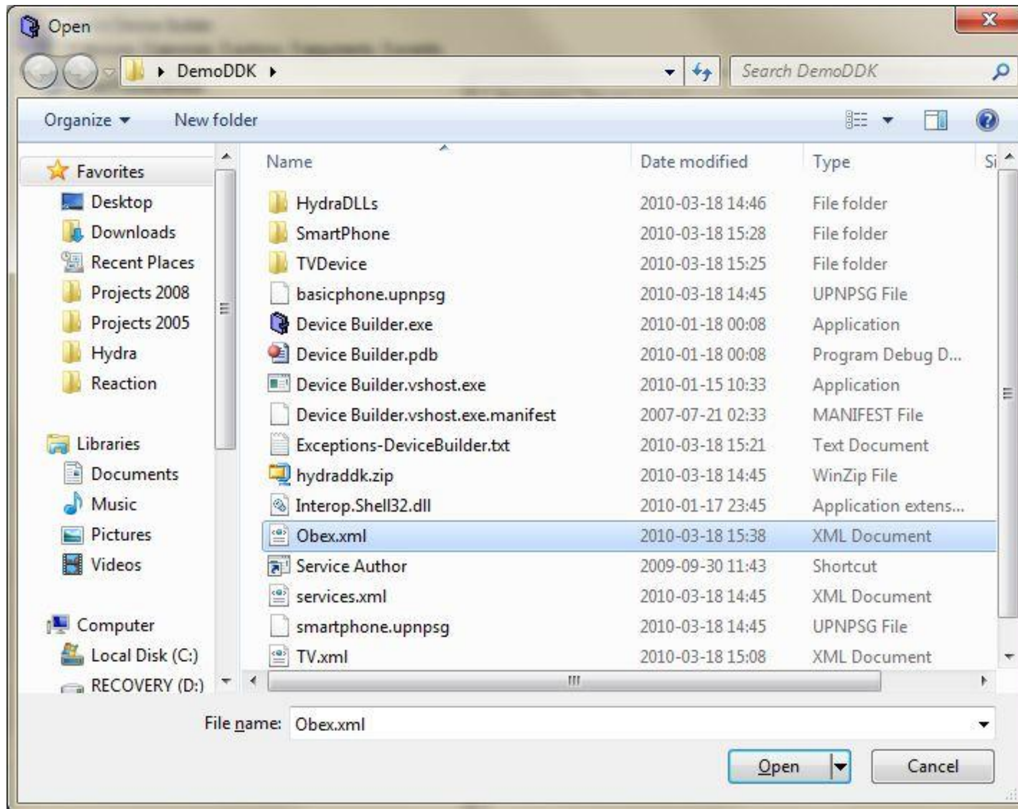

Figure 101: Choosing a file in explorer

Now we have added the OBEX service and we can see all the methods in that service.

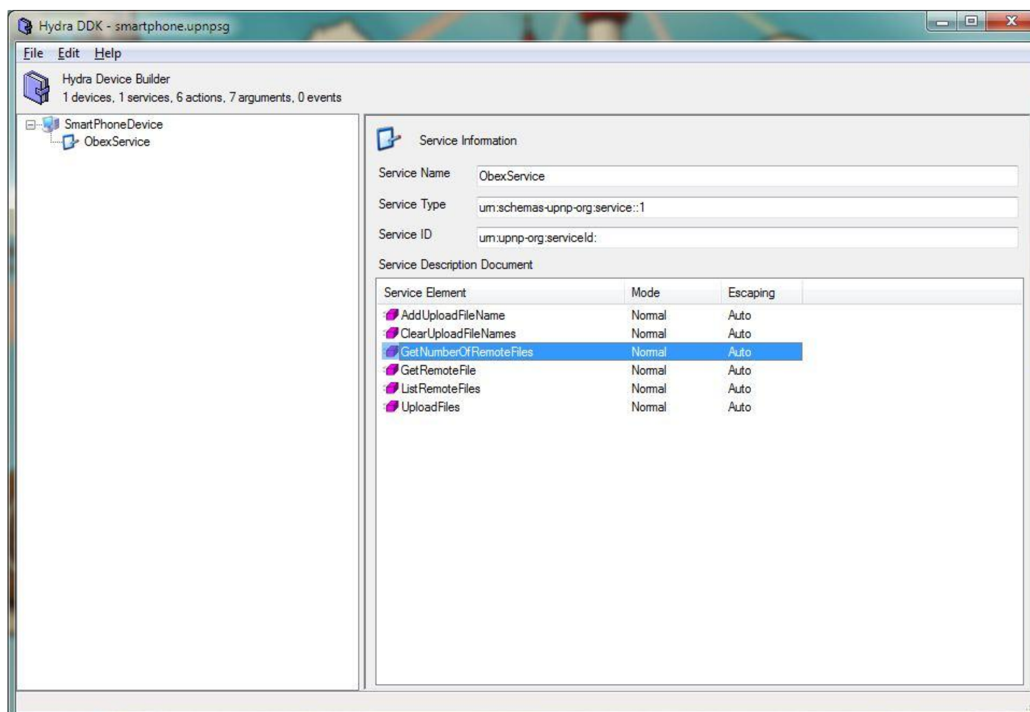

Figure 102: Obex service window

Finally generating the code for the Hydra device. Select the “File” menu and choose “Generate Hydra Device”.

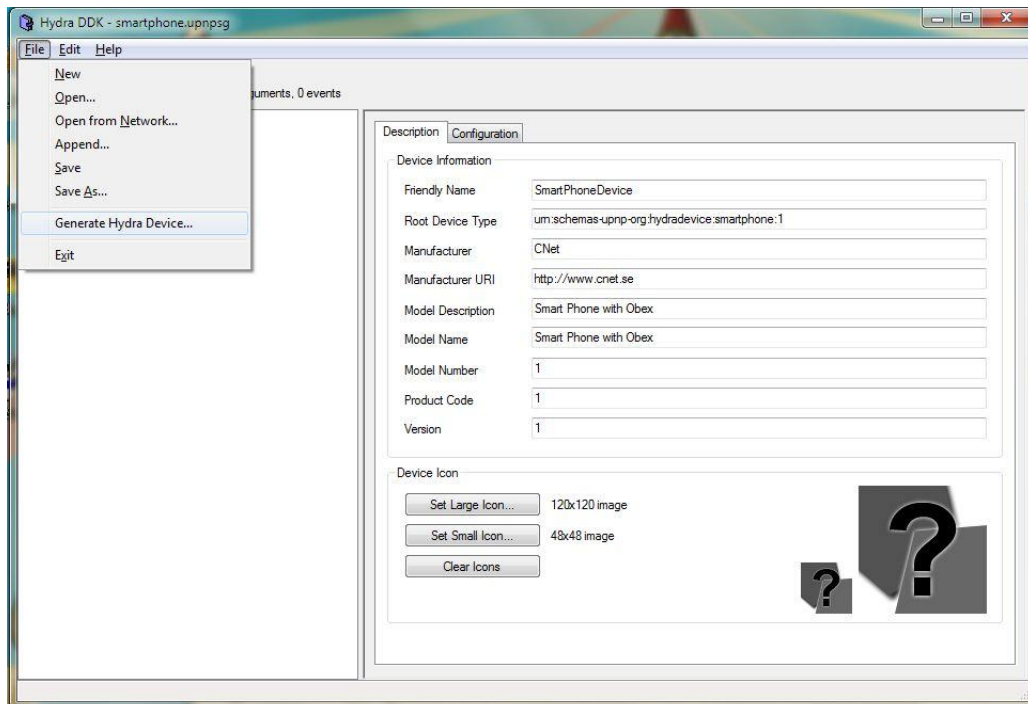

Figure 103: Generating a Hydra device dialogue

In the code generation dialogue the project name and optional Namespace for the generated code needs to be set.

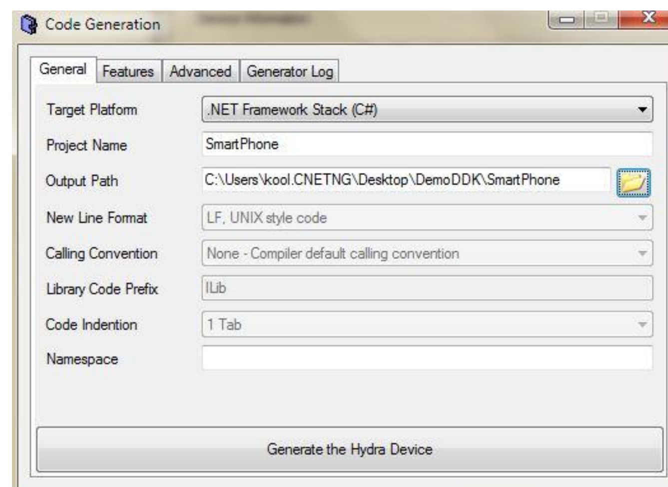

Figure 104: Code Generation Window

A complete Visual Studio project is created with the necessary Hydra references.

The Visual Studio project can be opened now.

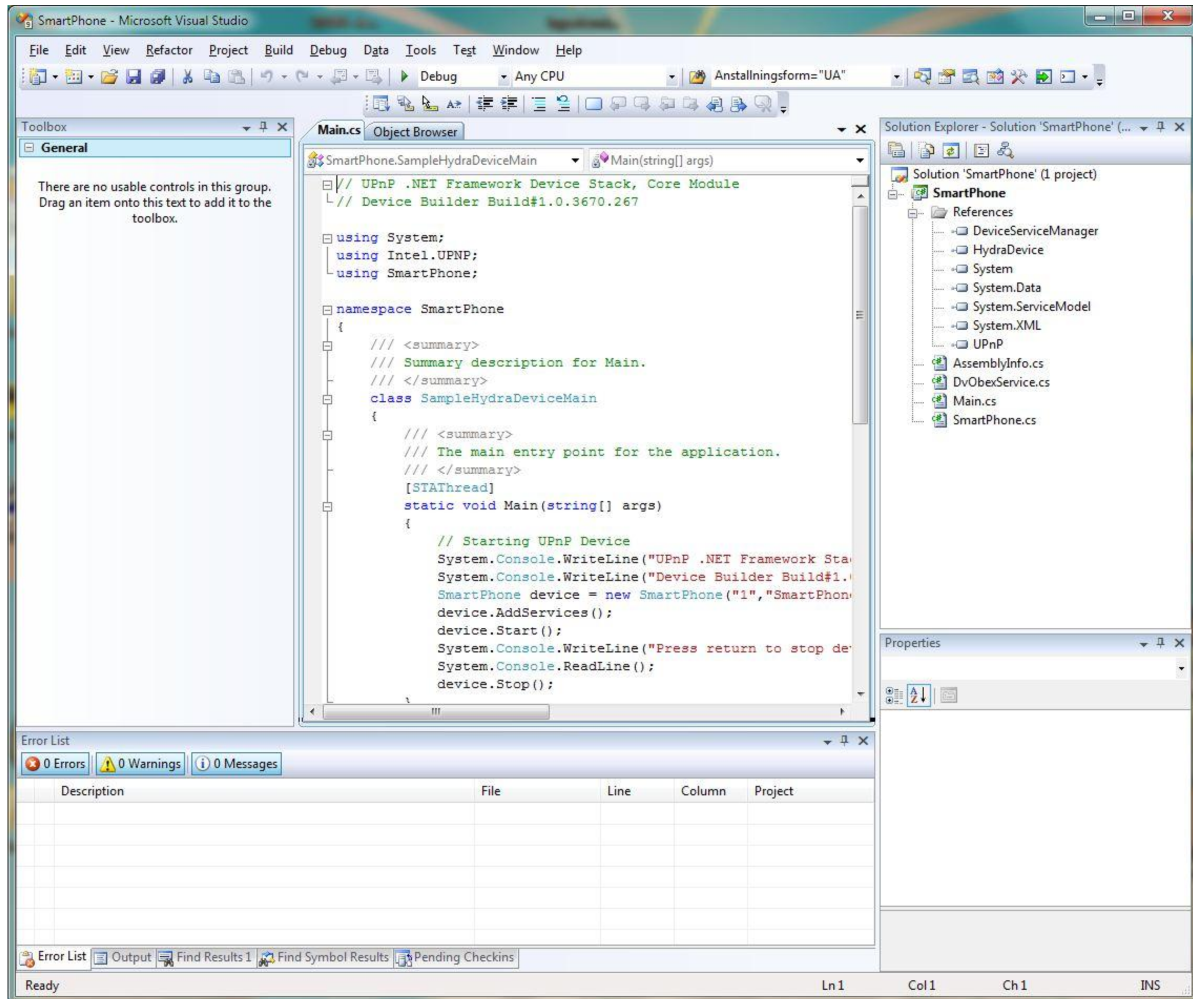

Figure 105: Hydra .Net-IDE

The device code is already runable since all methods are stubbed. The code should be changed in the stubs to carrie out the actual device communication. The location of the stubs that needs to be changed is in "Device name".cs, i.e. SmartPhone.cs in this example. But in this case we will start the device by opening the "Debug" menu and selecting "Start debugging".

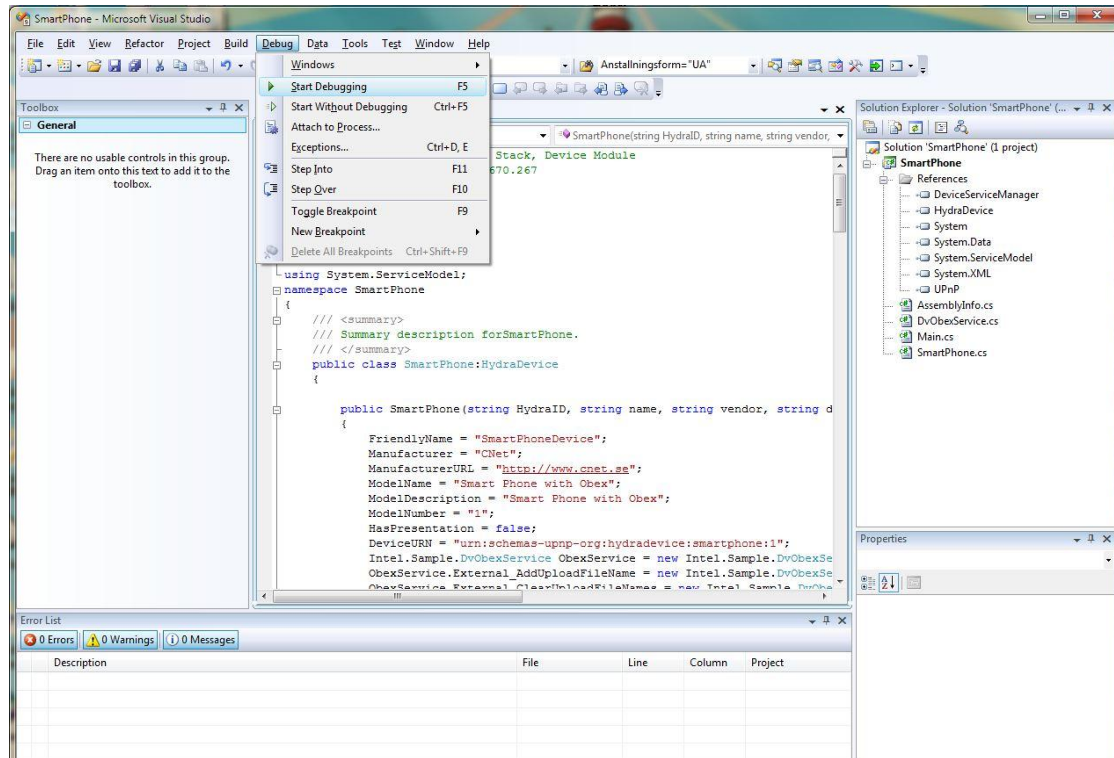

Figure 106: Hydra .Net IDE

The running Hydra DAC tool shows our newly created device with all its services. The relevant Hydra services: HydraService, EnergyService, LocationService and Memory service were created. It is shown that the Device is properly discovered and has all the Hydra properties such as HID.

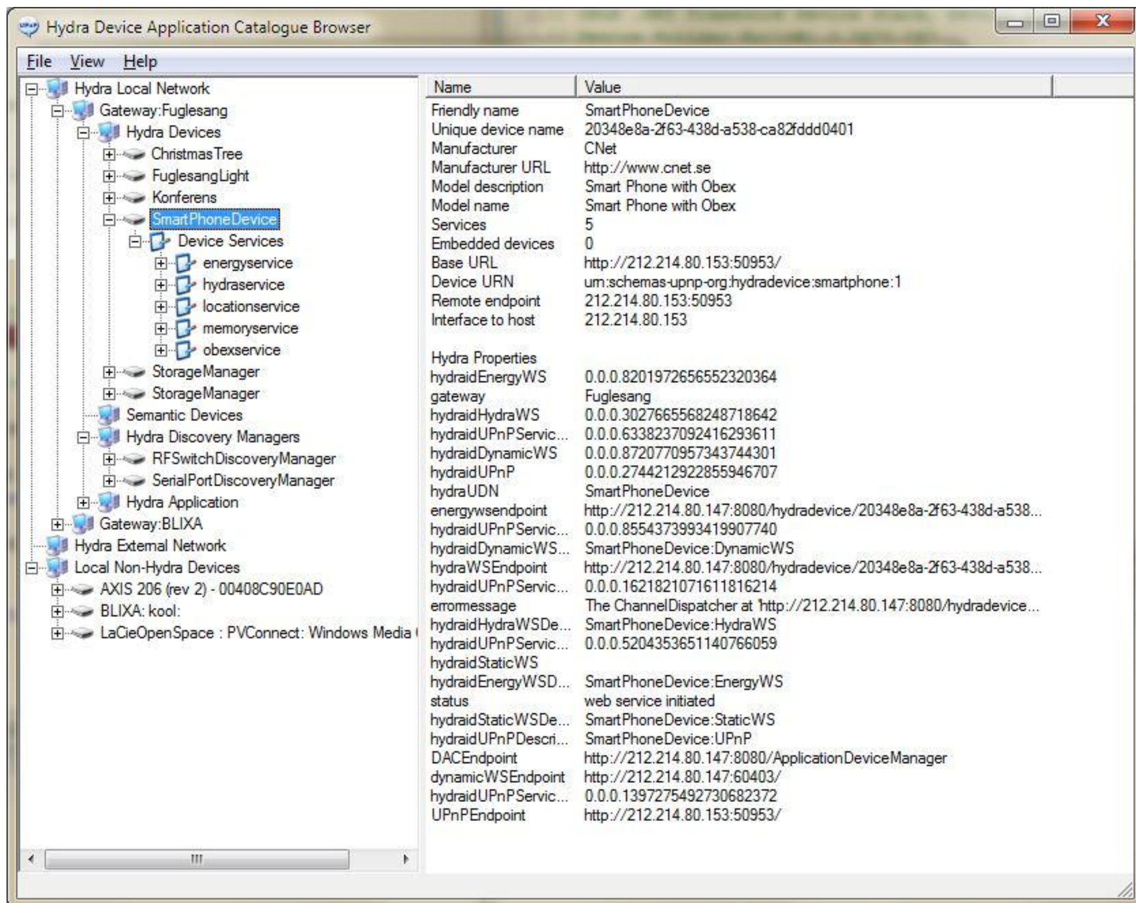

Figure 107: DAC with example SmartPhone device

9. Summary

Having sufficient training material and documentation along with the development of the software components is an ongoing process. The details given in this document reflect the final status of the software during the period of the project.

The HydraMiddleware will become an Open Source project, which means that the development is not finished yet and that programmers form around the world use this version as a basis on which to build applications and make improvements to the middleware itself.

A known fact is that technology will change and protocols and standards will arise which have to be integrated in the Hydra layer. That means that the development of the training is also given into the hands of future Hydra developers. Nevertheless a solid base is needed to create easy to use documentation materials.

Along with this document there will be other ways in which to obtain information about Hydra and its application development as a software middleware which includes a Software development Kit, a Device Development Kit and an Integrated Development Environment.

The Hydra training material outcome will also include websites, APIs, online-webinars, how-to videos and slides for the different views and functionalities of Hydra.

This material is aimed at software developers as well as business managers who have to decide which software they want to use and to base their products on.

The D12.9 is aimed at external developers who are not familiar with the process of using the HydraMiddleware to build powerful, useful and beneficial applications on top of it.

With the support and help by all partners in the Consortium this training material was developed. It was also used among the current Hydra developers to integrate their components with the core of the Hydra functionality.

This document is limited to text and figures and can not represent the whole training material which will be used to support third party developers and support decision makers in their choices.

The further reading section and link collection at the end of this document should be used as a starting point to get familiar with the overall Hydra concepts and different solutions which were extended during the entire process of the core components and middleware development.

10. References and further Reading

[1]	Brinkmann, A., Effert, S., and Gao, Y. (2009). D3.12 Updated Grid Architecture Report. Technical Report, University of Paderborn.
[2]	Ingstrup, M., and Zhang, W. (2008). D4.8 Self-star properties DDK prototype and report. Technical report, UAAR.
[3]	Al-Akkad, A.-A., Kostelnik, P., and Zhang, W. (2009). D4.10 Quality-of-service enabled hydra middleware. Technical report, Fraunhofer FIT.
[4]	Drools – Business Rules Management System. <http://labs.jboss.com/drools>
[5]	eXist-db Open Source Native XML Database. <http://exist.sourceforge.net>
[6]	OASIS - http://www.oasis-open.org
[7]	Sun XACML Implémentation. <http ://sunxacml.sourceforge.net>
[8]	eXtensible Access Control Markup Language (XACML) Version 2.0 (2005) - http://www.oasis-open.org/committees/xacml/
[9]	IBM, Web Services Security http://www.ibm.com/developerworks/library/specification/ws-secure
[10]	W3C, XML Security Introduction http://www.w3.org/2004/Talks/0520-hh-xmlsec/slide4-0.html
[11]	http;//www.osgi.org/Links/DeveloperKits
[12]	The SENSORIA Development Environment, CASE Tool for SOA Development http://home.mit.bme.hu/~rath/ppt/SDE.pdf
[13]	http://msdn.microsoft.com/en-us/netframework/aa904594.aspx
[14]	http://www.mono-project.com
[15]	http://www.oscaf.org/nrl_ontolog]
[16]	http://protege.stanford.edu/plugins/owl/jena-integration.html http://protege.stanford.edu/plugins/owl/api/guide.html
[17]	http://msdn.microsoft.com/en-us/netframework/aa904594.aspx
[18]	http://www.w3.org/Submission/OWL-S
[19]	http://www.w3.org/2002/ws/sawsdl/
[20]	http://www.uddi.org/
[21]	M. Kostoulas, M. Matsa, and N. e. a. Mendelsohn. XML screamer: an integrated approach to high performance XML parsing, validation and deserialization. 15th international conference on World Wide Web, pages 93–102, 2006.
[22]	M. Shaw. Some Patterns for Software Architectures. Pattern Languages of Program Design, 2:255–269, 1996.
	Badii, A., et.al. (2010). D12.8 D12.9 - Final External Developers Workshops Teaching Materials. Technical Report, UR.

	Ingstrup, M., and Zhang, W. (2010). D4.9 Embedded AmI components prototype. Technical report, UAAR.

Further information about the Hydra Middleware Project

Hydra Website	http://www.hydramiddleware.eu
Hydra Public Deliverables	http://www.hydramiddleware.eu/articles.php?article_id=90
Hydra in Wikipedia	http://en.wikipedia.org/wiki/Hydra_Project_%28EU_Project%29
CNet Demo	http://hydra.cnet.se

11. Glossary

This chapter aims to provide a comprehensive understanding of important terms used in and derived from the Hydra project. In addition, the terms listed in this chapter try to convey a sense of their application and present the background of the fundamental concepts. Even if some of the subsections seem to be a repetition of things already documented, this chapter can be seen as a central point of access to a description of the Hydra terms. The definitions listed here have been agreed on by the Hydra Consortium. (The terms are ordered from high-level to low-level)

Physical Device:

A "Physical Device" is a common device that offers some functions that affect the "physical world".

Such functions could for example be providing light, heat, wind, open door, or reports physical properties such as temperature, blood pressure, pulse, movements, etc. Hydra constitutes a middleware that enables networking of physical devices.

Appliance:

An "Appliance" represents a physical device that is dedicated to a single purpose. Appliances refer to more complex physical devices and are especially prominent in the field of home automation or home entertainment.

Hydra-Compliant Physical Device:

A "Hydra-compliant Physical Device" is a physical device that can be Hydra-enabled. Hydra-compliant physical devices divide into 5 different classes (see Section 3) that determine the procedure to be used to Hydra-enable devices and integrate them into a Hydra network. In the smallest class, such devices need to offer some external interface for communication and control.

Examples of such external interfaces supported by the Hydra middleware are Bluetooth, ZigBee, RF, RFID, serial ports, USB, etc.

Hydra-Enabling a Device:

"Hydra-enabling a Device" means the process of making the functions of a Hydra-compliant physical device available and controllable for other devices in a Hydra network. Depending on its device class, three methods make such a device Hydra-enabled:

- Installing (parts of) the Hydra middleware on the device
- Using the Limbo tools to embed Web Services on the device and generate a Proxy
- Using a Proxy to represent the device on a Gateway

At the end of this process the functions of this device can be invoked using Web Services, and metadata about the device is provided in the format and protocol required by Hydra.

Hydra-Enabled Device:

A "Hydra-Enabled Device" is a Hydra-compliant physical device that has successfully run through the Hydra-enabling process. A Hydra-enabled device owns a software representation, i.e. a Hydra Device, in a Hydra network and

- Can be discovered by other devices in a Hydra network
- Makes all or a subset of its functions accessible as Web Services
- Offers its Web Services either natively (embedded code) or through a proxy
- Supports UPnP and advertises its entry into a Local Area Network through UPnP broadcasting
- Supports Hydra Generic Services and Hydra Energy Service.

Hydra Device:

A "Hydra Device" constitutes the software representation of a Hydra-enabled device and its functionalities, in order to enable access and control. The Hydra Device can either run as a Proxy for the Hydra-compliant physical device on a gateway or it can run while embedded in the device. A Hydra Device can obtain Hydra identifiers for its services (HID) and also application specific identifiers. Furthermore, a Hydra Device implements the "Hydra Generic Services" and "Hydra Energy Services". For one physical device there might exist one or more Hydra Devices. A Hydra Device might also incorporate services from several physical devices.

Semantic Device:

A "Semantic Device" represents a composition of one or more Hydra devices and constitutes an SDK construct. A semantic device is dynamically bound to its Hydra devices at runtime. Therefore a semantic device might only be partially instantiated at runtime. A semantic device is discoverable in the same way as and also acts as any Hydra Device. The description of the semantic device is part of the Hydra Device Ontology.

Gateway:

A "Gateway" is a physical device with IP capabilities, which manages a set of proxies for controlling Hydra devices. A gateway must support Web Services and UPnP and should also be able to run Hydra Discovery Managers. In addition, a gateway may also host other components of the Hydra middleware.

Proxy:

A "Proxy" is a Hydra Device that consists of a software component responsible of communicating with a physical device, understanding the technology used and the format of the data exchanged. It is deployed on a gateway and represents the device to be controlled.

Bridge:

A "Bridge" represents a software component that resides inside a Gateway and translates any non-IP communication into an IP based communication. It is used by Hydra-enabled devices with non-IP capabilities to communicate inside the Hydra network.

Hydra Network:

A "Hydra Network" represents a network of Hydra Devices and applications that communicate with each other using Web Services and IP communication on top of a Peer-to-Peer overlay.

Hydra Middleware:

The "Hydra Middleware" is a collection of interrelated components, i.e. Hydra Managers, that work together to realise a platform of networked heterogeneous physical devices. The Hydra middleware allows such devices to be part of an ambient intelligence environment.

Device Discovery:

The process "Device Discovery" covers several steps where a physical device is discovered, semantically resolved and made accessible as a Hydra Device. In order for a device to be discovered in a Hydra network, a definition of the device type must exist in the Hydra Device Ontology.

Hydra Manager:

A "Hydra manager" (or short "manager") constitutes the major building blocks that make up the Hydra middleware. A Hydra manager encapsulates a set of operations and data that realise a specific functionality and is mostly subdivided into several internal components.

Hydra Generic Services:

The Hydra Generic Services are supported by all Hydra Devices and contain a set of meta-data methods that can be used to query the device about its properties.

Hydra Energy Services:

The Hydra Energy Services are supported by all Hydra Devices and provide methods to retrieve information from the energy profile of the device and from the energy policy.

Hydra Identifier (HID):

A "Hydra identifier" (or simply "Hydra ID" or shorter "HID") constitutes a unique identifier for every Hydra Device, service or resource within a Hydra network. The Network Manager generates the HID, is responsible for the matching between logical and physical identifiers and for the propagation of this information to other peers of the Hydra network.

CryptoHID:

An application developer has the opportunity to assign his own CryptoHID to a certain Hydra Device. This CryptoHID can directly be used throughout the application code and referred to when expressing security, energy and other policies.

Session:

A "Session" traces the communication between elements of a Hydra network, in order to keep the communication coherent. Sessions allow the maintenance of the state of each network element as they communicate with each other. The Network Manager comprises a dedicated Session Manager that creates and maintains the lifecycles of the session objects.

Ontology:

An "Ontology" is a representation of the knowledge of a formally defined system of concepts and relationships. In addition, an ontology can contain inference to derive new knowledge and integrity rules to assure its validity. Therefore, an ontology forms a network of information and logical relationsships described through a formal language such as the Web Ontology Language (OWL).

Hydra Device Ontology:

The "Hydra Device Ontology" is an ontology that contains knowledge about device classes, their properties and services offered.

Device Model:

A "Device Model" describes the properties and services that a certain device class offers. The Device Model is expressed using the SCPD XML format of the UPnP standard.

Hydra Peer-to-Peer Architecture:

The "Hydra Peer-to-Peer Architecture" allows Hydra Devices in different local Hydra networks to access and communicate with each other. This means Web Services calls can be executed remotely over a P2P overlay.