Contract No. IST 2005-034891

Hydra

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

D4.9 Embedded AmI components prototype

Integrated Project
SO 2.5.3 Embedded systems

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

Published by the Hydra Consortium **2010-03-25 - 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: D4.9 Embedded AmI components prototype.doc

Work package: WP4 – Embedded AmI Architecture

Task: T4.4 – Embedded Ami components prototype

Document owner: Matts Ahlsen (CNET)

Document history:

Version	Author(s)	Date	Changes made
0.1	Mads Ingstrup (UAAR) Klaus M. Hansen (UAAR)	11-03-2010	Initial outline, introduction
0.2	Amro Al-Akkad (FIT)	15-03-2010	Table 1 updated
0.3	Klaus Marius Hansen (UAAR)	17-03-2010	Executive summary. Table updates
0.4	Matts Ahlsén, Peeter Kool (CNET)	2010-03-18	Components update
0.6	Matts Ahlsen	2010-03-18	Tutorials update, final compilation & editing
0.9	Matts Ahlsen	2010-03-19	Version for internal review
1.0		2010-03-26	Updates based on review comments. Final version submitted to the European Commission

Internal review history:

Reviewed by	Date	Comments
Jesper Thestrup (IN-JET)	2010-03-19	Approved with comments

Index:

1. Executive summary

This deliverable consists of a set of prototype components for the Hydra middleware. The deliverable is a prototype so we here give a summary of the components that have been developed. In doing so, we provide two things

- An overview of which DDK components are provided by WP4. This takes its outset in D3.14
- A brief tutorial for each of the major components if a tutorial has not been provided elsewhere

The purpose of this deliverable is thus present a basic guide for developers to start using the components.

Although the components are finished, recent updates in requirements have necessitated further development of components. This means that they are functional now, but that we will issue an update of the prototype (expected M47) including an updated version of this report.

2. Components overview

As noticed in D4.4[1], the contents of a Development Kit (e.g., a DDK) may be logically divided into the following types of components:

- **Runtime components.** These are program components that are used at runtime to support applications using the DDK. Examples include link libraries, virtual machines, and debuggers.
- **Development time components.** These are program components that are used at development time to build applications based on the DDK. Examples include compilers, linkers, or assemblers.
- **Resource components.** These are non-program components that support programs, developers, and users when utilizing the DDK. Examples include developer documentation, sample code, and configuration files.

Table 1 shows the components in the DDK prototype, following the outline in Deliverable 4.4. The component prototypes are not all completed by the time of this deliverable. The status of the components is indicated in Table 1. The reason for the components not being finished is that they build on the design in deliverable D3.14 which was delivered in M43.

Category	Type	Component	WP	%	Deliverable[1]
Runtime	Libraries	Event Manager (wp4)	WP4	90%	This
		ASL Interpreter (wp4)	WP4	90%	This
		Flamenco (wp4)	WP4	70%	D4.11
		QoS Manager (wp4)	WP4	80%	D4.5 and D4.10
		Hydra Device Library	WP6	90%	D6.8, D6.10, D3.13
	Monitoring	Flamenco Probes (wp4)	WP4	100%	D4.11
	Tool	Tracing tool	WP4	50%	
		Ontology Manager web interface	WP6	90%	D6.7,D6.9
Development time	Compiler	Limbo	WP4	90%	D4.7, D3.14
		Limbo Plug-ins	WP4	90%	D4.7, D3.14
	Libraries	Flamenco	WP4	70%	D4.11
		DDK Class Library for .net	WP4/6	90%	D6.8,D6.10
		Visual Studio Project Templates	WP6	80%	D3.13/14
		DDK Tools .Net	WP4/6	85%	
Resources	Documentation	Tutorials	WP12		D12.5
		Documentation in source			

[1] See list of referenced deliverables at the end

	code (all)		
Ontologies	Limbo Configuration Ontology	WP4	100%
	Statemachine Ontology	WP4	100%
	Device Ontology	WP4/6	90%

Table 1. Components in the DDK prototype. The % column shows how complete the component is with respect to implementation of requirements and provision as open source. The Deliverable column indicates what deliverable the component is part of, if it (or a previous version) has been delivered previously.

The following sections contain tutorials for the components, as indicated in table 1. As they are frequently updated to correspond with the implementation they are hosted on the Hydra wiki , and the text appearing here is a snapshot of this source. The section is not intended to replace documentation produced, e.g., in WP 12, but merely to indicate what has been implemented in this prototype.

3. Using the .Net DDK tools

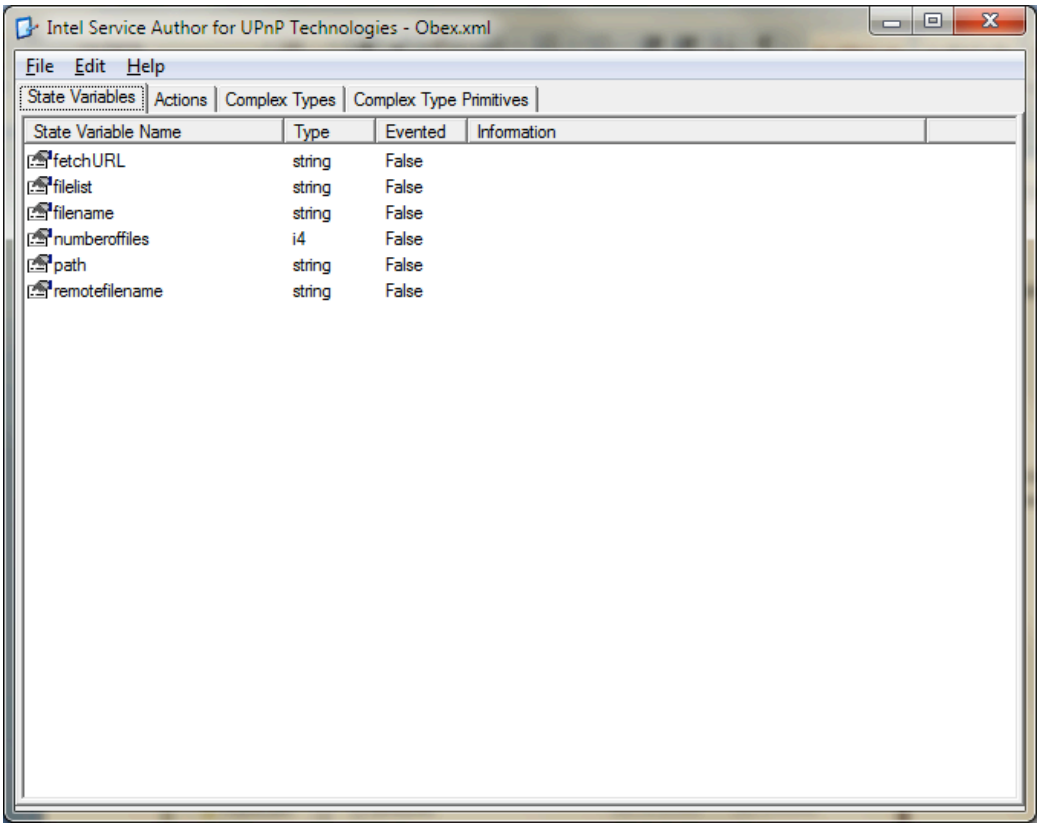
There are two main tools for creating device code for .Net in Hydra:

- 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.

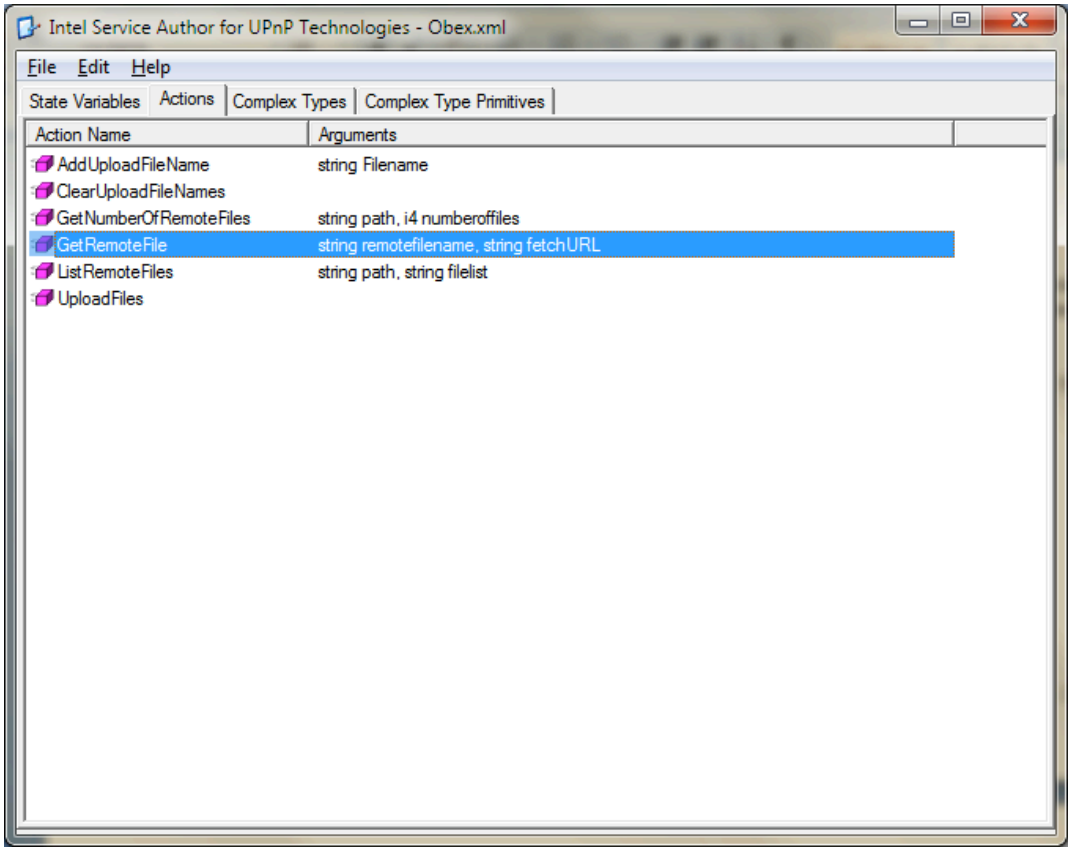
Using Intel Service Author for UPnP Technologies

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

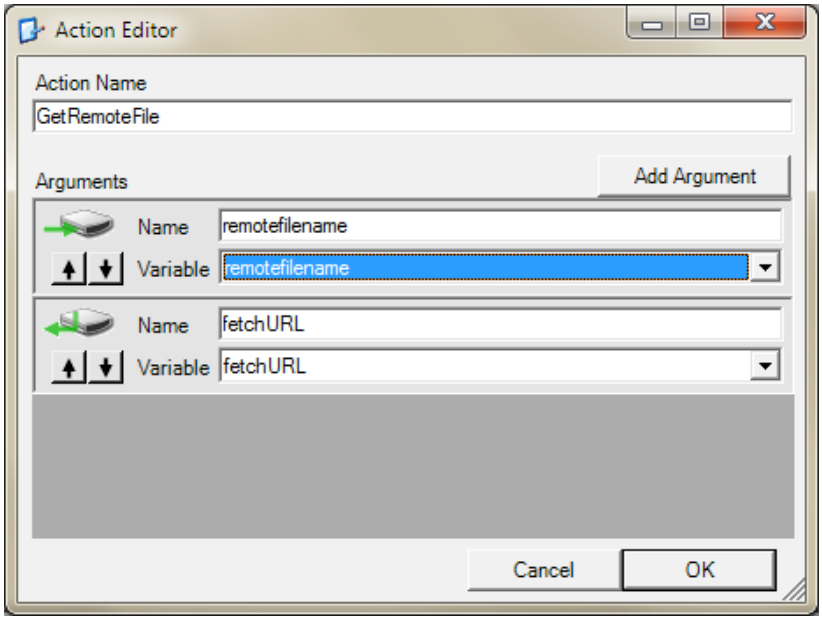


The first step is to define the state variables that will be used by the service. State variables have to be defined for all Input/output parameters used in the service. In this case we have a number of state variables defined with their respective types.

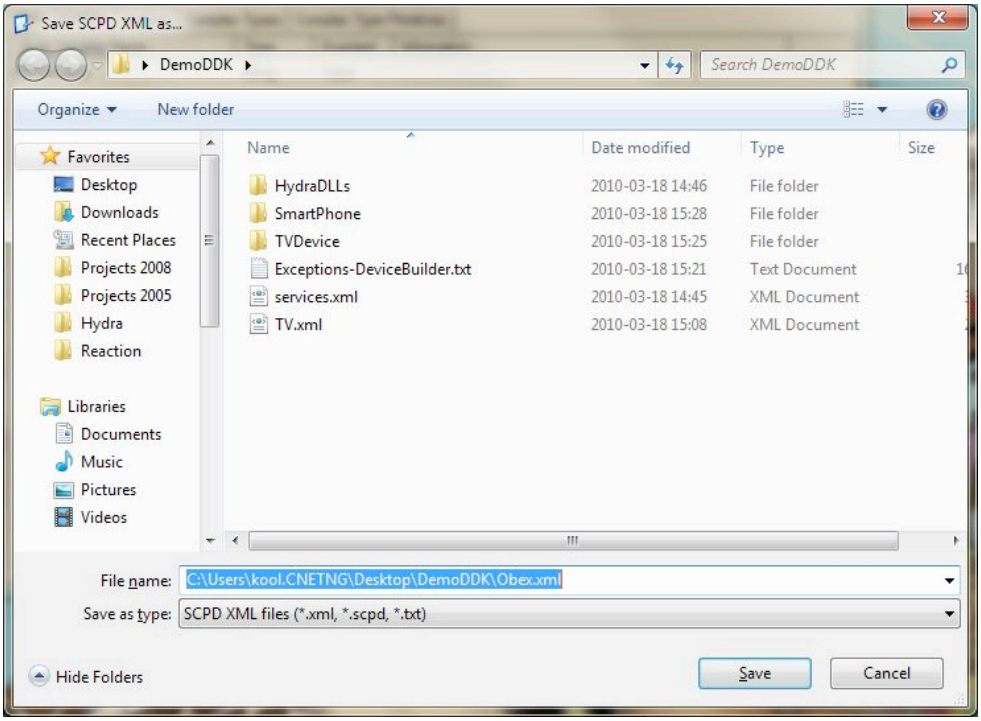
The next step is to define actions, i.e. the methods that this service should support. This is done in the "Actions" tab.



Here we have defined a number of methods 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, see below.



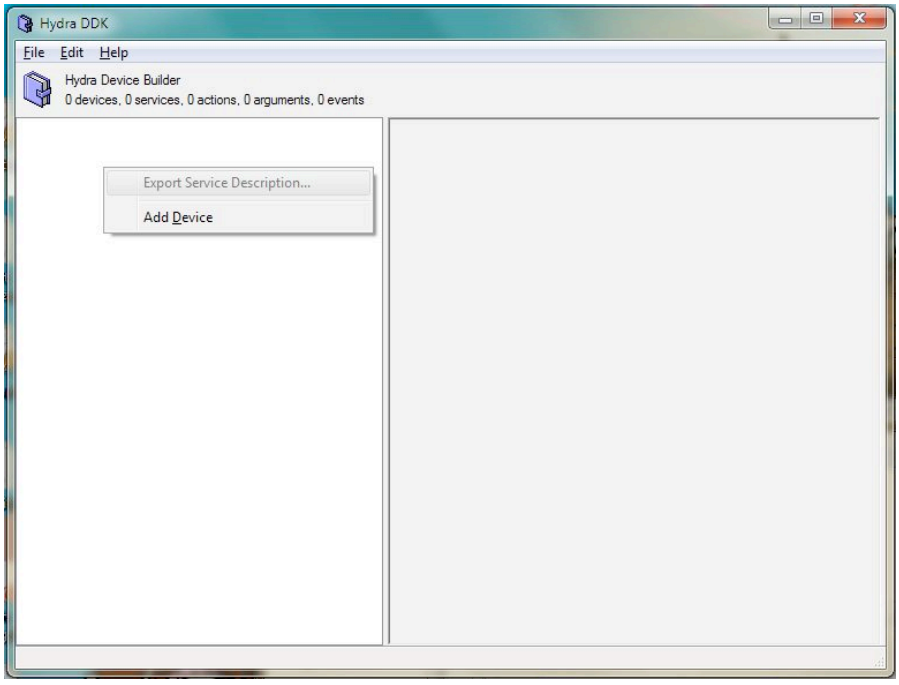
When one is finished it is time to save the SCPD to file for later processing in the Hydra .net DDK tool.



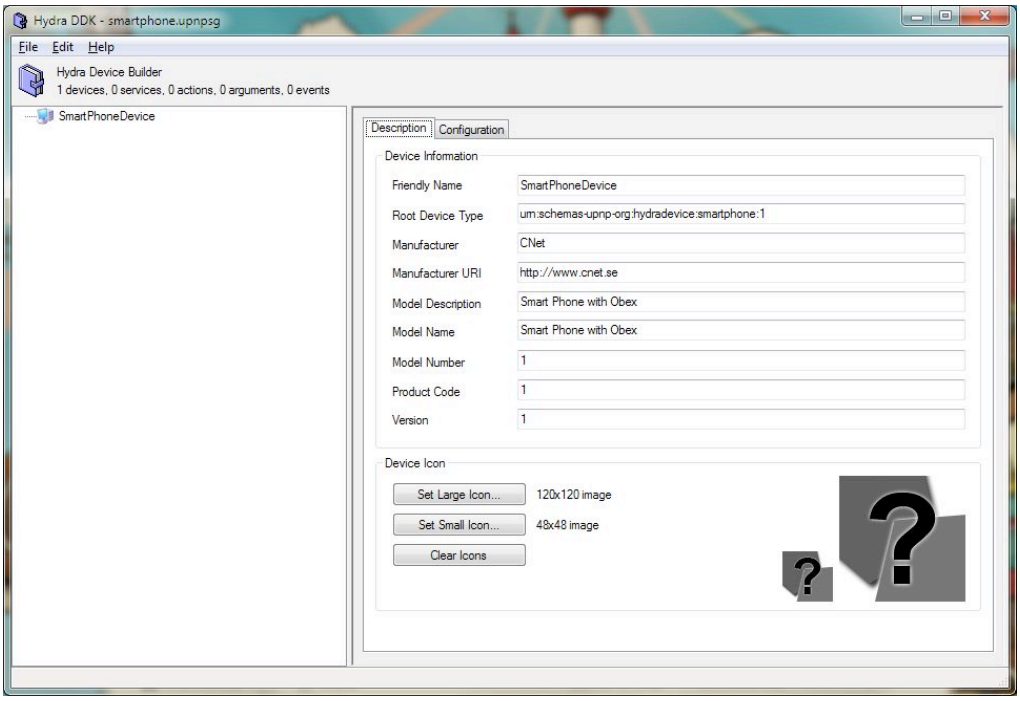
Using Hydra .Net DDK tool

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

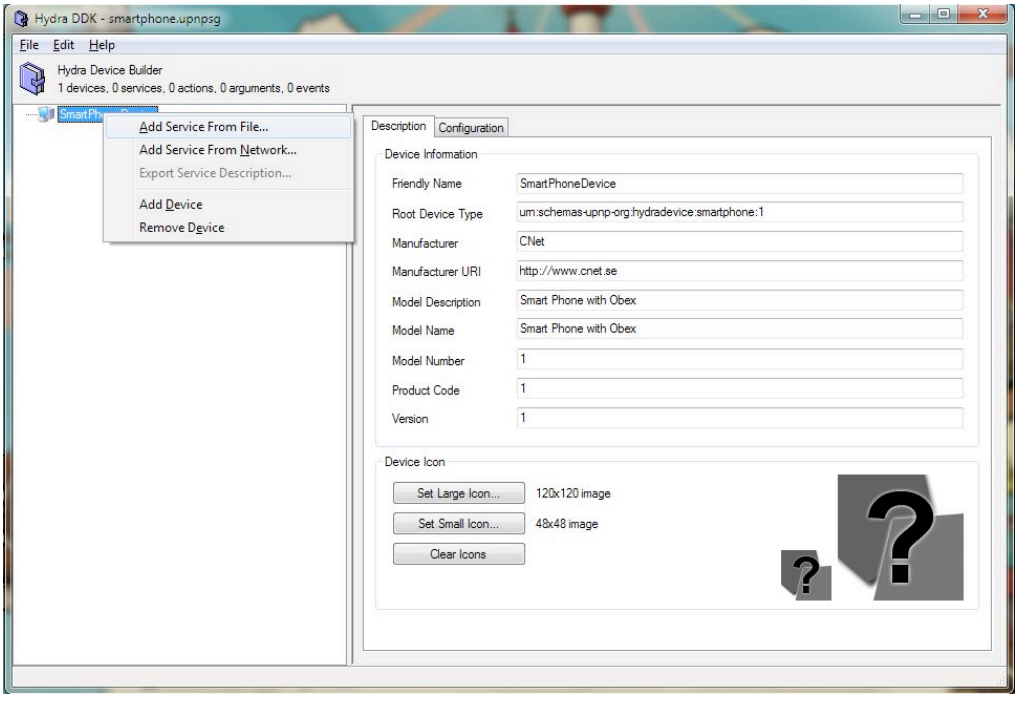
The first step is to "Add Device" by right clicking in the tools left pane.

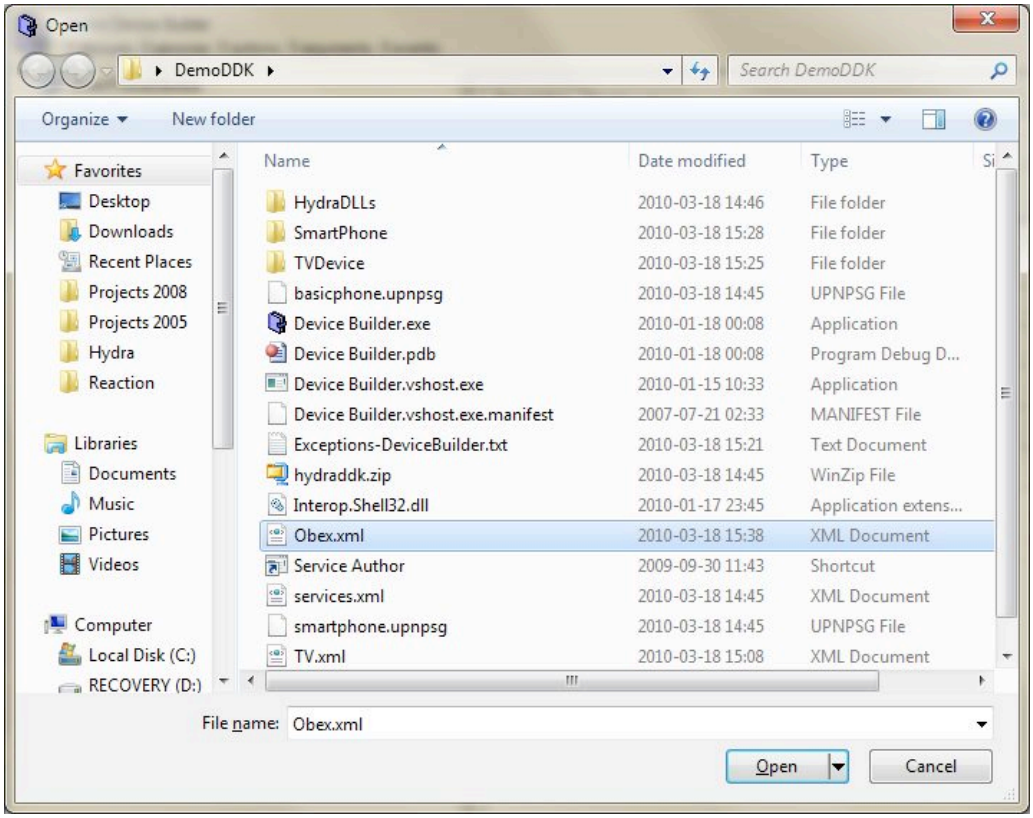


The next step is to edit the meta data for the device, i.e., device name, type, description etc.

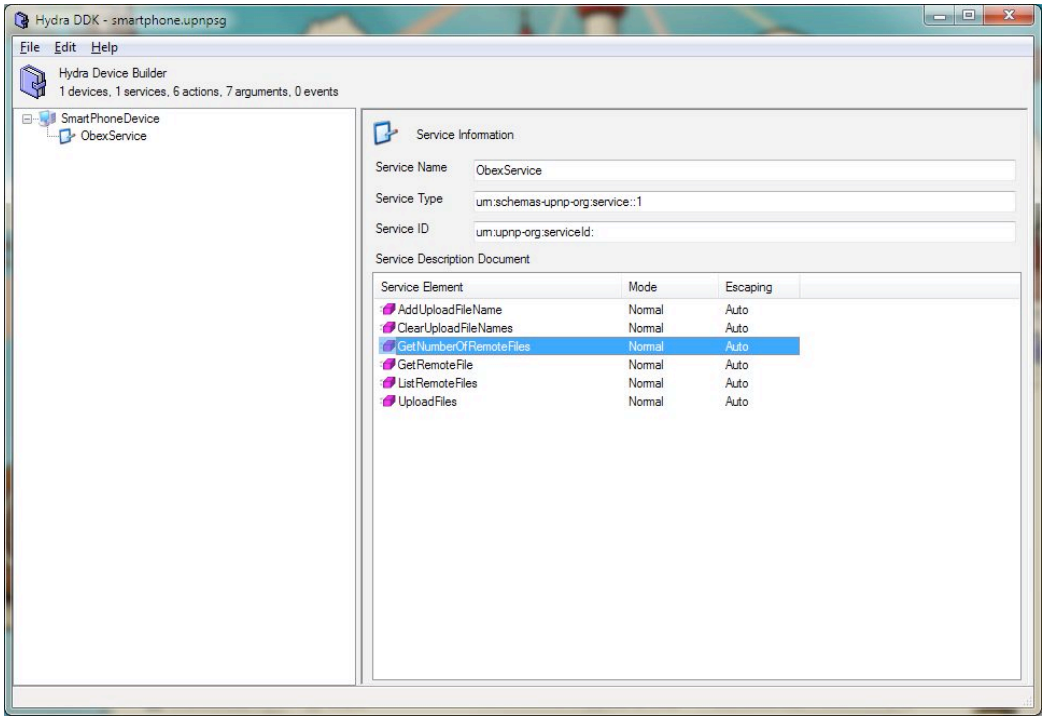


Then one adds the service created in the previous section by right clicking on the device in the left pane.

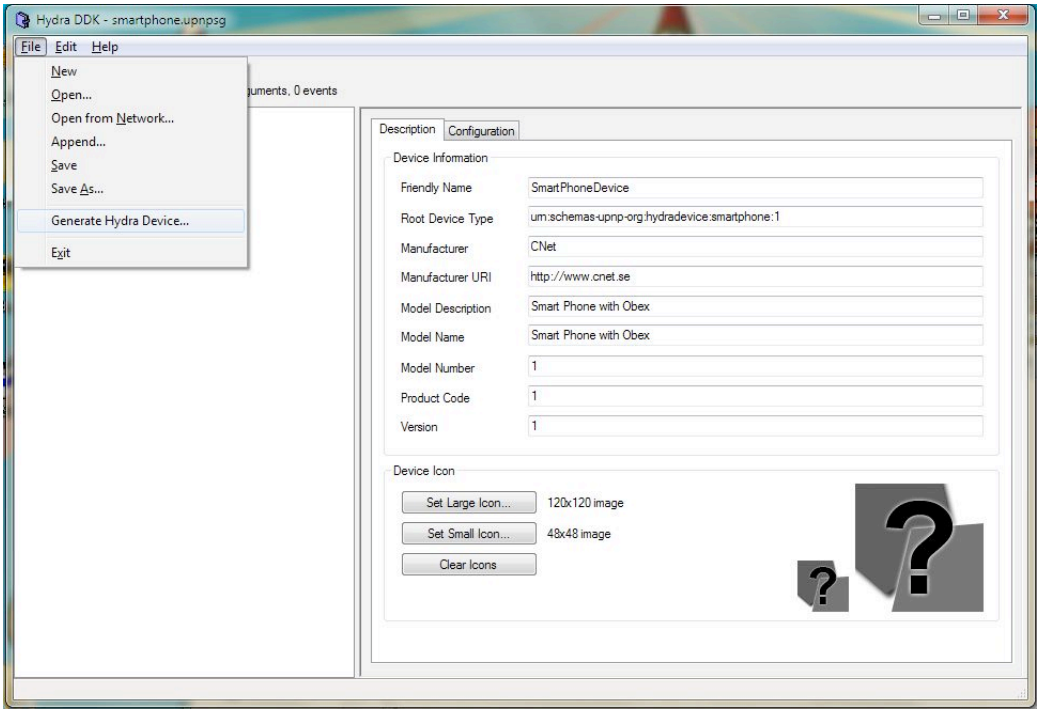




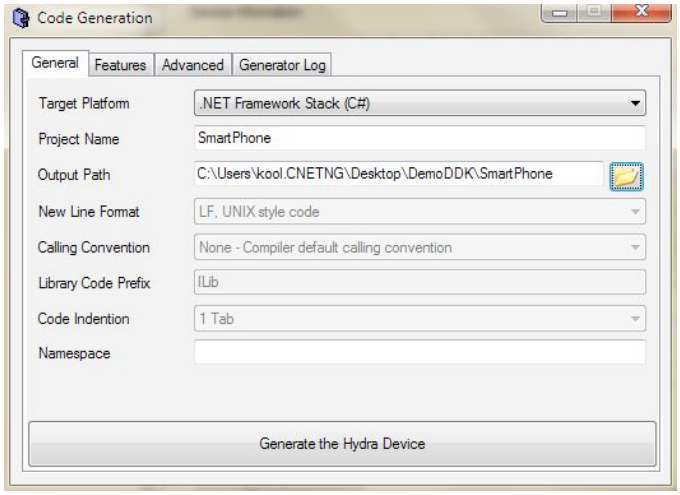
Now we have added the OBEX service and we can see all the methods in that service.



Finally we have arrived at the stage where it is time to generate the code for the Hydra device. Select the "File" menu and choose "Generate Hydra Device".

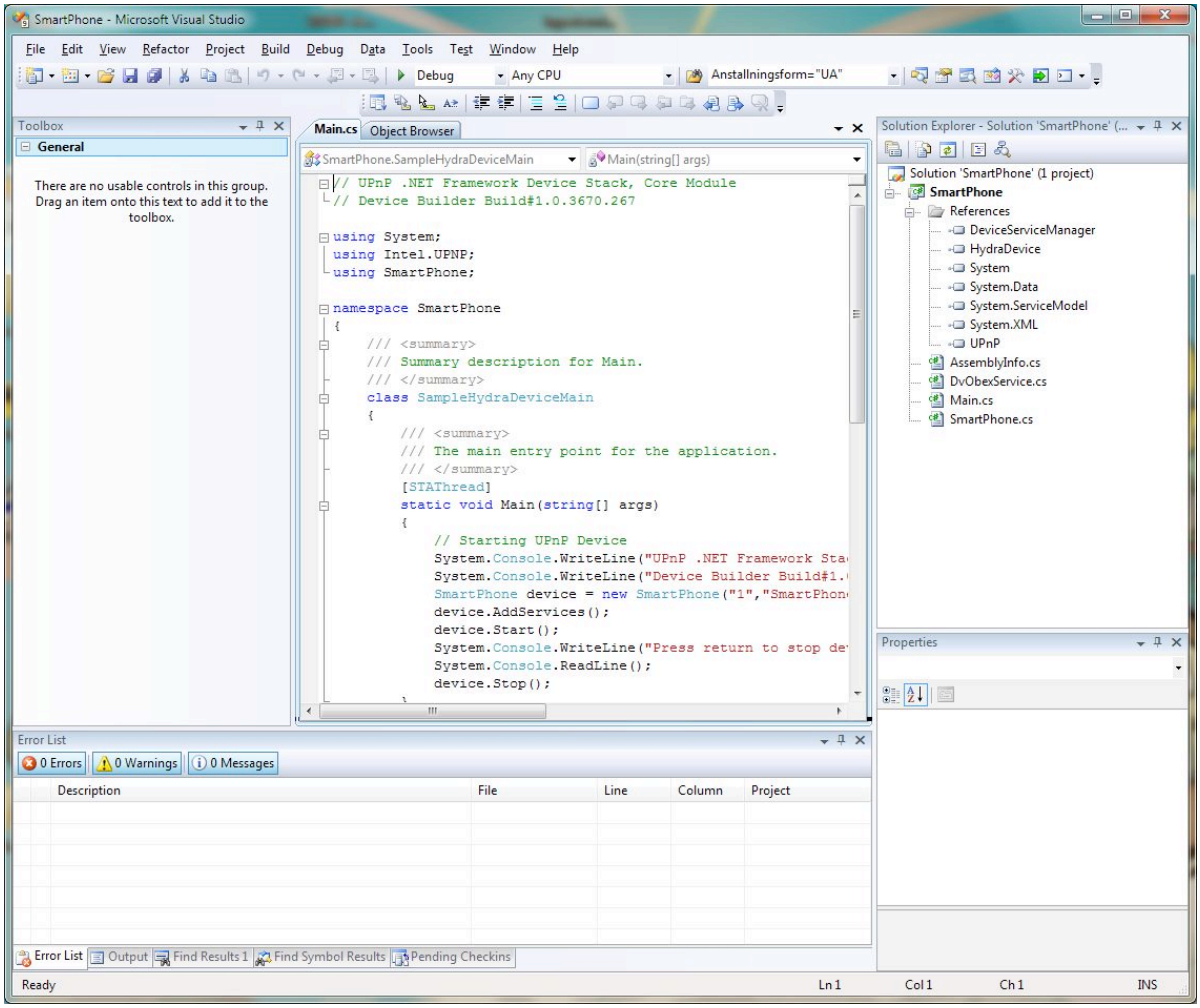


In the code generation dialogue one has to decide the project name and optional Namespace for the generated code.

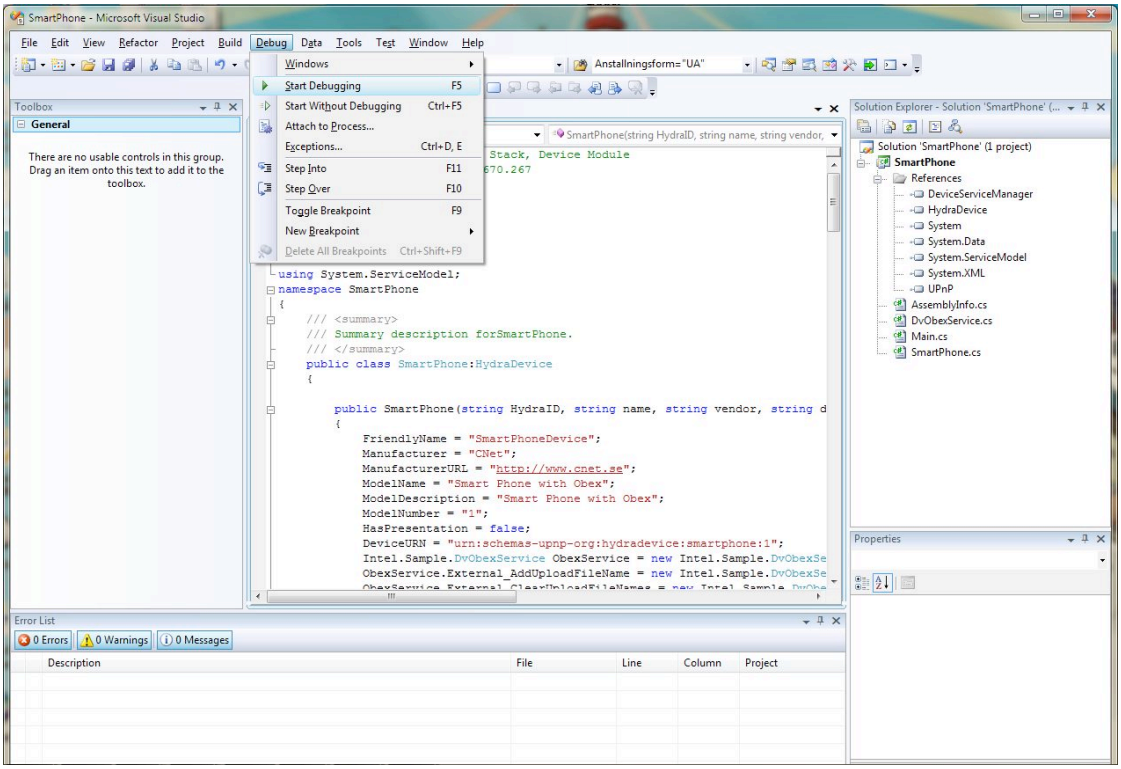


A complete Visual Studio project is created with the necessary Hydra references.

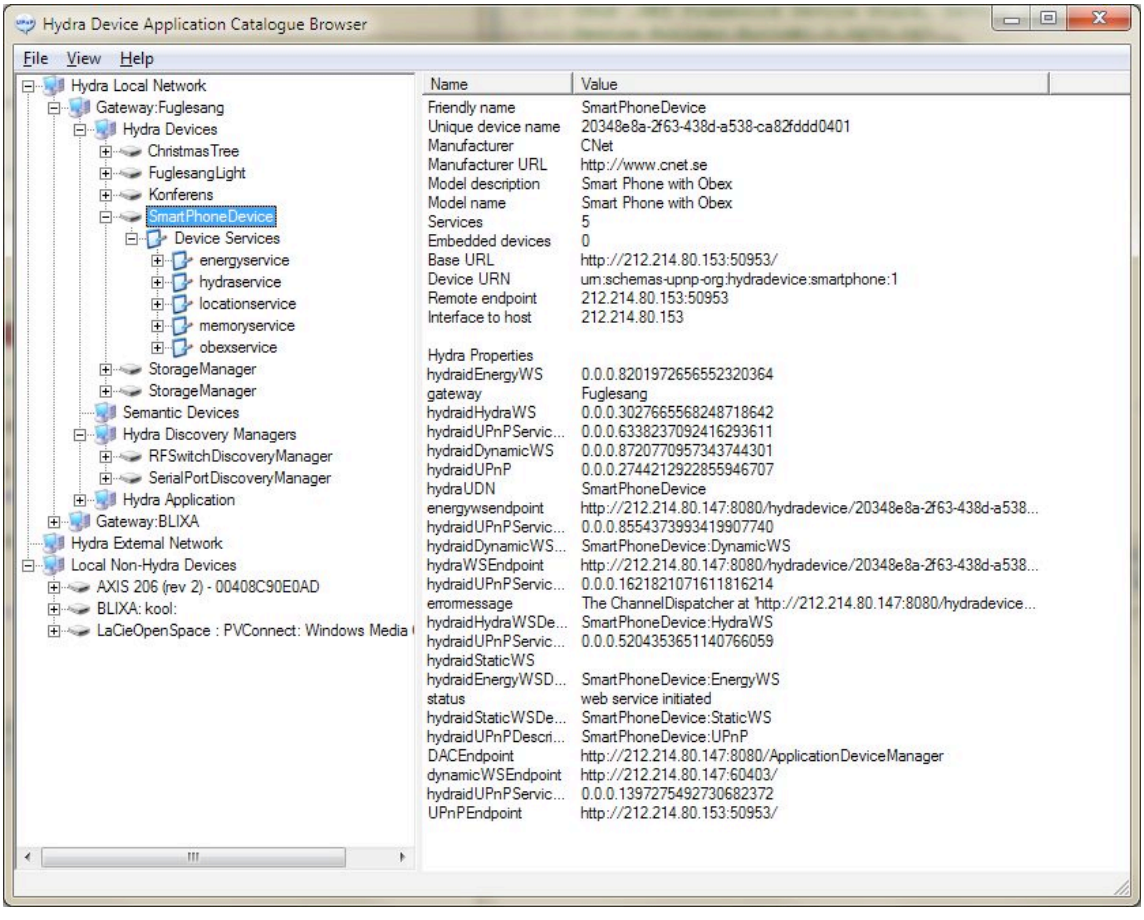
The next step is to open the Visual Studio project.



The device code is already runable since all methods are stubbed. Normally one would then change the code in the stubs to do the actual device communication. The location of the stubs to be changed is in "Device name".cs, i.e. SmartPhone.cs in this project. But in this case we will start the device by opening the "Debug" menu and selecting "Start debugging".



If we run a Hydra DAC tool we will find our newly created device with all its services. Note that we have automatically received the relevant Hydra services: HydraService, EnergyService, LocationService and Memory service. We can also see that the Device is properly discovered and has all the Hydra properties such as HID.



4. Quality of Service Manager (QoS Manager)

General

The Hydra QoS Manager has been developed as an OSGi bundle using Declarative Services.

To provide a well-structure between Hydra Managers, it has been ported completely to Hydra Commons.

Purpose

The QoS Manager provides an interface for selecting best-suitable services under specific QoS requirements. By the term "QoS", we refer to non-functional service parameters.

A detailed description of the QoS Manager including service design and real-world scenario can be found in deliverable D4.10 "Quality-of-Service Enabled HYDRA Middleware".

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 regurlarly updated from ontology.

Dependencies from Hydra components

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. In this sense, before we continue it strongly recommend to read the Hydra Commons tutorial.

For processing specific service requests, the QoS Manager needs to query Hydra Ontology.

For this the Hydra Ontology Manager is supposed to provide an accessible web interface (#WS method name) for retrieving data values in respect of QoS properties describing Hydra services and the (embedded) devices these services are running on.

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 compoonent requires regularly an update of quality views of services parameter, and thus needs to request a specific set of QoS properties for self-adaption

Prerequisites

There are following system requirements:

- Java 1.5
- eclipse 3.5 (we used Eclipse Java EE IDE for Web Developers. Build id: 20090621-0832)

1.1.1 OSGi Environment

Any environment able to run OSGi Java modules, e.g. Knopflerfish, felix, equinox etc. We used Eclipse 3.5 Java EE IDE for Web Developers. Build id: 20090621-0832 on Windows 7 Enterprise.

Installation

- Install new Hydra Commons and latest version of Network Manager (see new Hydra Commons tutorial).
- Download QoS Manager from SVN

Configuration

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 Hydra Commons Configurator.

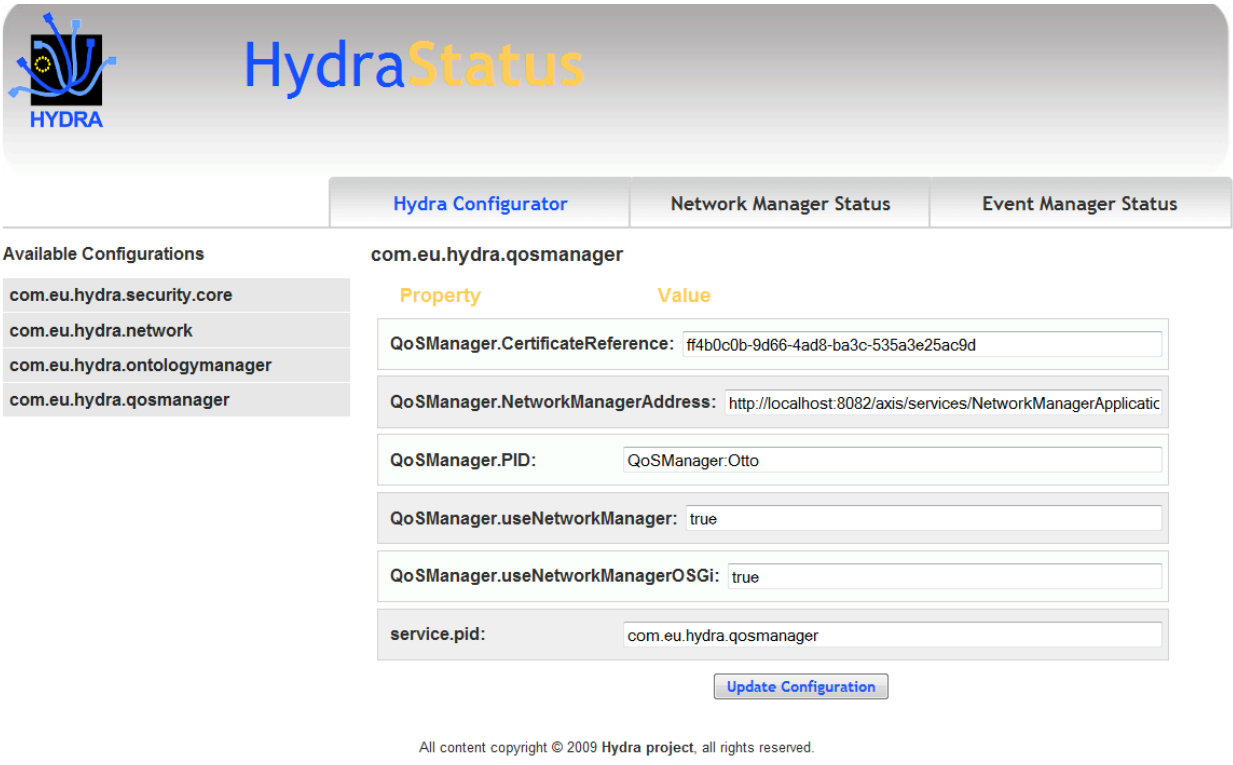
While running your OSGi configuration, you can change all listed properties via http://localhost:8082/HydraStatus#.

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

To make QoS Manager performing correctly, you need to adapt the following fields (PID, NetworkManager) accompanied with exemplified values:

- QoSManager.PID = QoSManager:Hydra,
- QoSManager.useNetworkManager = true, or
- service.pid = com.eu.hydra.qosmanager.

See also this screenshot (or D5.11 Wireless Network IDE Prototype):



Testing

- See QoSManagerTester bundle for invoking the QoSManager and also
- See JUnit Tests for OSGi-based Quality-of-Service Manager

5. ASL Interpreter Tutorial

The architectural scripting language consists of a set of operations for manipulating a runtime architecture. It is expressed in terms of a general ontology for software architecture, which is mapped to concrete concepts realized by implementation platforms such as OSGi. This tutorial addresses the OSGi implementation of ASL.

The ASL interpreter is an OSGi bundle. It has one method, executeScript(String script). To use the ASL interpreter it must be loaded and started into the OSGi platform.

ASL Scripts

An asl script is a sequence of operations separated by semicolon. The available operations are:

```
init_component(<handle>, <device>, <designator>)
init_device(<handle>)
init_service(<handle>,<device>, <component>)
deploy_component(<componenthandle>, <devicehandle>)
undeploy_component(<componenthandle>,<devicehandle>)
start_service(<servicehandle>)
stop_service(<servicehandle>)
start_device(<devicehandle>)
stop_device(<devicehandle>)
```

Designators and handles

A handle is just a variable in the script. It references an architectural entity such as a service, a component or a device. A handle is assigned its value with one of the init_ operations. This is done by specifying a designator as argument to the init_ operation. A designator is of the form &<property>=<value>. For instance, an adequate designator for a component handle to the OSGi event service would be:

```
init_component(eventcomp, d1, &Symbolic-Name=org.apache.felix.eventadmin);
```

The following table lists which properties are supported for designators on the osgi platform.

```
SymbolicName
Location
Id
```

Note that since the concepts of component and service in ASL's general architecture ontology are both mapped to a bundle in the OSGi ontology, the set of designators is the same for components and services in OSGi, but this is not necessarily the case for other platforms.

String variables

One intended usage of ASL is for initiating a platform instance. In this case, we experienced that we often needed to load several components from the same location. To avoid writing the same path many times ASL now supports definition of string variables. Only string variables exists, and their occurrence in operations is replaced by the string they denote before the script is executed. As such they are a kind of macros, except that they cannot be used for general macros since scripts must support the asl grammar. This grammar is provided

Example

The following listing exemplifies the current capabilities of the ASL interpreter, and the syntax it accepts:

```
/*
 * an example script that stops and uninstalls the felix eventmanager
bundle
 */
```

```
init_device(local);

// normal java comments and blank lines are now ok

// strings can be defined, to make scripts less verbose:

$workspace=/Users/ingstrup/Documents/workspaces/Hydra2/;

// use + to concatenate
init_component(aql,$workspace+flamenco_aql/exported/flamenco_aql.jar);
deploy_component(local,aql);
init_service(local, aql,aql_s);
start_service(aql_s);

/*
 * we can also attach a handle to a bundle that wasn't installed in the
script
 * it's done with a designator of the form [&<field>=<value>]+
 * the supported fields are currently "Id" "SymbolicName" and "Location"
 */

init_component(eacomp,&SymbolicName=org.apache.felix.eventadmin);
init_service(local,eacomp,eas);
stop_service(eas);
undeploy_component(eacomp);
```

Notes

The current implementation does not support distributed scripts automatically. That is, there is not yet a way to designate other devices than the local one. The handle 'local' for the local device is initiated automatically.

ASL grammar

This section lists the grammar for ASL in the format of the CUP parser.

```
SCRIPT          ::=     EXPRLIST
EXPRLIST        ::=     EXPR



EXPRLIST EXPR

EXPR            ::=     OPCALL ';'

'$' STRING:varname '=' STRING:value ';'
OPCALL          ::=     STRING '(' PARLIST ')'

PARLIST                 ::=     PARAMETER

PARLIST ',' PARAMETER

PARAMETER       ::=     STRING

'$' STRING:varname '+' STRING:concat

DESIGNATOR

DESIGNATOR      ::=     '&' STRING:key '=' STRING:value

DESIGNATOR      '&' STRING:key '=' STRING:value
```

6. Event Manager Tutorial

Main Functionality

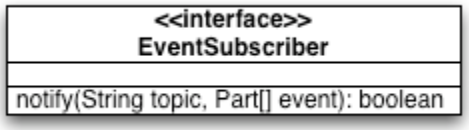
The Hydra Event Manager provides publish/subscribe functionality, i.e., the ability for publishers to send a notification to multiple subscribers while being decoupled from them (in terms of, e.g., not holding direct references to subscribers). The specific variant of publish/subscribe implemented is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

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

!eminterface.PNG!

Publishers and subscribers use this interface directly. Publishers invoke publish() while subscribers invoke subscribe() and unsubscribe(). The use of the interfaces is further explained in the documentation of the code.

Furthermore, subscribers need to implement the following interface:



Deployment

Currently, the easiest way to run the Event Manager is using Eclipse. We here assume that you have checked out the Hydra project from Subversion:

1. Import the following projects in Eclipse from the Hydra source

- EventManagerServerBundle
- Log4j
- NetworkManagerBundle
- org.os4os.forge.axisbundle
- CryptoManager
- XMLSecurity

2. In the file EM.properties in the EventManagerServerBundle root, you can set the following properties

```
EventManagerAddress=http://localhost:8082/axis/services/EventManagerPort
servicePort=8082
withNetworkManager=true
NetworkManagerAddress=http://localhost:8082/axis/services/NetworkManagerAp
plication
SOAPTunnelingAddress=http://localhost:8082/SOAPTunneling
EventManagerDescription=EventManager_AARHUS
```

In the first property you can set the endpoint of the Event Manager, the second is the port of the Event Manager, the third property you can set whether the Network Manager is used (true or false). The next properties are the endpoint of the NetworkManager, the endpoint of the SOAPTunneler and the description of the Event Manager as it will appear in the Network Manager

3. Create a new Run Configuration of type "OSGi Framework". In the Workspace bundle set, select the bundles that you imported in step 1.

The start level of the Event Manager needs to be higher than of the Network Manager (since we rely on HIDs from this), so you may want to increase the former to 5.

Press "Add Required Bundles". This should make sure that the following bundles are selected in Target Platform bundle set:

- javax.servlet
- org.apache.commons.logging
- org.apache.log4j
- org.eclipse.equinox.cm
- org.eclipse.equinox.ds
- org.eclipse.equinox.http.jetty
- org.eclipse.equinox.http.servlet
- org.eclipse.osgi
- org.eclipse.osgi.services
- org.junit
- org.morbay.jetty.

As a VM argument add the following

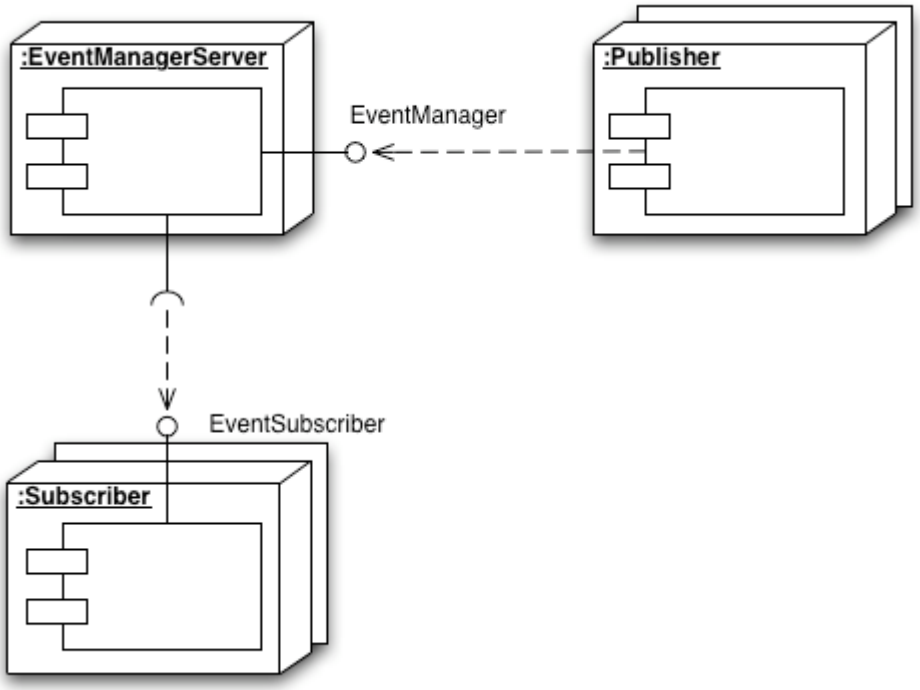
```
-Dorg.osgi.service.http.port=8082
```

This is the port where the Event Manager and Network Manager will run. Select Run.

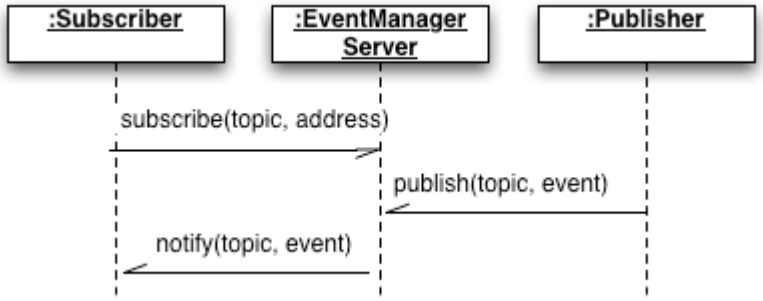
4. At http://localhost:8082/axis/services you should see the registered Axis services.

5. The EventManager provides also a functionality where you can see the status of the EventManager at runtime, i.e., subscriptions information, in your browser at http://localhost:8082/EventManagerStatus

The figure below shows the resulting deployment



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



7. Limbo tutorial

This tutorial aims at giving Limbo users a guide for generating services using the Limbo compiler. For this purpose, we will use a simple example of a thermometer service. The device used is a PICO TH03 thermometer, which is the same one used in the first Hydra prototype.

We define two operations in the service, `getStatus` and `getTemperature`, 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);
}
```

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)

You can get Limbo from:

- http://www.cs.au.dk/~marius/limbo/limbo.zip

To install Limbo, unzip to an installation directory, in the following called <limbo>. We assume that commands are invoked in the <limbo> directory.

Using Limbo

Using Limbo contains the following step:

1. Describe targeted device in Hydra's device ontology (optional)
2. Describe the service in a WSDL file. Reference the device description from the WSDL file
3. Describe the service-related statemachine in the device ontology (optional)
4. Run the Limbo compiler on the WSDL file
5. Implement and deploy the device-specific service
6. Run the service

Describing the device in the Hydra ontology

A device has basic device type information (modeled 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), we add the following code 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>
```

We also add its hardware information 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.

Describing the service in a WSDL file

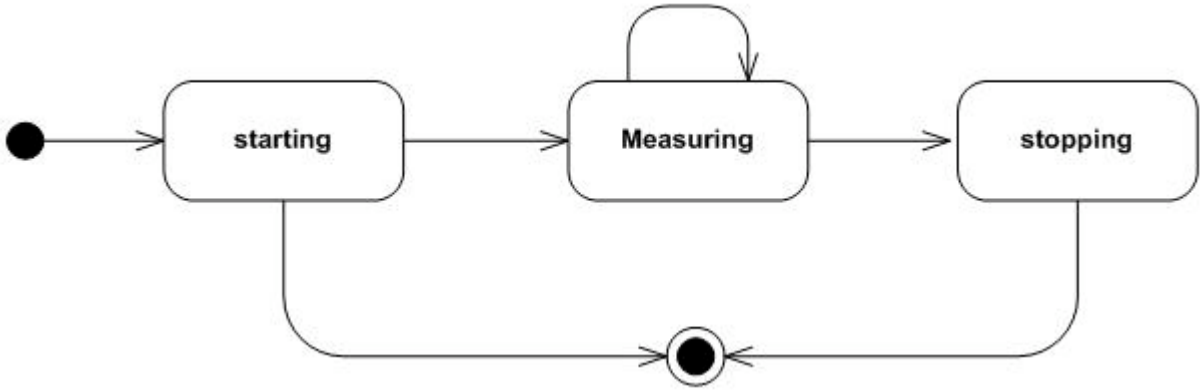
In this case, we will use a simple WSDL file for a thermometer service that contains two operations, one for `getStatus` and another for `getTemperature`. The WSDL file, which is also in <limbo>/tutorial/wsdl/, is shown below:

```
<?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>
```

Describing the service-related statemachine

For the current implementation of state machine stub code generation, we need a dummy state machine instance 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.



We can add the dummy state machine Thermometer_sm to the state machine ontology as followed. 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>
```

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, leave the argument list 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)

Implement and deploy the device-specific service

The generation created two directories in generated/:

- th03rClient
- th03rServer

Each of these contain a project that can be imported in Eclipse. The projects are self-contained and may be copied to your workspace.

The actual device binding is implemented in the com.eu.hydra.limbo.TH03PortOpsImpl class. Here we implement the `getStatus` and `getTemperature`. In the following, we just show a dummy implementation:

```
/**
 * 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;
```

```
}
```

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"));
```

Now also run TH03PortLimboClient in Eclipse. To change the default behavior of the generated server skeleton, change the com.eu.hydra.limbo.TH03PortOpsImpl class as described above.

8. Resource Manager Tutorial

General Description

The resource manager is a very simple access to loading bundles. It initiates a method in one of the included libraries which then reads an initiation file with instructions as to which bundles should be loaded initially. The resource manager is started by running the OSGiResourceManager.java file under the src folder. The file for designating the bundles to be loaded is the config.ini in lib/configuration. It can look like this:

```
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, \
../../../../../sdk/flamenco/ontodiagnosis/IPSniffer/dist/IPSniffer.jar@4:start
```

And the syntax for each line is:
BundlePath@#:start, \
Where BundlePath is the relative path to the jar file containing the bundle, # is used for assigning the order in which the bundles are started - bundles with the same numbers are started at the same time. Notice that only the last line does not end in , \

Tutorial

1. Set up SVN to download the source folder from:
https://hydra.fit.fraunhofer.de/svn/trunk/middleware/managers/ResourceManager/ResourceManager
2. Edit the config.ini file according to the description above.
3. Run the java file under the src folder.
4. You now have a console in which you can interact with the OSGi for instance using writing ss to get a short status of what is running or using stop 6 to stop bundle number 6. If you start a resolved bundle it will try to start again and this time you get the exceptions.

Management bundle

General Description

General description:
The management bundle is intended to provide a web service interface for managing the bundles on a server. This is done by allowing the user to either install, remove or replace bundles on the server. This functionality is offered through 3 different services:

- stop(String name)

This service stops a bundle designated by the name given. The name given must match the symbolic name of the package to be stopped (case indifferent).
If there are several packages by the same symbolic name the first one found is stopped.
This returns true if a bundle is stopped and false otherwise.

- start(URI location)

This service loads the service found at the specified location.
It will return true if a bundle is successfully loaded and false otherwise

- update(String name, URI location)

This is simply a combination of the two above which tries to remove the bundle designated by name and if successful tries to install the bundle designated by location.
The service will return true if both operations succeed and false otherwise.

Tutorial:

1. Set up SVN to download the source folder from: https://hydra.fit.fraunhofer.de/svn/trunk/middleware/managers/ResourceManager/Management
2. Compile the jar file using the default ant target: jar in the build file.
3. You should now have a new folder in management called dist and a file herein called management.jar. This is the bundle to load.
4. Load the management bundle using, for example the resource manager.
5. If you just want to try out the update or stop service in the next step load some other bundle, for example the abloy_el582 bundle.
6. Check which bundles are running (for example by typing ss in the console that is running a resource manager)
7. Create a client to make a call to management according to the WSDL file in .../Management/wsdl/ calling for instance:
a. update(abloy_el582, file:///C:\svn\trunk\middleware\managers\ResourceManager\Pico_TH03\dist\pico_th03.jar)
b. stop(abloy_el582)
c. start(file:///C:\svn\trunk\middleware\managers\ResourceManager\Pico_TH03\dist\pico_th03.jar)
You can use the test class ManualTester in the bundle for this - this is launched using the runManualTest ant target.
The ManualTest application has two text boxes for inputting the name and the address and will perform an update when the corresponding button is used.
Be aware that the parameters are very sensitive.
8. Check which bundles are running now to see that you have achieved the desired effect.

9. Flamenco Tutorial

Flamenco is a tool for supporting self-management in Hydra-based systems. It currently exists in two versions:

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

In the following we will guide users through a simple example of self-management and how to use Flamenco to realize a scenario of managing a flow meter-based agricultural system. And also we will explain how to make use of Flamenco to choose optimized solutions according to multiple (conflicting) objectives, for example QoS requirements on memory consumption, goodput, throughput, realiability and so on.

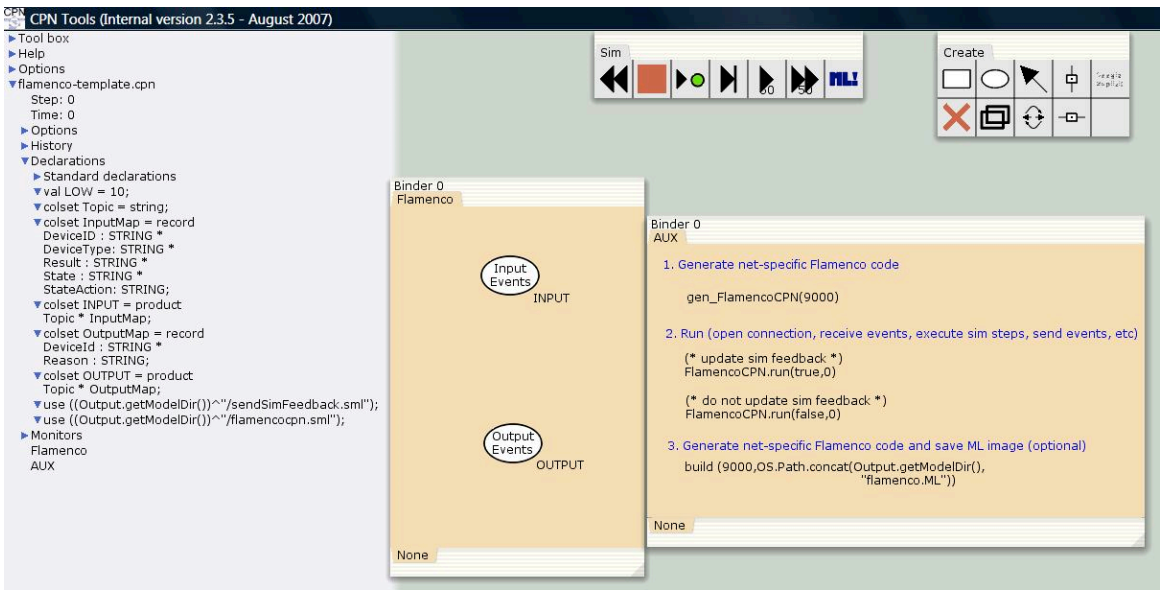
Flamenco/CPN

System Requirements and Installation

- Windows XP or Vista (CPN Tools only runs on Windows or Linux)
- Java 5 or later
- CPN Tools. Request a license and download the tool from: http://wiki.daimi.au.dk/cpntools/cpntools.wiki. You need version 2.3.5 which is an internal version. Either ask the CPN Tools people for it or ask UAAR for how to download.
- Access to the Hydra SVN repository
- Eclipse Europa or later for running Flamenco

Design Time Usage

You may use CPN Tools 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:



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 you change the net (and want to use it at runtime in Flamenco/CPN), you will need to evaluate the first expression. 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 if you want to be able to run Flamenco/CPN entirely without a user interface. Evaluating the expression will generate a Standard ML image that contains the specific net.

The net

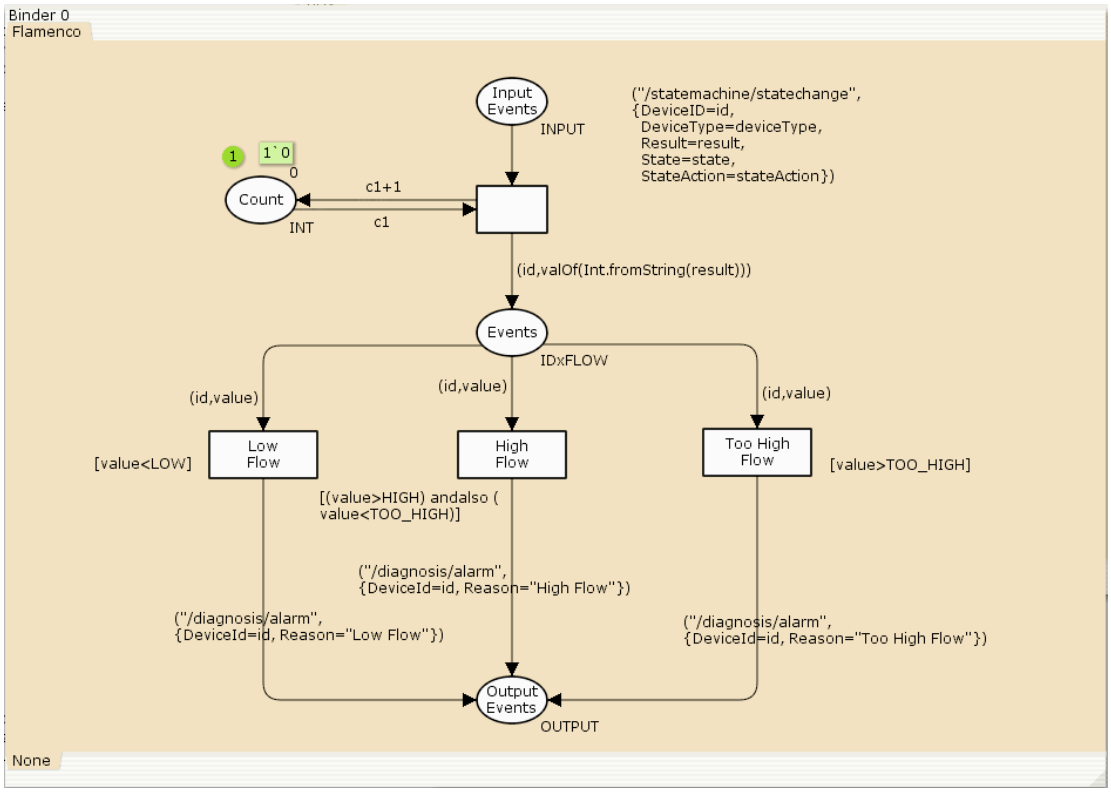
The template net is shown in the middle. It only contains two template places ("Input Events" and "Output Events"). These places will receive and send events from the Hydra middleware respectively at runtime. In general there should be one place with the colour INPUT and one place with the colour OUTPUT

The declarations

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

Runtime Usage

A full Flamenco/CPN example may be found in `HYDRA/sdk/flamenco/flamencocpn/resources/cpn/flamenco-scenario1.cpn`. The net for this is shown in the figure below:



To run Flamenco/CPN, you will need to

1. open CPN Tools on a Flamenco/CPN net as described in the previous section and evaluate the appropriate auxiliary declarations
2. run the Hydra Event Manager
3. run the Java part of Flamenco/CPN. The easiest way currently is to start Eclipse on the FlamencoCPN project (in `HYDRA/sdk/flamenco/flamencocpn/`) and then run `com.eu.hydra.flamenco.cpn.Flamenco`.

4. run a number of devices that will produce events

For simulating the last step, you may consider using the FlamencoTest project (in `HYDRA/sdk/flamenco/test/flamencotester/`). The class `com.eu.hydra.flamenco.cpn.FlowTester` will produce events from the flow meter scenario. If you have start CPN Tools with user interface updates, you will see tokens being produced and consumed in the net.

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.
Here is the list of tools needed for running it:

1. Tomcat 5 or later
2. Protege 3.4 build 130 or later

Installation

1. 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
2. Install Protege. The current SWRL APIs needs to access the ontologies coming with Protege, therefore, the running of OWL/SWRL based Diagnosis manager needs to point to the Protege installing directory. This is not necessary since protege 3.4 build 500, as the SWRL related ontologies can be accessed directly if you have internet connection. But 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
3. Download the Flamenco/SW from Hydra SVN: HYDRA\trunk\sdk\flamenco\ontodiagnosis. All ontologies are located in \ontodiagnosis\resources directory.
4. Copy all ontologies (including rule ontologies) to the newly created 'ontology' directory. Now all the rules and ontologies are ready for use.
5. Install the testing client by downloading from HYDRA\trunk\sdk\flamenco\test\flowmeter or HYDRA\trunk\sdk\flamenco\ThermometerGeneric

Usage

Flamenco/SW listens to topic of '/statemachine/statechange', '/flamenco/socketwatch'. Therefore if you want to diagnosis a system/application/device, you must publish events on these topics. 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 you the one you publish events, in order to make use of the Network manager and Trust manager functionalities.

Please follow these steps:

1. Start Tomcat.
2. Check that the Event manager is running (for the moment it is using EventManager_CNET). Start Event manager or else.
3. Start Network manager
4. 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 that you have. After this start the diagnosis manager with ant build. Alternatively, you can start Flamenco/SW by running as Java application by click on class

5. start one of the test clients. In this tutorial we use the flowmeter client which is in the SVN directory: HYDRA\sdk\flamenco\test\Flowmeter. Build it with the ant build file in order to create a jar file, called Flowmeter.jar. Copy the Flowmeter.jar to Resource manager (resource manager is under HYDRA\trunk\middleware\managers\ResourceManager\ResourceManager) lib directory, and then change the config.ini under the lib\configuration as follows:

6. osgi.bundles = \

7. ../lib/org.eclipse.equinox.log_1.0.1.R32x_v20060717.jar@2:start, \

8. ../lib/org.eclipse.equinox.common_3.2.0.v20060603.jar@2:start, \

9. ../lib/org.eclipse.osgi.services_3.1.100.v20060601.jar@2:start, \

10. ../lib/javax.servlet_2.4.0.v200706061611.jar@3:start, \

11. ../lib/org.eclipse.equinox.http_1.0.2.R32x_v20061218.jar@3:start, \

12. ../lib/Flowmeter.jar@4:start

13. The last line is used to start the flowmeter test client. Now start the client by simply running it as a java application. Now you can see 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.

14. You can try the thermometer scenario by the thermometer client under svn: \HYDRA\sdk\flamenco\test\ThermometerGeneric. Remember to change the config.ini and change Flowmeter.jar to Thermometer.jar (the jar name built from thermometer client).

One thing to note is that if you try the testing multiple times using ant build file coming with Flamenco/SW, you may have to kill the Java process with task manager (in windows Vista/XP we experienced this problem), remember to leave the one for Tomcat5.5, which is usually using around 45-50M memory. But this will not happen to running the approach of running as java application directly.

Development

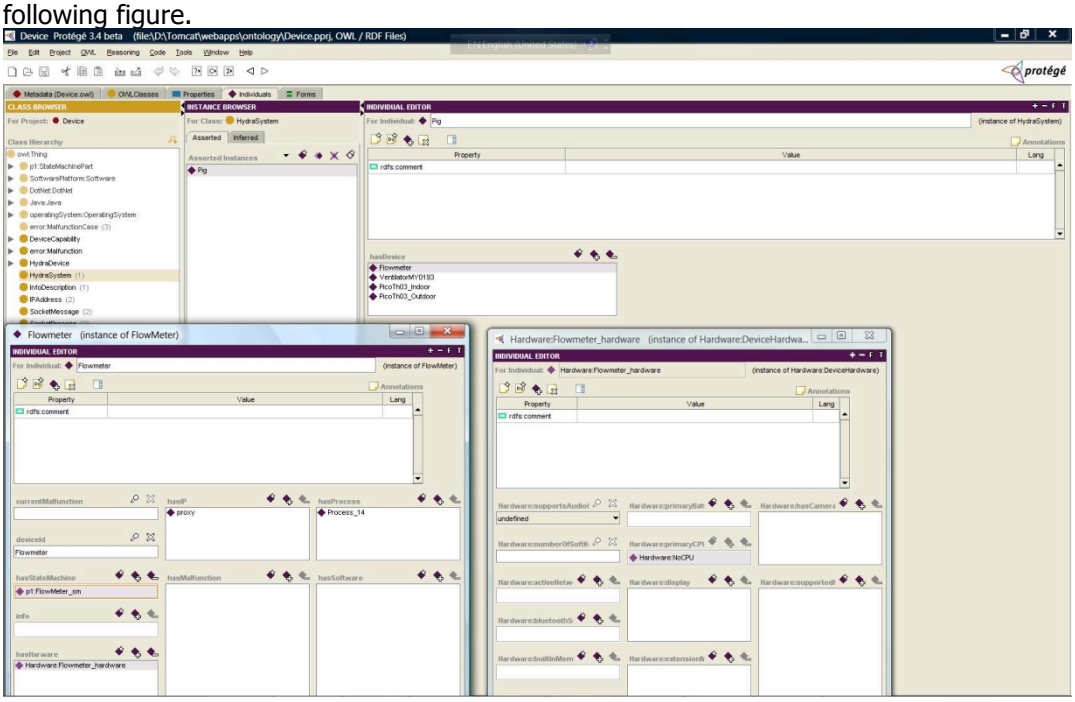
There may be two kinds of developers who can utilize the Flamenco/SW diagnosis manager SDK, knowledge developer, who is responsible for the development of rules and the addition of diagnosis cases based on the existing ontologies, and Java application developer who make use of the rules and ontologies for realizing 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 in HYDRA\trunk\sdk\flamenco\ontodiagnosis\resources 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 is:

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



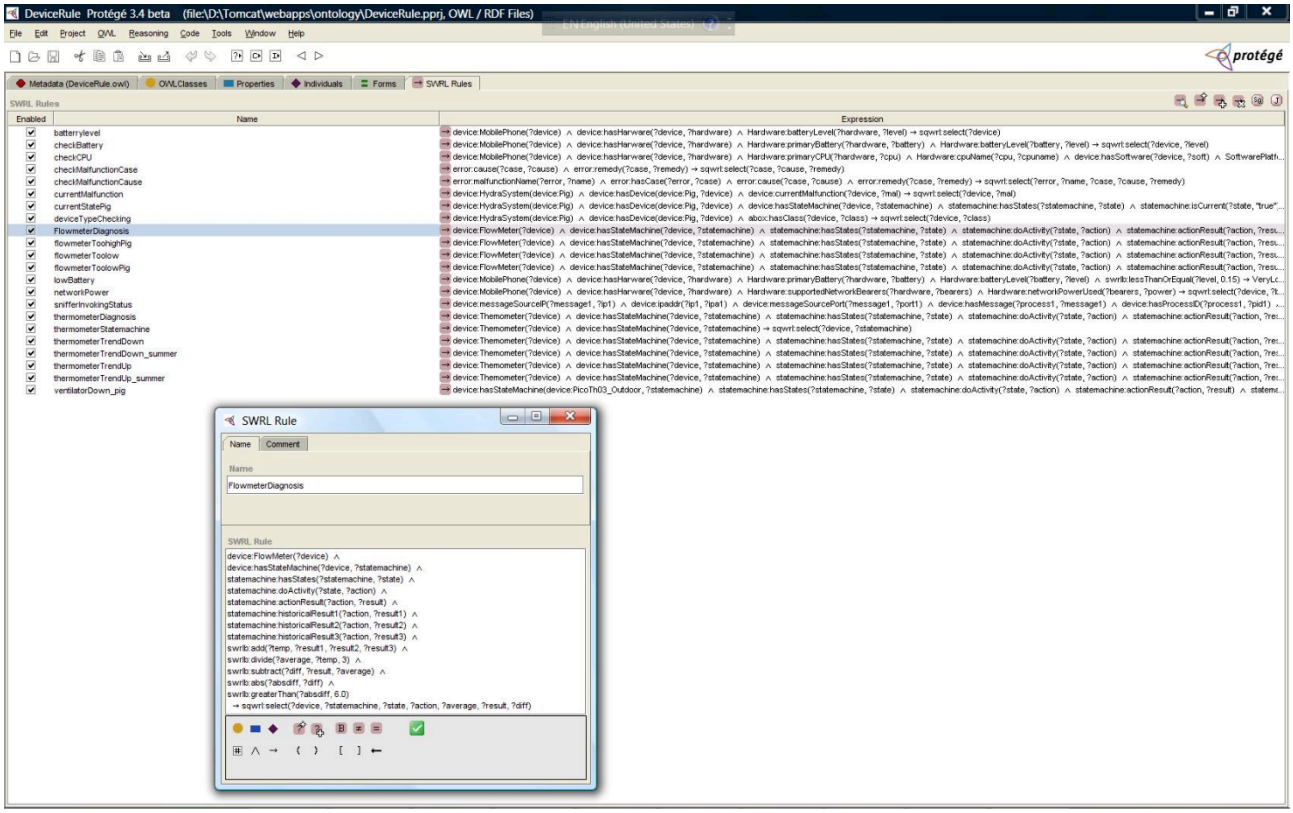
2. Add the flowmeter state machine instance to the StateMachine ontology. It is called "Flowmeter_sm" in our case, if it does not exist. This step can refer to the WP4 ontology tutorial.

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.



5. Add diagnosis case to the Malfunction ontology. For example, we can add the "flowToohigh" instance 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 case 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 build 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 optimized solutions for a problem, based on the JMetal GA framework. Currently the planning layer supports three GAs, NSGA-II, FastGPA or MOCell.

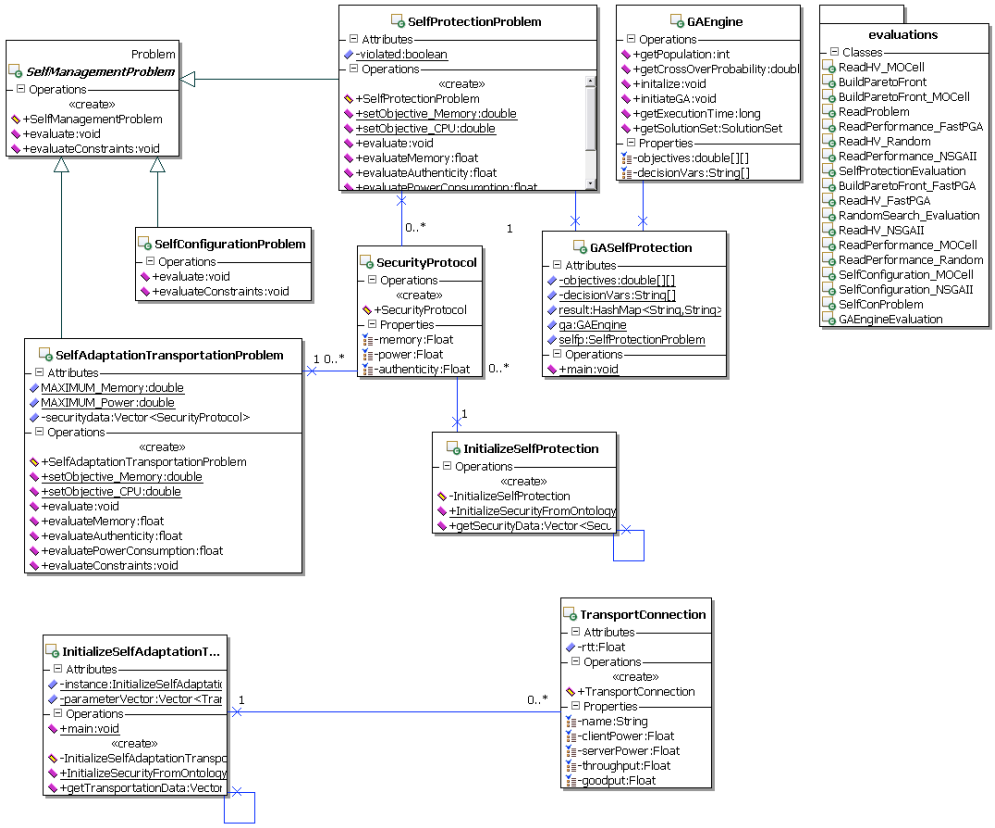
Usage

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 we use the self-adaption example to illustrate how to run the planning layer. From the tests/evaluations we have done, NSGA-II is the best GA for self-management problem, in which usually

In package eu.hydraproject.selfmanagement.planning, run TestSelfConfig, the NSGA-II genetic algorithm will find a number of optimized solutions, and its corresponding variables.

Development

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



Chromosome encoding and fitness evaluations

The representation of chromosome in our case is using integer (starting from 0). That is to say, we are using a 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) 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 our case, it chooses AxisSENM, AxisMECM, LimboMEDM, and so on. Based on the chosen components, the GAs then decide its fitness using the objective equations as introduced in Section 2.2, and will evaluate whether the constraints are meet at the same time.

Define optimization problem

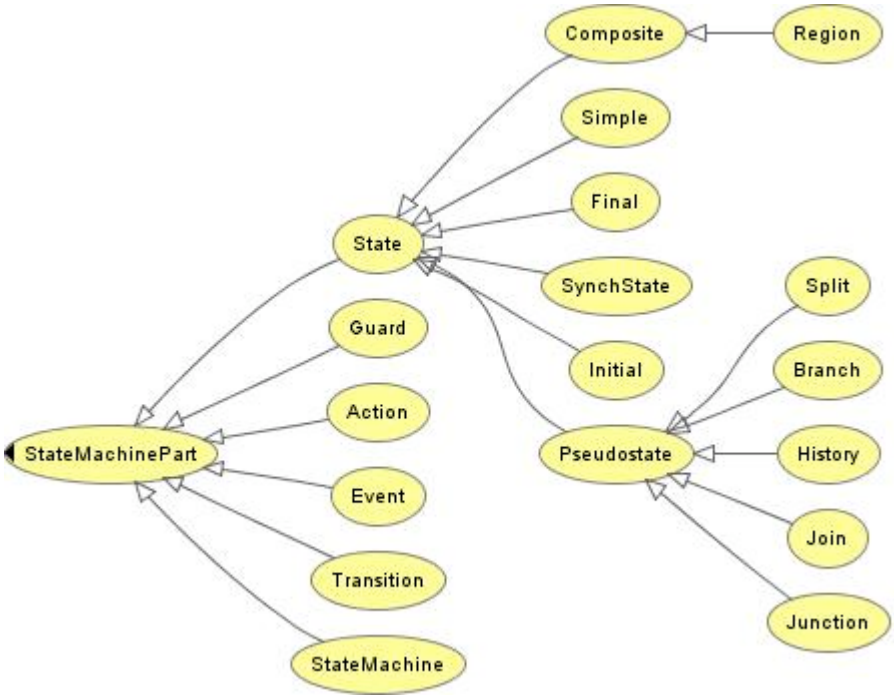
A number of steps are need to abstract an optimization problem

1. Define the problem class (e.g. SelfConfigurationProblem), which should extend the SelfManagementProblem, which extends JMetal:Problem interface.
2. Define the methods for evaluating fitness of a solution, which is defined also in SelfConfigurationProblem 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 component in our case) can be obtained.

10. Ontologies Tutorial

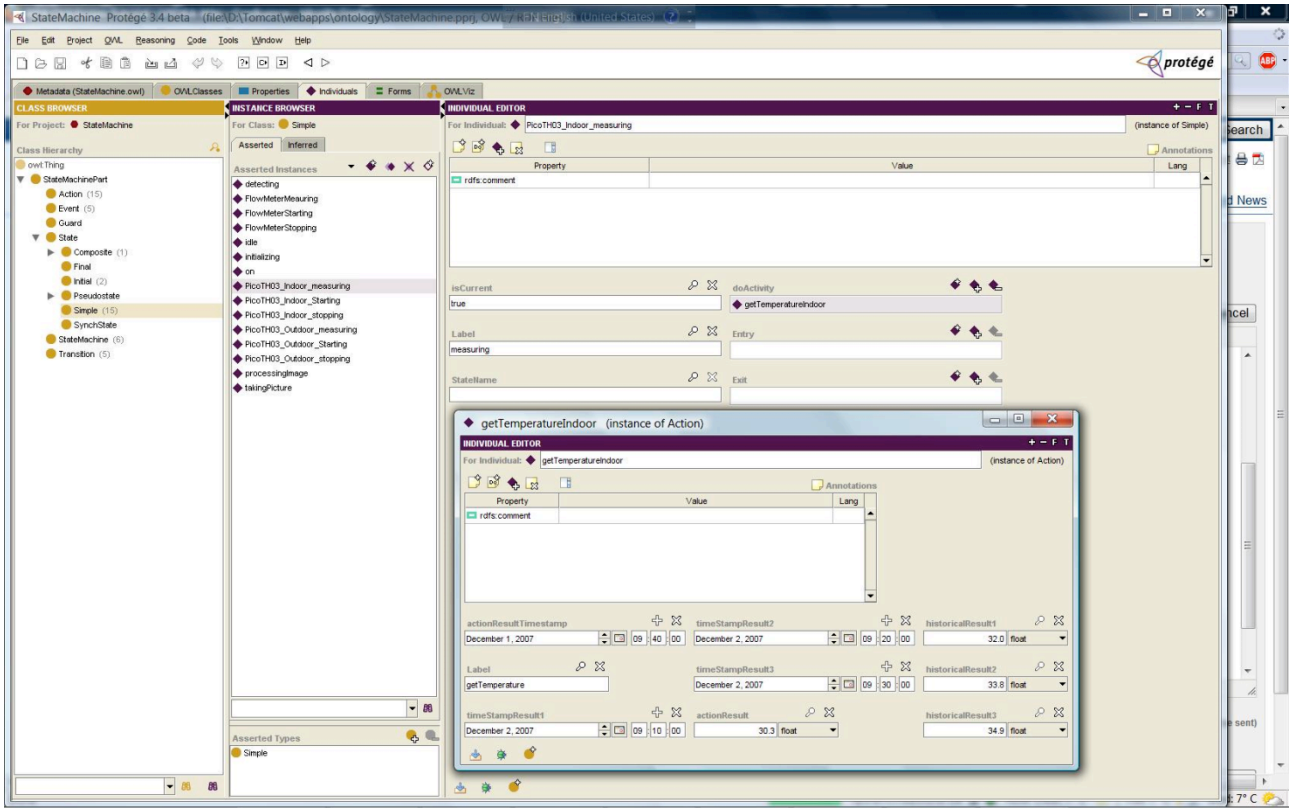
This tutorial will focus on the creation of StateMachine ontology, which is used for the creation of state machine stubs to handle device run time status changes, and also for the diagnosis and monitoring rules used for achieving self* properties.

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



Now lets look at how to create a state machine instance, take the simple Thermometer device (as shown in WP4 Limbo Tutorial) 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 ontology is achieved with the "Individuals" tab, by clicking on the concept (for exmaple "Simple"), and then click on the button "create instance" button. This is shown in the following figure. We are using Protege as the OWL development environment. As we need to differentiate every state for all device, we name the state 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.



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 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 you just created to reflect the transition direction.

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

11. Flamenco Probe Tutorial

General Description

The Flamenco Probe is written to monitor IP communication from a computer and currently uses Windump (and is thus only available on Windows for now). This corresponds to performing network layer reporting by instrumenting the service host. This is used among others in the Hydra Flamenco tool.

When started, the Flamenco Probe starts a DLL helper file which reports any communication to a designated port. When run, this DLL will produce a popup window saying that it has been started and can be stopped by clicking OK so DO NOT close this popup if you intend to keep running the Flamenco Probe.

Conversely, you must close it if you wish to stop or restart the Flamenco Probe. The DLL takes a few seconds to shut down so please wait before restarting. The probe then parses the messages that are communicated and, if they are relevant, publishes them to the Event Manager.

The use reads as follows (and will be printed if the sniffer is invoked with no arguments):

Usage:

```
     IPSniffer -help   prints this usage
or   IPSniffer [options]
```

Where options MUST include

```
   -port:<port#>          portnumber for the IPSniffer to use
   -eventManager:<ip>     ip where the eventmanager can be contacted

And where options CAN also include

   -reportAll:<bool>      true => reports traffic on all ports except
"exceptions"
                          false => reports only on ports in "exceptions"
                          default value is true
   -exceptions:<ports>    ;-seperated list of ports to be monitored or ignored
                            depending on the value of "reportAll"
                          default value is empty
```

Tutorial

1. Download and install the Event Manager according to the WP4 Event Manager Tutorial
2. Download Flamenco Probe from https://hydra.fit.fraunhofer.de/svn/trunk/sdk/flamenco/frlamencoprobe
3. Compile the probe using the Ant target "build" in the Ant file in the root of the project
4. You must now run the Flamenco Probe using parameters as specified above
    a. Port: Any unused port to be used for internal communication.
    b. eventManager: The address for the Event Manager. The default port is 8080 for the Event Manager
    c. reportAll: This is a boolean for defining on which ports the traffic is reported. If set to true all ports except the one given in port: and the exceptions given will be reported on. If set to false only the communications from the ports given in exceptions will be reported.
    d. exceptions: a list of ports as described just above. The list is ;-seperated.

5. Provoke communication on one of the ports that you are listening to. If you are listening to all ports you can just open a web page (see current limitations).
6. Check that the communication is reported to the EventManager... This is done easily by accessing the log file at .../EventManager/EventManager.log and checking that you have one or more lines resembling
7. 1161330 ...[btpool0-1] INFO com.eu.hydra.eventmanager.axis.EventManagerPortBindingImpl - EventManager publish on: /flamenco/socketwatch

with the important characteristics here being "/flamenco/socketwatch" at the end of the line. If you have a long log file you find the newest entries in the bottom.

Current limitations

If the sniffer is monitoring Firefox it somehow stops the communication. This does not occur with IE however.
Also the system only works on Windows at the moment.

12. List of referenced deliverables

D4.4 Hansen, K. M., J. Fernandes, et al. (2008). D4.4 Embedded AmI components prototype. Hydra Project Deliverable, IST project 2005-034891.

D4.5 Scholten, M. and L. Shi (2008). D4.5 Quality-of-Service Enabled HYDRA Middleware. Hydra Project Deliverable, IST project 2005-034891.

D4.7 Ingstrup, M., J. Fernandes, et al. (2009). D4.7 Embedded service DDK prototype and report. Hydra Project Deliverable, IST project 2005-034891.

D4.10 Al-Akkad, A.-A. and W. Zhang (2009). D4.10 Quality-of-Service Enabled Hydra Middleware. Hydra Project Deliverable, IST project 2005-034891.

D4.11 Hansen, K. M. and M. Ingstrup (2010). D4.11 Consolidated Embedded Architecture Report. Hydra Project Deliverable, IST project 2005-034891.(Delivery date M50)

D5.11 Milagro, F., P. Antolín, et al. (2010). D5.11 Wireless Network IDE Prototype. Hydra Project Deliverable, IST project 2005-034891.

D6.7, D6.9 Kostelnik, P. and M. Ahlsen (2009). D6.7 Device Ontologies Tools/Update Prototype. Hydra Project Deliverable. TUK, IST project 2005-034891.

D6.8, D610 Ahlsen, M., P. Rosengren, et al. (2009). D6.8 SOA Middleware Components Prototype for the SDK. Hydra Project Deliverable, IST project 2005-034891.

D3.14 Ingstrup, M., K. M. Hansen, et al. (2010). D3.14 Device Developer Tools Report. Hydra Project Deliverable, IST project 2005-034891.

D12.5 Badii, A., P. Kool, et al. (2008). D12.5 External Developers Workshops Teaching Material. Hydra Project Deliverable, IST project 2005-034891.