Contract No. IST 2005-034891

Hydra

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

D4.7 Embedded service DDK prototype and report

Integrated Project
SO 2.5.3 Embedded systems

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

Published by the Hydra Consortium
Lead Contractor: Fraunhofer FIT

24 February, 2009- version 1.1

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

Dissemination Level: Public

Document File:	D4.7 Embedded service DDK prototype and report
Work Package:	WP4
Task:	T4.2
Document Owner:	Joao Fernandes, Weishan Zhang, Klaus Marius Hansen, Mads Ingstrup

Document history:

Version	Authors	Date	Changes Made
0.1	Mads Ingstrup and Weishan Zhang (UAAR)	2009-01-15	Outline
0.2	Mads Ingstrup (UAAR)	2009-01-30	added description of TCP UDP, Bluetooth, and a bit about SOAP
0.3	Joao Fernandes (UAAR)	2009-02-04	SOAP over UDP and SOAP over Bluetooth
0.4	Francisco Milagro (TID) and Marco Vettorello (TCON)	2009-2-5	Limbo experiences
0.5	Joao Fernandes (UAAR)	2009-02-09	General overview
0.6	Weishan Zhang (UAAR)	2009-02-10	added semantic section, changed structure
0.7	Mads Ingstrup (UAAR)	2009-02-13	Added section on integration of limbo and ASL
0.8	Weishan Zhang (UAAR) and Joao Fernandes (UAAR)	2009-02-16	Executive summary, Hydra standard services, proof reading
0.85	Mads Ingstrup (UAAR)	2009-02-17	Proof reading with changes.
0.9	Klaus Marius Hansen (UAAR)	2009-02-17	Section on REST
1.0	Weishan Zhang (UAAR) and Joao Fernandes (UAAR)	2009-02-17	Structure changes, update of Bluetooth section, proof-reading changes
1.1	Joao Fernandes (UAAR) and Klaus Marius Hansen (UAAR) and Weishan Zhang (UAAR)	2009-02-24	Updates based on internal review

Internal review history:

Reviewed by	Date	Comments
Pablo Antolin (TID)	2009-02-19	Approved with comments
Marco Vettorello (TCON)	2009-02-24	Approved with minor changes

Contents

List of Figures

List of Tables

Executive Summary

This deliverable documents the achievements of Hydra within the area of embedded service DDK until month 32. The following tasks have been achieved compared to the last deliverable D4.2b (Hansen et al., 2008c). They are listed in the order in which they are documented in this deliverable:

- Add support in Limbo for generation of Web Service code using UDP as communication protocol.
- Add support in Limbo for generation of Web Service code using Bluetooth protocol as communication protocol.
- Add support for project resources generation in Limbo.
- Add support for Hydra standard services generation, which will help to Hydra-enable a device.
- Add initial support for RESTful web services.
- Evaluations of Limbo by other partners, reported as experiences.
- Enhanced support for Limbo architecture and configuration validation, and support for component interface mismatch.
- Support for consideration of Quality of Service (QoS) for pervasive service code generation, in which communication protocols and power consumption characteristics are considered.
- Support for enabling/disabling devices and services with architecture scripting language, which itself is described in deliverable D4.8 (Ingstrup and Zhang, 2008).
- Six papers are published related to this task (SEKE 2008 (two papers), SASO 2008 (two papers), MRC 2008 workshop, IOT&S workshop, APSEC 2008), as in the Appendix. Limbo was also demonstrated in ICSR 2008 in the tool demonstrate session.

The following sub-tasks of T4.2 remain outstanding but should be finished by month 36, as according to the plan in the new DOW (V7.28):

- Add full support in Limbo for generation of RESTful services.
- Add support in Limbo for generation of .NET services.
- Performance evaluations with REST-based services and SOAP web services.

The role of this deliverable in Hydra The Device Development Kit (DDK) is important to help developers to implement pervasive web services in an efficient way. This includes the work on generating stub and skeleton code for a device, and on enabling a device for Hydra, mainly make the device UPnP enabled and having standard Hydra operations. Enabling a device for Hydra also includes the feature of getting an HID at runtime, which Limbo will also help accomplish, as explored in the implementation of self-management work reported in deliverable D4.3 (Ingstrup et al., 2008) and D4.8 (Ingstrup and Zhang, 2008). Specifically, this deliverable serves for Hydra:

- Hydra enabling a device, any device including D0 to D4 devices. The devices without sufficient resources for holding required features on the device itself will have a proxy.
- Facilitating the development of pervasive web services. Limbo is a generic tool to ease the development of pervasive web services, and can be used for web services development for any devices, as reported in our papers (Hansen et al., 2008d) (Hansen et al., 2008a) (Hansen et al., 2008b) and former deliverables D4.2 (Hansen et al., 2007) and D4.2b (Hansen et al., 2008c).
- Facilitate the development of other Hydra components, like the self-management component in Hydra, as reported in D4.3 (Ingstrup et al., 2008) and D4.8 (Ingstrup and Zhang, 2008). Limbo is used also in the Network Manager development.
- Paving the way for considering the quality of service in the development of pervasive web service. As a starting point, we are considering different web service transportation protocols, and the power consumption of web service calls. This will enhance the work on self-management, as well as on the QoS manager of Hydra.
- Architectural Scripting Language can be used to actuate a device or a service, as reported in details in D4.8 (Ingstrup and Zhang, 2008).

1 Introduction

1.1 Components Overview

The following components are provided as a result of the work documented in this deliverable.

- Limbo, a web service compiler for resource constrained devices. Limbo supports developers by generating web service code, code for making the device discoverable via UPnP protocol, Hydra generic services, state machine stubs and probes for diagnostics proposes. In Hydra, devices are classified into 5 categories as in D5.4 (Sperandio et al., 2007), namely D0 to D4 devices as shown in Table 1.1.In the two last columns, we listed possible usage of Limbo for each of the device types and a list of devices that Limbo has been used to generate code for them.

- Architectural Scripting Language. This is documented in D4.8 (Ingstrup and Zhang, 2008), and can be used to start and stop devices, deploy and undeploy code to them, start and stop services and for some platforms also establish bindings among services or runtime components.

Table 1.1: Devices classification in Hydra

Type	**Description**	**Limbo usage**	**Devices Limbo has been used to generate code for them**
D0	Non-HED. Specific communication protocol (BT, ZigBee). Need of a proxy in D4	Limbo can be used in order to generate proxy code, by generating JSE OSGi code.	– Pico TH03 thermometer service. – Grundfos Magna 32 pump service. – Abloy EL582 door lock service. – Java SunSPOT thermometer service. – UC-322PBT A&D bloodpressure service. – UC-321PBT A&D weight scale service. – Xbee Series 2 ZB OEM thermometer, accelerometer, humidity, air pressure and ambient light services. – smartLAB genie glugose service. – Promedia 705IT BT bloodpressure service.
D1	Non-HED. WS support.	Use Limbo to generate code to embedd in the device, this code can be JSE standalone or JME, corresponding to the available software platform on a device.	– Nokia6630 camera, lock and thermometer services. – NokiaN80 SMS service. – NokiaN95 8GB lock service. – Nokia3110 lock service.
D2	HED. Specific communication protocol (Bluetooth, Zigbee). Need of a bridge (dedicated or in D3-D4)	If the device supports Bluetooth communication and can host JSE or JME, Limbo generate code using bluetooth as communication protocol used by the JSE or JME server code running on this device. And proxy code (JSE OSGi) and bluetooth client code are deployed on D3-D4 device in order to communicate with the physical device.	
D3	HED. Bridge hosting	Powerful devices which can host JSE standalone or JSE OSGi code for proxying D0,D1,D2 devices.	
D4	Gateways. Proxy and bridge hosting	Powerful devices which can host JSE standalone or JSE OSGi code for proxying D0,D1,D2 devices.	

2 SOAP transportation over different protocols

In this section we describe the protocols that are supported for use with the Hydra code base. First, we give an overview of each protocol, and then describe the combinations of protocols that are supported. Next, we describe in detail the design and implementation of SOAP over UDP and Bluetooth.

2.1 Introduction to the supported protocols

The immense success of the Internet has established its protocol suite as a de facto standard for use with a range of application level protocols. Yet as the proliferation of mobile and ubiquitous computing gains momentum, a more heterogeneous set of devices needs to be supported, and with those comes a greater variation in what protocol features are desirable. For instance, the Bluetooth protocol was developed with a mobile computing environment in mind to allow discovery of devices and services.

In order to understand the relation among the protocols we consider it is useful to map them to the open systems interconnection reference model, or OSI model. It defines seven layers each responsible for realizing certain functions required for data communication over a network of devices. The OSI model is shown in figure 2.1.[1]

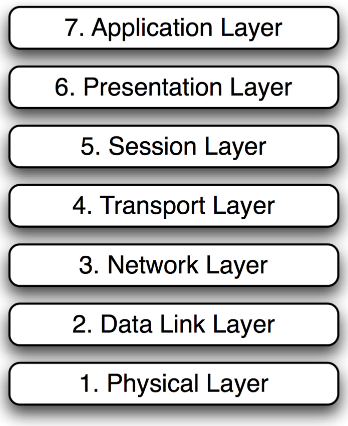


Figure 2.1: The seven layers of the OSI reference model.

The TCP and UDP are part of the IP protocol suite used on the Internet. Although these protocols do not conform strictly with the OSI model, it is nevertheless easier to understand the services provided by them in relation to this model.

2.1.1 TCP

The Transport Control Protocol, or TCP, is a transport layer protocol. TCP is connection-oriented which means that clients of the protocol send requests for connections to the re-

[1]See Cisco's Internet working Technology Handbook for more information on the protocols described in this chapter. Available from http://www.cisco.com/en/US/docs/internetworking/technology/handbook/Intro-to-Internet.html

ceptors and use the established connection to transfer data. A connection consists of end-points, or Internet sockets, which are stateful and once established can be used to send arbitrary amounts of data before being closed again. The TCP protocol guarantees reliable data transfer and that the data is received in the same order as it was sent. Further, the protocol controls the flow of data so that the transfer rate is that of the slowest node, in order to guarantee reliability.

2.1.2 UDP

The User Datagram Protocol, or UDP, is also a transport layer protocol, but it provides a different quality of service from TCP. UDP is connection-less, or message oriented, and used by sending datagrams, individually addressed packages of data. In contrast to TCP it does not provide re-transmission when packages are lost or contain faults, and as such it is unreliable. In addition, packages may arrive out of order. UDP is lightweight compared with TCP and has less overhead in the transmission of data.

2.1.3 Bluetooth

The Bluetooth protocol is a short-range radio link specification. It was designed to be robust, have low complexity, cost and power consumption. In terms of the OSI model Bluetooth implements the physical layer and parts of the data link layer. In addition to this, a set of specialized protocols that cannot be mapped directly to the OSI model exist. These are for applications such as cable replacement (RFCOMM), service discovery (SDP), and telephony (TCS-BIN) among others. A number of protocols that are not part of Bluetooth have nevertheless been adopted and are frequently used in combination with Bluetooth. The Point-to-Point Protocol is often used in combination with Bluetooth to provide the Data Link layer functionality required in order to use the Internet Protocol (IP) and thus UDP or TCP on top of Bluetooth.

2.1.4 SOAP

The Simple Object Access Protocol[2], or SOAP, is used for invoking web services. It commonly makes use of the HTTP application layer protocol, which in turn uses TCP. It is XML based which is beneficial because it is, at least in principle, human readable, but since XML tends to be verbose it is also the cause of significant overhead compared with e.g. Corba.

2.2 SOAP over UDP

A study conducted on D5.9 (Sperandio et al., 2008), we performed several tests and evaluated the performance of SOAP over TCP and UDP. In these tests we measured the average Round-Trip-Time (RTT), Throughput and in UDP the Goodput. The results showed that when both client and server are using wired connection the performance of TCP and UDP is very similar. But in the case of having client and/or server using wireless connection, despite being a non-reliable protocol, the performance of UDP is much better than TCP. This motivate us to add the support for different communication protocols in Limbo and leave the choice of the most adequate protocol by utilizing QoS Ontologies which will consider

[2]This was initially the official acronym, but appears to have been dropped in later versions of the specification.

```
try {
    Socket clientSocket = new Socket(this.host, this.port);
    OutputStream cos = clientSocket.getOutputStream();
    cos.write(SOAPMessage.getBytes());
    InputStream cis = clientSocket.getInputStream();
    ...
    n = cis.read(buffer);
    request = new String(buffer, 0, n);
    ...
} catch (Exception e) {
    e.printStackTrace();
}
```

Figure 2.2: JSE Client Stub code using TCP connection

parameters such as network type, network bandwidth and network traffic and give the most suitable protocol to Limbo as a parameter, as discussed in Chapter 8.

As a first step, we defined a new parameter in Limbo that specifies the protocol used for communication between client and server, which at current stage is given by the developer to Limbo.

As the next step we identified which classes have to be changed in order to support different kinds of protocols. The classes are the client stubs and server skeletons, these classes contain code specifying the communication used.

2.2.1 Client stub for different protocols

Figure 2.2 shows part of the code of a JSE client stub using TCP connection, and Figure 2.3 shows a JSE client stub using UDP as communication protocol, where a socket timeout is set to a value greater than zero, this timeout was introduced because of non-reliability of UDP protocol, so if the client does not receive the response from the server in "timeout" seconds the call is given as lost. In case of not setting this timeout, in case of loosing a message the client would be blocked waiting for a response from the server.

Figure 2.4 shows a JME client using TCP connection, the client makes use of Socket-Connection class provided by CLDC1.1 (Connected Limited Device Configuration version 1.1) libraries. CLDC provides also libraries for UDP connections, as we can see in Figure 2.5, this client is making use of DatagramConnection and Datagram classes in order to connect to the server, send the request, and finally receive the response.

2.2.2 Server skeleton for different protocols

As for the JSE server skeleton using TCP connection Figure 2.6 shows the creation of the TCP socket, and the loop of waiting for request and return the response to the client. The server using a UDP connection uses a DatagramSocket variable instead, and Datagram-Packet variables that encapsulate the received and sent data, as shown in Figure 2.7.

Figure 2.8 shows us parts of a Limbo generated server skeleton JME version using a TCP connection, defined by a ServerSocketConnection variable. For an UDP version of the JME server, as shown in Figure 2.9 the server makes use of DatagramConnection variable and Datagram variables in order to communicate with clients.

```
try {
   InetAddress server = InetAddress.getByName(this.host);
   DatagramSocket clientSocket = new DatagramSocket();
   clientSocket.setSoTimeout(timeout);
   ...
   DatagramPacket dp = new DatagramPacket(SOAPMessage.getBytes(),
                        SOAPMessage.getBytes().length, server, this.port);
   clientSocket.send(dp);
   DatagramPacket receive = new DatagramPacket(buffer, buffer.length);
   clientSocket.receive(receive);
   String request = new String(buffer).trim();
   ...
 } catch (Exception e) {
   e.printStackTrace();
}
```

Figure 2.3: JSE Client Stub code using UDP connection

```
try {
   SocketConnection clientSocket = (SocketConnection)
                Connector.open("socket://" + this.host + ":" + this.port);
   OutputStream cos = clientSocket.openOutputStream();
   cos.write(SOAPMessage.getBytes());
   InputStream cis = clientSocket.openInputStream();
   ...
   n = cis.read(buffer);
   response = new String(buffer, 0, n);
   ...
} catch (Exception e) {
   e.printStackTrace();
}
```

Figure 2.4: JME Client Stub code using TCP connection

```
try {
   DatagramConnection clientSocket = (DatagramConnection)
                    Connector.open("datagram://"+this.host+":"+this.port);
   Datagram dgram = clientSocket.newDatagram(SOAPMessage.getBytes(),
                                                   SOAPMessage.length());
   dgram.setAddress(dgram);
   clientSocket.send(dgram);
   ...
   Datagram response = clientSocket.newDatagram(buffer, buffer.length);
   clientSocket.receive(response);
   request = new String(response.getData());
   ...
} catch (Exception e) {
   e.printStackTrace();
}
```

Figure 2.5: JME Client Stub code using UDP connection

```
ssc = new ServerSocket( this.port ) ;
...
while(!stopped){
   ...
   sc = ssc.accept() ;
   is = sc.getInputStream();
   OutputStream cos = sc.getOutputStream();
   ...
   String request = sb.toString();
   System.out.println("Request Received. Attendance in progress...");
   result = ((abloy_el582EndPoint)
          abloy_el582EndPoint.getEndPoints().elementAt(i)).handleRequest(
                        p.getRequest(), SOAPAction, "servicebegin", "", cos);
   if(result != null){
      cos.write(result.getBytes());
      cos.flush();
      cos.close();
      i = th03EndPoint.getEndPoints().size();
   }
   ...
}
```

Figure 2.6: JSE Server Skeleton code using TCP connection

```
sc = new DatagramSocket( this.port );
...
while(!stopped){
   ...
   DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
   sc.receive(dp);
   InetAddress clientHost = dp.getAddress();
   int clientPort = dp.getPort();
   String request = new String(buffer).trim();
   ...
   result = ((abloy_el582EndPoint)
            abloy_el582EndPoint.getEndPoints().elementAt(i)).handleRequest(
                       p.getRequest(), SOAPAction, "servicebegin", "", null);
   if(result != null){
      DatagramPacket response = new DatagramPacket(result.getBytes(),
                      result.getBytes().length, dp.getAddress(), clientPort);
      sc.send(response);
      i = abloy_el582EndPoint.getEndPoints().size();
   }
   ...
}
```

Figure 2.7: JSE Server Skeleton code using UDP connection

```
ssc = (ServerSocketConnection)Connector.open("socket://:9002");
SocketConnection sc = null;
...
while(true){
   sc = (SocketConnection) ssc.acceptAndOpen();
   is = sc.openInputStream();
   OutputStream cos = sc.openDataOutputStream();
   ...
   String request = sb.toString();
   ...
   result = ((abloy_el582EndPoint)
          abloy_el582EndPoint.getEndPoints().elementAt(i)).handleRequest(
                        p.getRequest(), SOAPAction,"servicebegin", "", cos);
   if(result != null){
      i = abloy_el582EndPoint.getEndPoints().size();
      cos.write(result.getBytes());
      cos.flush();
      cos.close();
   }
   ...
}
```

Figure 2.8: JME Server Skeleton code using TCP connection

```
ssc = (DatagramConnection)Connector.open("datagram://:9002");
...
while(true){
   ...
   Datagram dgram = ssc.newDatagram(buffer, buffer.length);
   ssc.receive(dgram);
   String request = new String(dgram.getData()).trim();
   ...
   result = ((abloy_el582EndPoint)
           abloy_el582EndPoint.getEndPoints().elementAt(i)).handleRequest(
                        p.getRequest(), SOAPAction,"servicebegin", "", null);
   if(result != null){
      i = abloy_el582EndPoint.getEndPoints().size();
      Datagram response = ssc.newDatagram(result.getBytes(),result.length());
      response.setAddress(dgram);
      ssc.send(response);
   }
   ...
}
```

Figure 2.9: JME Server Skeleton code using UDP connection

Having identified the differences of the generated code using TCP and UDP code, the next step was to design a generalized approach that will allow the developer to add new protocols in an easy way.

2.2.3 Handling variabilities of transportation protocols

Due to the architecture of the generated services the classes that differ when using TCP and UDP are the server skeletons and client stubs. This makes it easier to allow inclusion of new protocols by the developer. It also allows the combination of skeletons using different protocols at runtime for the same web service.

For this reason we designed a new interface named ServerProtocol for server sides, containing the following operations:

- void createConnection(): Creates a new connection.
- String receive(): blocks until a new request arrived, parses the received message and returns it to the server skeleton.
- void send(String response): Sends the given parameter response to the client.
- void closeConnection(): Closes the connection.

When creating a new skeleton for the web service the developer gives a new parameter specifying the protocol that will be used. This parameter can be an instance of TCPProtocol or an instance of UDPProtocol.

In order to add support for new protocols the developer will have to implement a class specifically for the new protocol, and this class must implement the ServerProtocol interface.

It is possible, at runtime, to have multiple skeletons that use different communication protocols. One example of this would be a web service with one skeleton accepting TCP connections and another accepting UDP connections. If the developer wants to register such a service in the NetworkManager, there is the need to create two HIDs, since the server will have two different endpoints, although these two endpoints refer to the same web service. The support for communication using different protocols through the Network Manager will be a case of study from WP4 and WP5 in the future.

For client sides we added a new ClientProtocol interface, this interface contains the following methods:

- String communicateWithServer(String SOAPMessage, String host, int port): Sends the SOAPMessage to the given url, waits for response and returns the response to the client stub.

Two classes were also developed for the two protocols (TCPProtocol and UDPProtocol classes) that implement the ClientProtocol interface. After this we added a new parameter to the client stub constructor, specifying the protocol the client will use for communicating with the server. This parameter can be either a TCPProtocol instance or a UDPProtocol instance. There is also the possibility for the developer to add new protocols. In order to do so he or she has to implement a new class for the specific protocol, and this class must implement the ClientProtocol interface.

With the scheme described here, a client of a web-service needs to know what protocol the service is using. As for the process of choosing the correct protocol from the client side: In case of registration from the server side in the Network Manager, the server could provide information about the communication protocol it is using to the Network Manager. This way, when the client wants to communicate with the server it looks up in the Network Manager asking which communication protocol the server is currently accepting. This will be also case of study from WP4 and WP5 in the future considering QoS requirements.

2.3 SOAP over Bluetooth

The study made in D5.9 (Sperandio et al., 2008) also included performance and power consumption tests on SOAP over Bluetooth, the performance tests showed that Bluetooth does not perform as well as TCP or UDP. But, as expected, the power consumption tests Bluetooth showed to be the solution that consumes less power when comparing with wireless solution. Since power consumption is one of the dominating constraints for resource constrained devices we decided to add support in Limbo for services using bluetooth as communication.

For JME services, the libraries jsr82 (Java Specification Requests 82) and CLDC1.1 provide classes for bluetooth communication. As for JSE services we used Bluecove library, which interfaces with Mac OS X, WIDCOMM, BlueSoleil and Microsoft Bluetooth stack in Windows XP SP2 or Windows Vista and WIDCOMM and Microsoft Bluetooth stack on Windows Mobile.

Figure 2.11 shows us parts of a client using Bluetooth as communication protocol, Figure 2.10 shows us parts of code of a server skeleton using Bluetooth as communication, the code is the same for JSE or JME versions, since the needed classes have the same naming, but provided by different libraries.

During the development of web services over bluetooth we experienced some failures, when in the server side we closed the connection and in the client side the response was still not received. In order to overcome this failure we implemented a mechanism of acknowledgments, as shown in Figure 2.13. The client sends the SOAP request and awaits the response from the server, after receiving the response from the server the client sends an acknowledgment to the server, which the server also acknowledges to the client again. After the exchange of acknowledges both client and server can close the connection.

For adding bluetooth support in Limbo we followed the same design as for the support of UDP and TCP protocols, for that, in server sides we implemented a new class named BTProtocol, which also implements the ServerProtocol interface. As for client side the same procedure was done by implementing BTProtocol class that implements the ClientProtocol interface. But in the case of Bluetooth clients we will always need to run the discovery process in order to be able to retrieve the server url and be able to communicate with it. To do that we generate an extra class, a specific bluetooth client class that runs the discovery process and passes the url to the service to the client stub. Figure 2.12 shows us a class diagram for the current supported protocols for both client and server sides.

```
LocalDevice device = LocalDevice.getLocalDevice();
device.setDiscoverable(DiscoveryAgent.GIAC);
String url = "btspp://localhost:"+UUID+";name=abloy_el582";
StreamConnectionNotifier notifier = (StreamConnectionNotifier)
                                                        Connector.open(url);
...
while(!stopped){
   ...
   StreamConnection sc = (StreamConnection) notifier.acceptAndOpen();
   is = sc.openInputStream();
   OutputStream cos = sc.openDataOutputStream();
   result = ((abloy_el582EndPoint)
             abloy_el582EndPoint.getEndPoints().elementAt(i)).handleRequest(
                                p.getRequest(), SOAPAction, null, null, cos);
   if(result != null){
      cos.write((result+"/$").getBytes());
      cos.flush();
      i = abloy_el582EndPoint.getEndPoints().size();
   }
   ...
```

Figure 2.10: JSE/JME Server Skeleton code using Bluetooth connection

```
StreamConnection stream = (StreamConnection) Connector.open(this.url);
OutputStream cos = stream.openOutputStream();
cos.write(SOAPMessage.getBytes());
cos.flush();
InputStream cis = null;
try {
   cis = stream.openInputStream();
}catch(Exception e) {
...
}
do {
  bytesRead = cis.read(buffer);
  if (bytesRead > 0) {
     response = response.concat(new String(buffer, 0, bytesRead));
  }
} while (!response.endsWith("/$"));
...
```

Figure 2.11: JSE/JME Client Stub code using Bluetooth connection

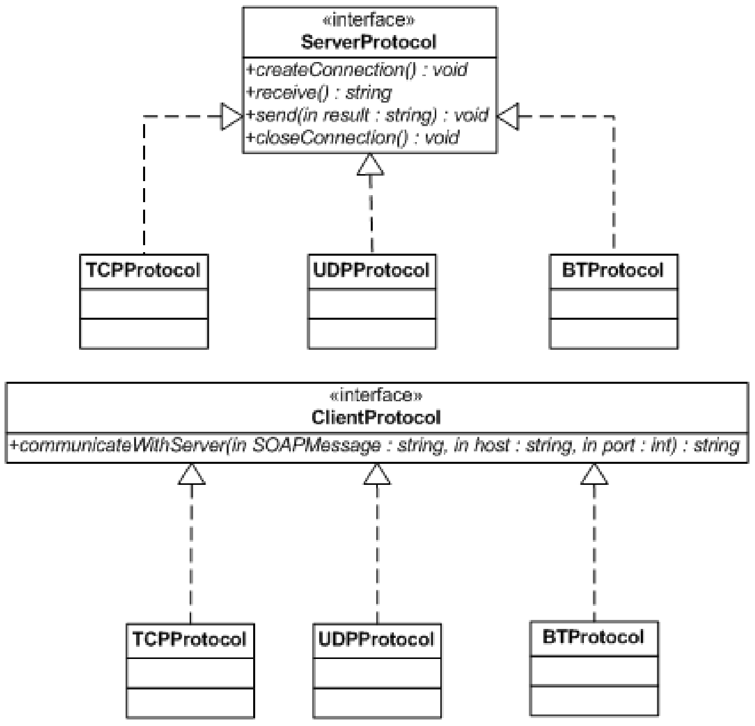

«interface»
ServerProtocol
+createConnection() : void
+receive() : string
+send(in result : string) : void
+closeConnection() : void
TCPProtocol
UDPProtocol
BTProtocol
«interface»
ClientProtocol
+communicateWithServer(in SOAPMessage : string, in host : string, in port : int) : string
TCPProtocol
UDPProtocol
BTProtocol

Figure 2.12: Class diagram for Protocol support in Limbo.

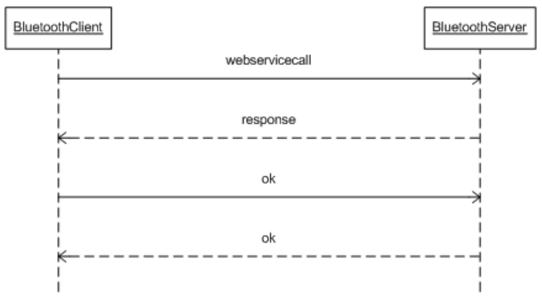


Figure 2.13: Web Service call using Bluetooth communication.

3 REpresentational State Transfer (REST)

3.1 REST Overview

REpresentational State Transfer (REST) is an architectural style on which central web architectures such as HTTP are built (Fielding, 2000). It is also increasingly used on an application level for communication with services on top of HTTP[1], the rationale being that by following the architectural style of the Web, qualities such as scalability and interoperability are to an extent supported.

As an architectural style, REST may be described through the constraints it puts on an architecture (Fielding, 2000, chap 5):

- *Client/Server*. REST is client-server-based in that separates user interface concerns (at the client) from storage concerns (at the server)
- *Stateless*. A request from the client to the server must contain all data for the server to understand the request and should not use state information on the server
- *Caching*. Servers may mark data as cacheable and thus allow clients to reuse responses
- *Uniform Interfaces*. Component interaction is through uniform interfaces for identifying, manipulating, describing, and navigating resources
- *Layering*. Systems should be allowed to be composed of layers in which components are not visible "through" layers
- *Code-on-demand*. Client functionality can be extended by downloading and executing code. This is an optional constraint on the REST style

The architectural elements of REST are data elements, connectors, and components. In contrast to distributed object styles (of which SOAP is an example), the key abstraction in REST are data elements in the form of *resources* (e.g., a document) of which current or intended *representations* are transferred. Resources are accessed through uniform interfaces of connectors and a key point is that hypermedia is used to drive the application state, i.e., links in representations inform clients (and servers) of what current and possible states are.

The REST architectural style (in particular as realized on top of HTTP) is of interest in Hydra in that it leverages existing Web features (such as caching or exception handling through error codes), imposes only small communication overhead (on top of, e.g., HTTP), and forces an application structuring in terms of state-chaning resources. The first two characteristics potentially makes implementations more lightweight (and well-integrated with existing web infrastructure) and the last characteristic is arguably a good match with many embedded systems.

[1] A Google search on "rest "web services"" 2009-02-17, e.g., gave 4,1 million hits. A prominent commercial example of RESTful services are Amazon's web services that both exist in SOAP and REST variants

3.2 Supporting REST in Limbo

3.2.1 Describing REST Services using WSDL

Limbo relies on a service description (currently in the form of WSDL files) to generate web service support. As REST is an architectural style, it is quite abstract and there is no standard (or default) mapping to implementations. In our context we are concerned with HTTP and thus map REST to that protocol. We have chosen also to describe REST services using WSDL since this allows for a uniform description format for services that Limbo may generate code for.

The WSDL 1.1 specification is available at `http://www.w3.org/TR/wsdl`. A WSDL document is an XML document that describes how a web service may be accessed.

Figure 3.1 illustrates the concepts of WSDL as an ontology. The elements that are concrete in the sense that they are bound to specific network protocols and message formats are marked as so with a stereotype; conversely, abstract elements are also marked with a stereotype. *Operation*, *Input*, *Output*, and *Fault* may be both abstract and concrete depending on whether they are part of a *PortType* or *Binding*.

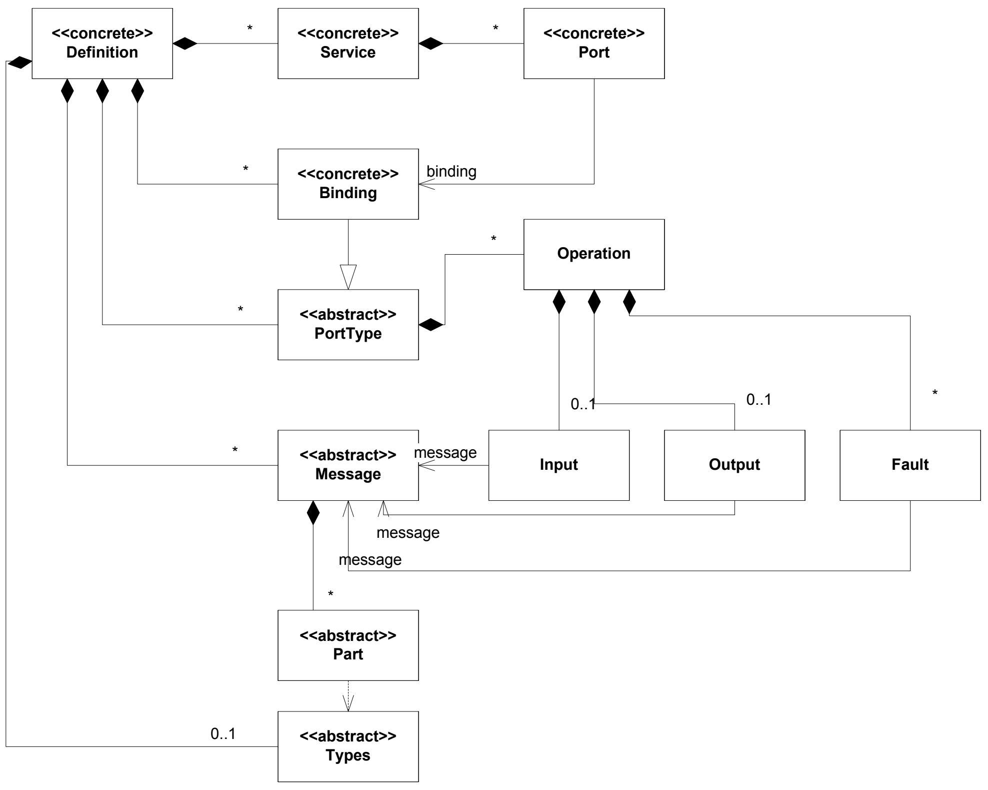


Figure 3.1: A UML-based ontology of WSDL concepts

The WSDL specification gives a grammar for WSDL documents which is shown below (`<wsdl:documentation .../>` productions removed):

```
<wsdl:definitions name="nmtoken"? targetNamespace="uri"?>
```

```
        <import namespace="uri" location="uri"/>*
        <wsdl:types> ?
            <xsd:schema .... />*
            <-- extensibility element --> *
        </wsdl:types>
        <wsdl:message name="nmtoken"> *
            <part name="nmtoken" element="qname"? type="qname"?/> *
        </wsdl:message>
        <wsdl:portType name="nmtoken">*
            <wsdl:operation name="nmtoken">*
               <wsdl:input name="nmtoken"? message="qname"/>?
               <wsdl:output name="nmtoken"? message="qname"/>?
               <wsdl:fault name="nmtoken" message="qname"/> *
            </wsdl:operation>
        </wsdl:portType>
        <wsdl:binding name="nmtoken" type="qname">*
            <-- extensibility element --> *
            <wsdl:operation name="nmtoken">*
               <-- extensibility element --> *
               <wsdl:input> ?
                   <-- extensibility element -->
               </wsdl:input>
               <wsdl:output> ?
                   <-- extensibility element --> *
               </wsdl:output>
               <wsdl:fault name="nmtoken"> *
                   <-- extensibility element --> *
               </wsdl:fault>
            </wsdl:operation>
        </wsdl:binding>
        <wsdl:service name="nmtoken"> *
            <wsdl:port name="nmtoken" binding="qname"> *
               <-- extensibility element -->
            </wsdl:port>
            <-- extensibility element -->
        </wsdl:service>
        <-- extensibility element --> *
    </wsdl:definitions>
```

The *extensibility elements* of the grammar are intended for enabling, e.g., the specification of concrete bindings. The WSDL specification specifies three bindings (SOAP, HTTP, and MIME) but mandates the use of neither. This section discusses how we will support REST in Limbo. There are at least the following issues to take care of:

- REST is an architectural style and not a standard or a specification. This means that we need to have a specific interpretation of REST
- WSDL operations are somewhat at odds with REST URIs in that the latter are identifying resources rather than operations on resources
- REST input corresponds to one of the HTTP request forms (e.g., GET, PUT, POST, or OPTIONS), but WSDL describes the use of general XML which may be a problem as input to operations

- The HTTP binding in the WSDL specification is not the solution although it comes close: it specifies that a binding uses a specific verb for all operations

Thus we define a new binding, *REST*, that is derived from the HTTP binding, having the following schema:

```
<schema xmlns="http://www.w3.org/2000/10/XMLSchema"
        xmlns:rest="http://schemas.xmlsoap.org/wsdl/rest/"
        targetNamespace="http://schemas.xmlsoap.org/wsdl/rest/">
   <element name="address" type="rest:addressType">
      <complexType/>
   </element>
   <complexType name="addressType">
      <attribute name="location" type="uriReference" use="required"/>
   </complexType>
   <element name="binding"/>
   <element name="operation" type="rest:operationType"/>
   <complexType name="operationType">
      <attribute name="method" type="NMTOKEN" use="required"/>
      <attribute name="location" type="uriReference" use="required"/>
   </complexType>
</schema>
```

This gives the following structure of REST-bound WSDL definitions:

```
<definitions .... >
    <binding .... >
        <rest:binding/>
        <operation .... >
           <rest:operation location="uri" method="nmtoken"/>
        </operation>
    </binding>
    <service ...>
        <port .... >
            <rest:address location="uri"/>
        </port>
    </service>
</definitions>
```

The extension elements have the following semantics:

rest:address. The location attribute specifies a base URI for the port. This base URI is combined with the URI for individual operations to provide a URI for an operation

rest:operation. The location attribute must be a relative URI that specifies the location of the resource/representation that the operation addresses. Depending on whether the verb attribute is GET or PUT/POST, the input operations will be either encoded in the request URI or be part of the request entity body

It should be noted that although a service is described using the REST binding they are not necessarily RESTful (just as SOAP service description in WSDL do not necessarily follow good software engineering practices). In fact simple RPC XML applications can easily be described using the binding. Rather, it is up to the developer to design the interfaces and XML datatypes so that the instances of the datatypes are representations of resources, HTTP verbs are used in a reasonable way, and interfaces allows for a hypertext-based interaction with resources.

3.2.2 Implementing REST Support in Limbo

Our current implementation has been directed towards supporting REST services on a specific platform (in this case the LEGO NXT platform[2]) meaning that the code generated is specific to this embedded platform, but the design of the generation can be reused.

Figure 3.2 shows a module view of the current Limbo REST implementation. Gray classes are interfaces from the base Limbo and white classes are the main extension classes. The standard implementations of the Limbo interfaces are not shown for clarity. Next, Figure 3.3 shows how the REST implementation is deployed. All dependencies shown

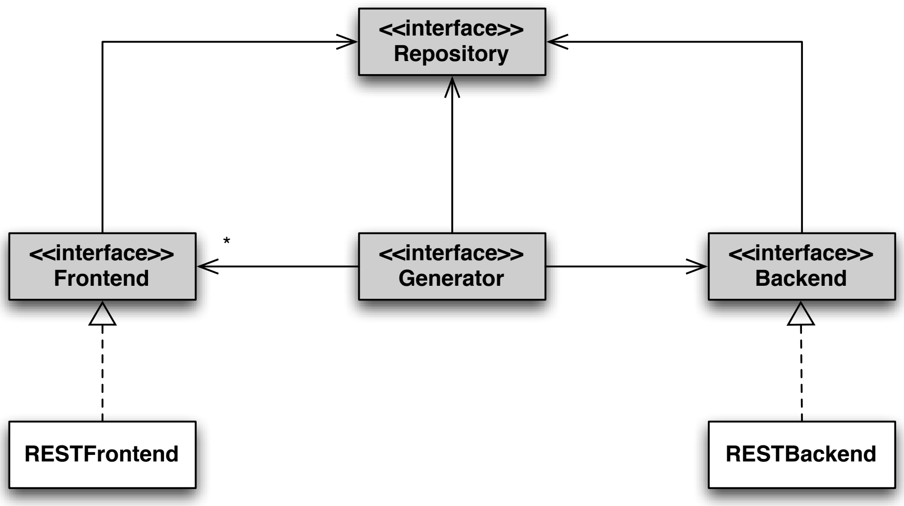


Figure 3.2: Class diagram of Limbo REST implementation

are made through Declarative Services declarations. Further, Figure 3.4 shows a dynamic view of how the REST Frontend and REST Backend are invoked by Limbo. In the current implementation, the REST Frontend is responsible for checking whether REST is applicable in the compilation situation (that it is indeed a WSDL file with a REST binding that is being processed) and for setting parameters. To do this, the Frontend uses the Repository (which it receives a reference to through OSGi's Declarative Services). The REST Backend, correspondingly, uses the REST Frontends checks and generates a REST-based web service skeleton (currently only for the Lego NXT platform).

3.2.3 A REST Example

The following shows a very simple example of a description of a REST service:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions targetNamespace="http://nxtservice.hydra.eu.com"
             name="NXTService" xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:impl="http://nxtservice.hydra.eu.com"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:rest="http://hydramiddleware.eu/rest/">
```

[2]http://mindstorms.lego.com/

```
<types>
 <xsd:schema>
    <xsd:import namespace="http://nxtservice.hydra.eu.com" schemaLocation="NXTService_
  </xsd:schema>
</types>

<message name="lightsensorRequest"/>
<message name="lightsensorResponse">
  <part name="result" type="impl:lightsensor"/>
</message>

<message name="touchsensorRequest"/>
<message name="touchsensorResponse">
  <part name="result" type="impl:touchsensor"/>
</message>

<portType name="NXTServicePort">
  <operation name="lightsensor">
    <input message="impl:lightsensorRequest" name="lightsensorRequest"/>
    <output message="impl:lightsensorResponse" name="lightsensorResponse"/>
  </operation>
  <operation name="touchsensor">
    <input message="impl:touchsensorRequest" name="touchsensorRequest"/>
    <output message="impl:touchsensorResponse" name="touchsensorResponse"/>
  </operation>
</portType>

<binding name="NXTServiceBinding" type="impl:NXTServicePort">
  <operation name="lightsensor">
    <rest:operation location="/sensor/light" method="GET"/>
    <input name="lightsensorRequest"/>
    <output name="lightsensorResponse"/>
  </operation>
  <operation name="touchsensor">
    <rest:operation location="/sensor/touch" method="GET"/>
    <input name="touchsensorRequest"/>
    <output name="touchsensorResponse"/>
  </operation>
</binding>

<service name="NXTService">
  <port name="NXTServicePort" binding="impl:NXTServiceBinding"/>
  <rest:address location="/"/>
</service>
</definitions>
```

The referenced XML schema is shown next:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0" targetNamespace="http://nxtservice.hydra.eu.com"
                         xmlns:impl="http://nxtservice.hydra.eu.com"
                         xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:complexType name="lightsensor">
   <xs:sequence>
       <xs:element name="port" type="xs:integer" minOccurs="1"/>
```

```
            <xs:element name="value" type="xs:integer" minOccurs="1"/>
        </xs:sequence>
     </xs:complexType>
     <xs:complexType name="touchsensor">
        <xs:sequence>
            <xs:element name="port" type="xs:integer" minOccurs="1"/>
            <xs:element name="value" type="xs:boolean" minOccurs="1"/>
        </xs:sequence>
     </xs:complexType>
      </xs:schema>
```

To fully utilize the nature of REST, this description could, e.g., be extended with methods for listing the sensors of the robot. The XML listing would then contain URIs of the individual sensors.

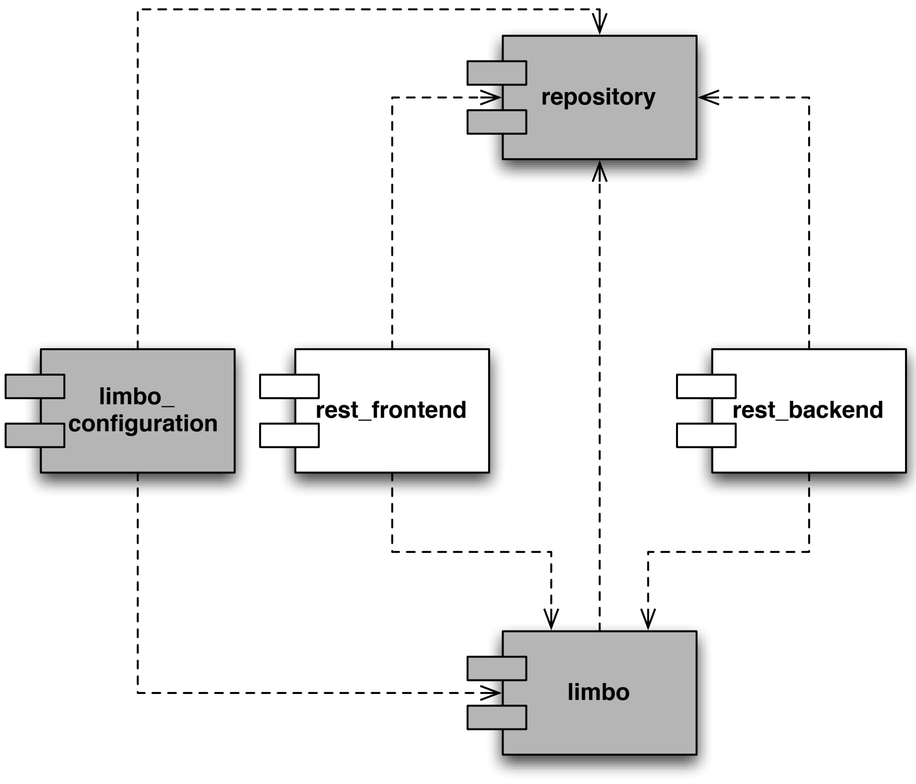


Figure 3.3: Component diagram of REST deployment

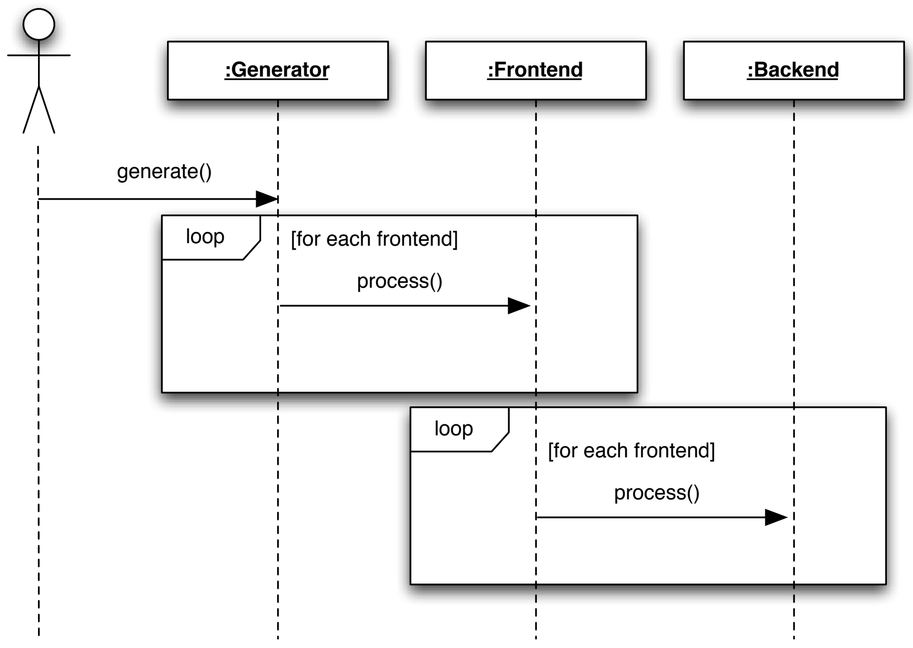


Figure 3.4: Sequence diagram of Limbo control flow

4 Development environment support

From the WP4 lessons learned, we knew from developer's feedback that the process of using Limbo-generated code to create their own web services was time consuming because Limbo only generated the needed classes, not the project structure needed to utilize them. The developers would have to create projects, source packages, add the needed libraries, etc. This process would take time for the developers before they could have their web service up and running.

In order to save some time, we decided to add support in Limbo for project resources generation. This includes generation of normal JSE projects, plug-in projects and MIDlet projects (JME version).

This feature was added in Limbo plug-in, since we will always want the generation of project resources. The activity diagram in Figure 4.1 shows us the project generation feature behavior.

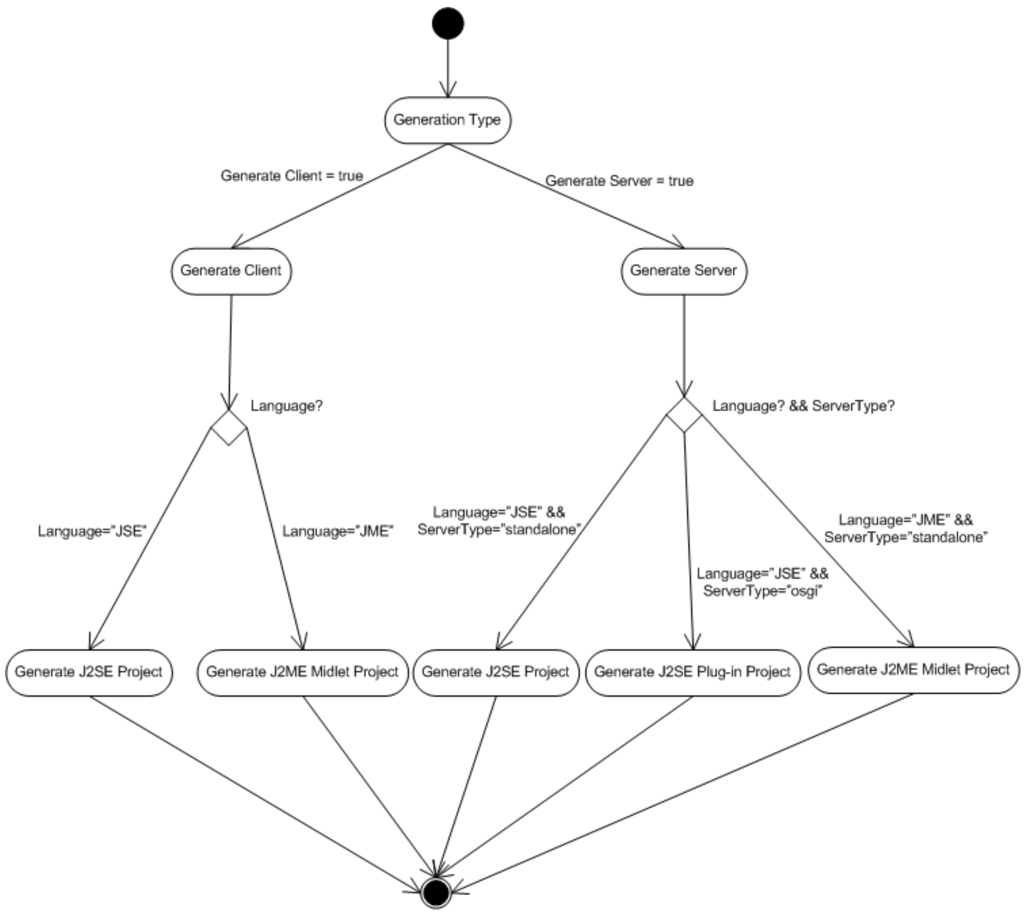

Generation Type
Generate Client = true
Generate Server = true
Generate Client
Generate Server
Language?
Language? && ServerType?
Language="JSE"
Language="JME"
Language="JSE" && ServerType="standalone"
Language="JME" && ServerType="standalone"
Language="JSE" && ServerType="osgi"
Generate J2SE Project
Generate J2ME Midlet Project
Generate J2SE Project
Generate J2SE Plug-in Project
Generate J2ME Midlet Project

Figure 4.1: Activity diagram for project generation feature

For client sides, if the language is J2SE and for server sides if the language is JSE and the server type is "standalone" we generate the following resources:

- .classpath file: Stores the Java Build Path, i.e., the classpath used at compile-time and the classpath at runtime.
- .project file: Specifies general project properties.

As for JME projects the following resources are required for both server and client projects:

- .classpath file: Stores the Java Build Path, i.e., the classpath used at compile-time and the classpath at runtime.
- .project file: Specifies general project properties.
- .mtj file: Specifies the MIDlet project meta data, this information includes jad file name and emulated device information.
- .settings/org.eclipse.core.jdt.prefs file: This file specifies which JVM to use at compile time.

For server sides, in case of JSE language and server type OSGi, we need to generate resources for a plug-in project, these resources are:

- .classpath file: Stores the Java Build Path, i.e., the classpath used at compile-time and the classpath at runtime.
- .project file: Specifies general project properties.
- MANIFEST.MF file: Specifies plug-in information, this information includes dependencies to other plug-ins, plug-in classpath, plug-in activator, imported and exported packages.
- build.properties file: Specifies project build properties.

In addition to these resources information generation, Limbo adds all the needed libraries for project compilation and runtime. With this feature the developer will have in general to import existing projects into the workspace browsing the project directory and the project is added to the workspace.

5 Hydra-enabling a device using Limbo

In this chapter we will show how Limbo can automatically hydra-enable a device. There are two general requirements for a device to be hydra-enabled, they are:

- Device discoverable by UPnP, Bluetooth, etc. protocol, providing a set hydra-standard services as listed below, these services are not web services yet.

  The Hydra DAC searches for devices in the network discoverable by UPnP protocol among other discovery protocols. In order to be hydra-enabled the device has to provide an hydra-standard service. These services contain general operations for a device, as follows:

  – String CreateWS(): Creates the web service for the device, returning the endpoint of the created web service.
  – String GetDACEndpoint(): Returns the endpoint of the Hydra DAC.
  – String GetDiscoveryInfo(): Returns general discovery information of the device.
  – String GetErrorMessage(): Returns an error message.
  – boolean GetHasError(): Returns true if the device is in an error state false otherwise.
  – String GetHydraID(): Returns the HydraID of the device.
  – String GetHydraWSEnpoint(): Returns the endpoint of the Hydra Standard web service.
  – String GetProperty(String Property): Returns the value of the given property.
  – String GetStatus(): Returns the status of the device, represented by a String value.
  – String GetWSDL(): Returns the content of the device WSDL file as String value.
  – String GetWSEndpoint(): Returns the device web service endpoint.
  – void SetDACEndpoint(String DACEndpoint): Sets the DACEndpoint to the given value.
  – void SetHydraID(String HydraID): Sets the device HydraID to the given value.
  – void SetProperty(String Property, String Value): Sets the given property to the given value.
  – void SetStatus(String Status): Sets the device status to the given value.
  – void Stop(): Stops the device.
  – void StopHydraWS(): Stops the Hydra Standard web service.
  – void StopWS(): Stops the device web service.

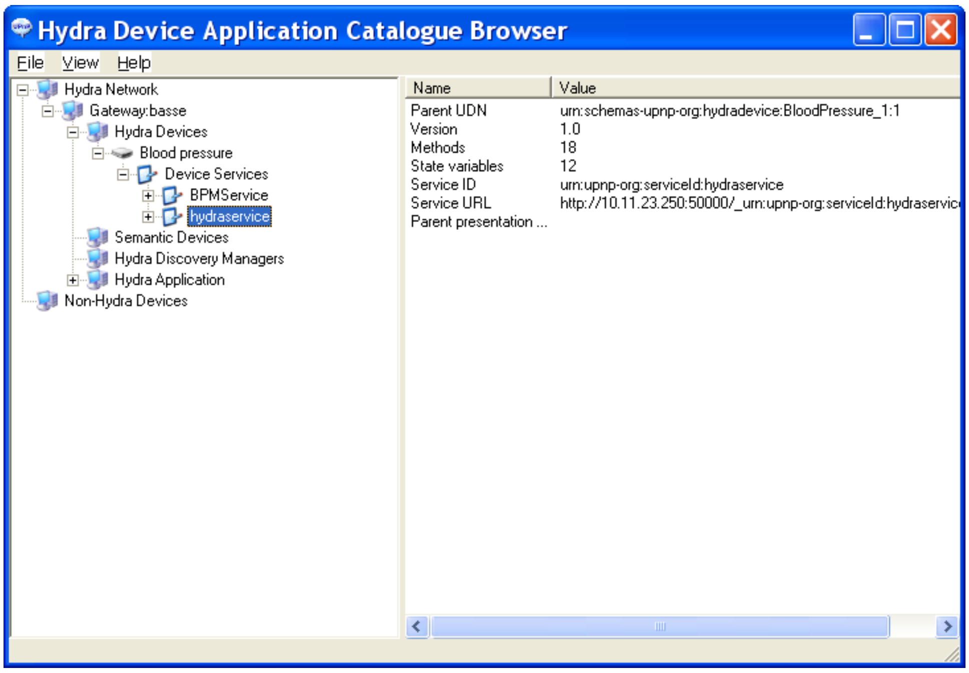
- Provide Hydra Standard web services for the above services set. The device has also to offer an Hydra Standard web service that provides a subset of this operations (currently not including Stop(), StopHydraWS(), StopWS() and CreateWS()).

In order to be able to support this functionalities in Limbo generated code we added a new parameter to the Limbo configuration specifying if we want to generate an hydra-standard service, as shown in Table 6.1 and designed a new Backend plug-in, the HydraStandard plug-in.

This Backend plug-in when called, checks the hydra standard parameter from the Limbo configuration and if it is set to "true", it calls the method generateCode of Limbo, giving as a parameter a WSDL file that implements all the operations needed by the hydra-standard service.

These operations need also to be available by UPnP protocol, in order to achieve this we extended the UPnP backend plug-in, by checking also this hydra standard parameter from the Limbo configuration, if set to "true" the UPnP backend generates also all the necessary resources to make this hydra standard service available via UPnP protocol.

At runtime the HydraDAC will find the device by UPnP protocol, and since it provides an hydra-standard service it resolves the device as a hydra device. As shown in Figure 5.1.



Name	Value
Parent UDN	urn:schemas-upnp-org:hydradevice:BloodPressure_1:1
Version	1.0
Methods	18
State variables	12
Service ID	urn:upnp-org:serviceId:hydraservice
Service URL	http://10.11.23.250:50000/_urn:upnp-org:serviceId:hydraservic
Parent presentation ...	

Figure 5.1: New Hydra Device found by the HydraDAC.

6 A general overview of Limbo features, Limbo generation process and its generated artifacts

This chapter will give a general high level overview of the updated Limbo features and its compilation process compared to the work reported in deliverables D4.2 (Hansen et al., 2007) and D4.2b (Hansen et al., 2008c). Also an overview of how to use these new features and the generated artifacts will be discussed.

6.1 Updated Limbo features and compilation process

Taking consideration of the work talked in the former sections, the updated Limbo feature and compilation process is shown in Figure 6.1 and Figure 6.2, in which the consideration of different SOAP transportation protocols are considered as variants in the feature diagram and also reflect in the compilation process.

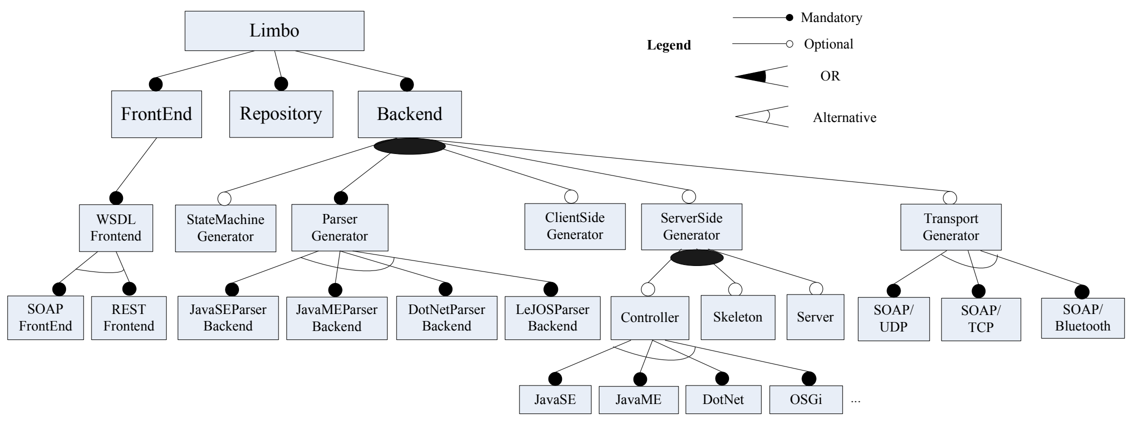


Figure 6.1: Limbo feature diagram

6.2 How to make use of the new features

Now we will briefly discuss on how to use Limbo with these new features. Table 6.1 shows us the configuration parameters given by the developer to Limbo for code generation.

We will explain now, in general steps, how to use Limbo using Eclipse IDE. In order to run Limbo you will need the project limbo_osgi that can be retrieved from svn: https://hydra.fit.fraunhofer.de/svn/trunk/sdk/limbo_osgi. Limbo is a set of plug-ins, e.g., limbo.jar, repository.jar, upnp.jar, statemachine.jar. This architecture allows the developer easily to add/remove features from the generated code. These bundles can be found in the lib folder of the limbo_osgi project and can be added/removed to/from that folder if the developer does/does not want to generate extra feature code.

First of all the developer needs to specify a WSDL file for the web service he wants Limbo to generate code to. After writing the WSDL file the developer will have to run the OSGiLimboManager class by right-clicking in the class and choosing Run As...->Open Run

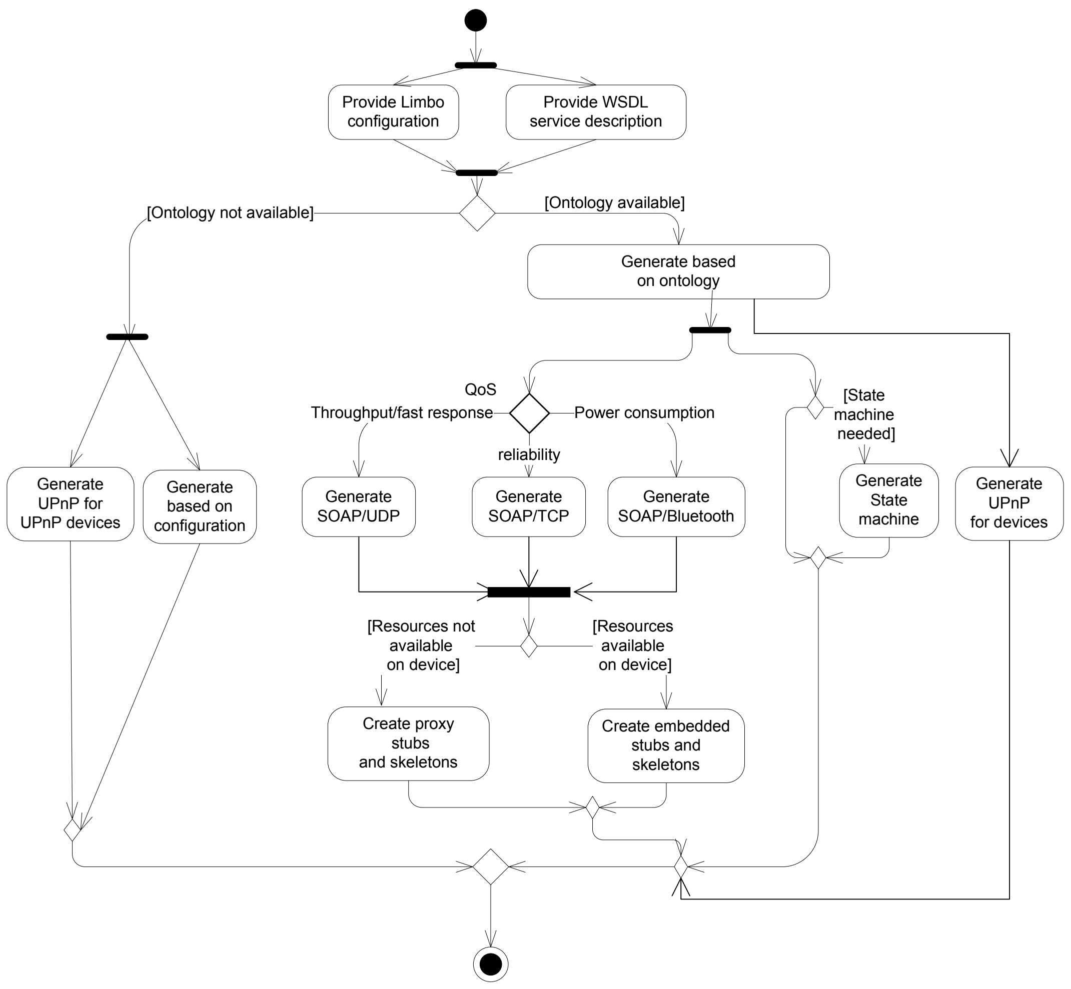

Provide Limbo configuration
Provide WSDL service description
[Ontology not available]
[Ontology available]
Generate based on ontology
QoS
Throughput/fast response
Power consumption
reliability
[State machine needed]
Generate UPnP for UPnP devices
Generate based on configuration
Generate SOAP/UDP
Generate SOAP/TCP
Generate SOAP/Bluetooth
Generate State machine
Generate UPnP for devices
[Resources not available on device]
[Resources available on device]
Create proxy stubs and skeletons
Create embedded stubs and skeletons

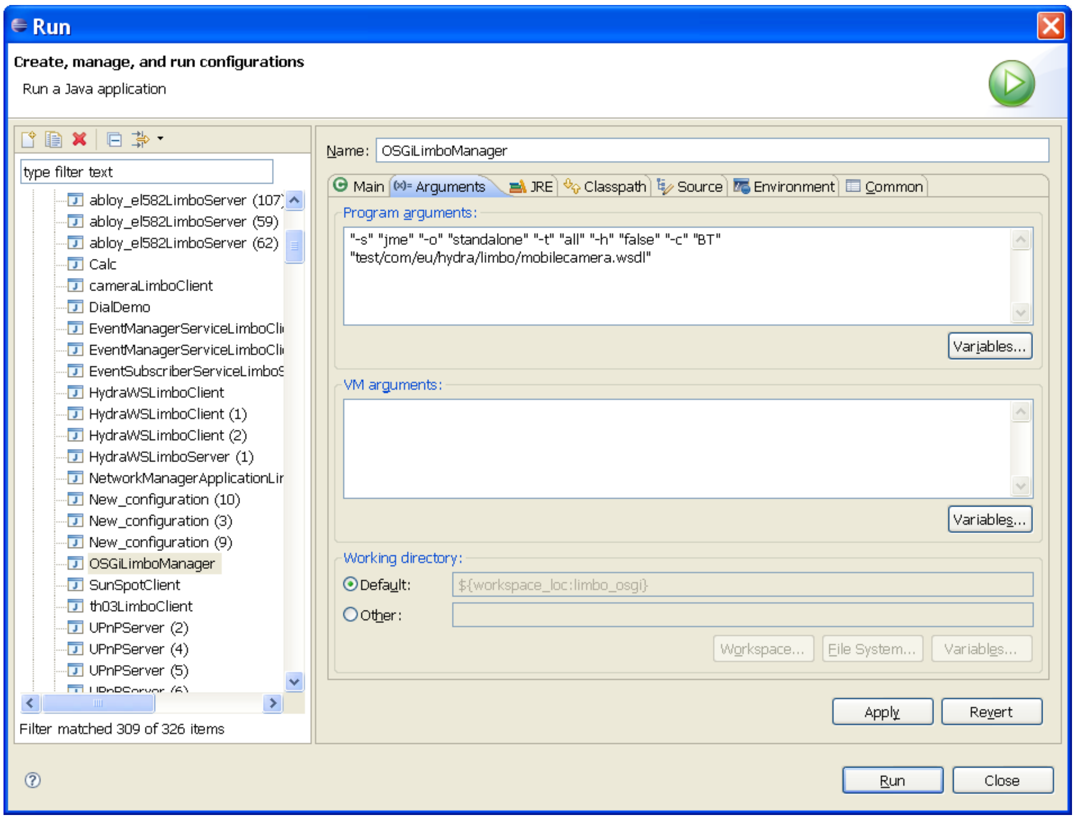
Figure 6.2: Limbo compilation process

Dialog. In the Run Dialog choose a new Java Application configuration and add in the program arguments tab the arguments described in Table 6.1. One example of it is shown in Figure 6.3 and select Run. A new instance of the OSGi framework will be launched and Limbo will generate code for the specified web service. After the code generation the developer can check that a new folder named “generated” was created in the limbo_osgi project by right-clicking in the project and selecting Refresh. One or two folders are inside this generated directory, e.g., xxServer for server code and xxClient for client code.

As next step the developer needs to import the generated projects into the workspace to he/she can start working. To do this go to menu File->Import..., select Existing Projects into Workspace and browse the generated folder of the limbo_osgi project as shown in Figure 6.4 this will import the generated projects into the workspace to the developer can start working on it.

Parameter	Explanation	Possible Values
-s	Language variant in which the generated code will be written.	"jse" or "jme"
-o	Target system. Limbo can generate standalone servers or osgi based servers.	"standalone" or "osgi"
-t	Generated components.	"client", "service" or "all"
-h	Specifies if we want to generate as hydra standard web service.	"true" or "false"
-c	The protocol used for communication between server and client.	"TCP", "UDP" or "BT"
wsdl	Path for the wsdl description of the web service.	p.e. "test/com/eu/hydra/limbo/mobilecamera.wsdl"

Table 6.1: Limbo configuration parameters



Name: OSGiLimboManager

Program arguments:

```
"-s" "jme" "-o" "standalone" "-t" "all" "-h" "false" "-c" "BT"
"test/com/eu/hydra/limbo/mobilecamera.wsdl"
```

VM arguments:

Working directory:
[✓ Default:] ${workspace_loc:limbo_osgi}
Other:

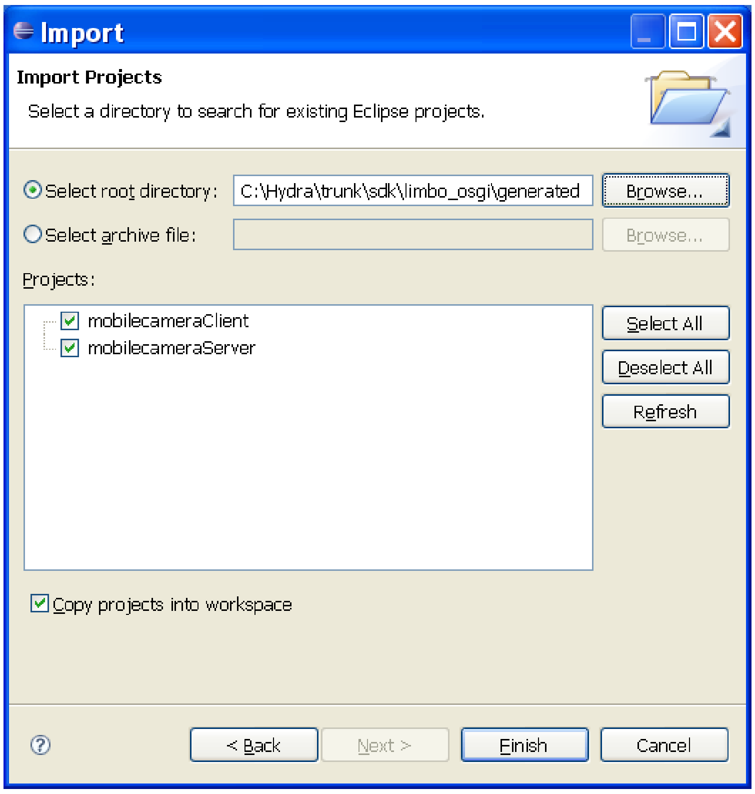
Figure 6.3: Limbo configuration arguments example.

6.3 Overview of the generated code

Table 6.2 summarizes the generated classes for the currently supported platforms. As we can see from the table, the client classes are the same, and differences are the server side for different platform.

We will briefly describe each of the generated classes by Limbo:

- JSE standalone/OSGi and JME client
  - Package com.eu.hydra.limbo.client:



Select root directory: C:\Hydra\trunk\sdk\limbo_osgi\generated

- [✓] mobilecameraClient
- [✓] mobilecameraServer

[✓] Copy projects into workspace

Figure 6.4: Import camera projects into workspace.

- ∗ StringTokenizer.java: Utility class.
- ∗ LimboClient.java: Defines the main method of the client.
- ∗ LimboClientParser.java: Class used for parsing of SOAP messages.
- ∗ LimboClientPort.java: Interface defining the operations the client can call.
- ∗ LimboClientPortImpl.java: Implementation of the operations call.
- ∗ ClientProtocol.java: Interface defining operations of a communication protocol.
- ∗ TCPProtocol.java: Defines the communication between client and server using TCP protocol.
- ∗ UDPProtocol.java: Defines the communication between client and server using UDP protocol.
- ∗ BTProtocol.java: Defines the communication between client and server using bluetooth.

- JSE standalone/OSGi and JME server
  - Package com.eu.hydra.limbo.handler:
    - ∗ Handler.java: Abstract class for handler.
    - ∗ Handlers.java: Class managing the queue of handlers of the service.
    - ∗ HandlerService.java: Interface defining the handler method.

	JSE OSGi	JSE Standalone	JME
client	ClientHeaderParser.java	ClientHeaderParser.java	ClientHeaderParser.java
	StringTokenizer.java	StringTokenizer.java	StringTokenizer.java
	LimboClient.java	LimboClient.java	LimboClient.java
	LimboClientParser.java	LimboClientParser.java	LimboClientParser.java
	LimboClientPort.java	LimboClientPort.java	LimboClientPort.java
	LimboClientPortImpl.java	LimboClientPortImpl.java	LimboClientPortImpl.java
	ClientProtocol.java	ClientProtocol.java	ClientProtocol.java
	TCPProtocol.java	TCPProtocol.java	TCPProtocol.java
	UDPProtocol.java	UDPProtocol.java	UDPProtocol.java
	BTProtocol.java	BTProtocol.java	BTProtocol.java
server	Handler.java	Handler.java	Handler.java
	Handlers.java	Handlers.java	Handlers.java
	HandlerService.java	HandlerService.java	HandlerService.java
	LogHandler.java	LogHandler.java	LogHandler.java
	SOAPHandler.java	SOAPHandler.java	SOAPHandler.java
	StringTokenizer.java	HeaderParser.java	HeaderParser.java
	Activator.java	StringTokenizer.java	StringTokenizer.java
	OpsImpl.java	EndPoint.java	EndPoint.java
	Parser.java	LimboServer.java	LimboServer.java
	Servlet.java	OpsImpl.java	OpsImpl.java
	ServerProtocol.java	Parser.java	Parser.java
	TCPProtocol.java	Service.java	Service.java
	UDPProtocol.java	ServerProtocol.java	ServerProtocol.java
	BTProtocol.java	TCPProtocol.java	TCPProtocol.java
		UDPProtocol.java	UDPProtocol.java
		BTProtocol.java	BTProtocol.java

Table 6.2: Limbo generated classes.

  * LogHandler.java: Handler used for logging features.
  * SOAPHandler.java: Handler responsible for handling SOAP messages.

- Package com.eu.hydra.limbo:
  * StringTokenizer.java: Utility class.
  * Parser.java: Class used for parsing SOAP messages.
  * OpsImpl.java: Class defining the implementation of the operations provided by the web service.
  * ServerProtocol.java: Interface defining operations of a communication protocol.
  * TCPProtocol.java: Defines the interaction of the server with the client using TCP protocol.
  * UDPProtocol.java: Defines the interaction of the server with the client using UDP protocol.
  * BTProtocol.java: Defines the interaction of the server with the client using Bluetooth protocol.

• JSE standalone and JME server
  - Package com.eu.hydra.limbo:
    * HeaderParser.java: Parser for HTTP messages.
    * EndPoint.java: Abstract class that defines the endpoints (i.e. services) that are provided by the server.
    * LimboServer.java: Limbo server main class.
    * Service.java: Extends Endpoint class, defines a service provided by the server.

- JSE OSGi server
  - Package com.eu.hydra.limbo
    - Activator.java: Activator class for the OSGi Server.
    - Servlet.java: The server class in this case a Servlet.

Based on the generated code, the device developer only needs to bind the device services to the actual device by adding related code in class OpsImpl.java or LimboClientPortImpl.java.

7 Developers evaluation

In this chapter we will present developers evaluation of Limbo. The developers involved were partners from TID and T-CON to whom it was asked to answer a small questionnaire on their experience with Limbo.

7.1 TID experience

We have used Limbo to generate code for several devices managers with the following results:

- Weight scale (UC-321PBT A&D) Bluetooth device proxy code

  The proxy WS code and the UPnP device were generated using Limbo compiler. The generation of Limbo code was easy, but we faced a problem because the WS code generated by Limbo was not compatible with a PHP client, so several corrections had to be performed in the SOAP message parsing in order to successfully invoke the services from a PHP client. As there were only two methods for the service, the corrections applied to the code did not take more than one hour. However, we would prefer that Limbo would deal with this compatibility issues in the future. The resulting code is quite efficient regarding resource consumption (processor, memory) compared to Axis generated code (more than half of memory usage).

- Blood pressure (UC-322PBT A&D) Bluetooth device proxy code

  The proxy WS code and the UPnP device were generated using Limbo compiler. The generation of Limbo code was easy, but we faced a problem because the WS code generated by Limbo was not compatible with a PHP client, so several corrections had to be performed in the SOAP message parsing in order to successfully invoke the services from a PHP client. As there were only two methods for the service, the corrections applied to the code did not take more than one hour. However, we would prefer that Limbo would deal with this compatibility issues in the future. The resulting code is quite efficient regarding resource consumption (processor, memory) compared to Axis generated code (more than half of memory usage).

- Network Manager J2SE

  The code (client and server side) for the Network Manager was generated using Limbo compiler. The code generation was easy, but we faced the same compatibility issues as before, with Axis client code and .Net client code. Several corrections had to be performed in the SOAP message parsing in order to successfully invoke the Network Manager Service from .Net and Axis generated code. The corrections applied to the code took three hours because there were more than 15 methods to modify. The resulting code is quite efficient regarding resource consumption (processor, memory) compared to Axis generated code (more than half of memory usage).

- Network Manager J2ME

  The server code for Network Manager for J2ME version was generated using Limbo compiler. The code generation was easy, but we faced several problems with the generated code:

  - J2ME virtual machine (Esmertec Jbed) used was not able to open server socket connections, so another virtual machine had to be installed (IBM MIDP Java Emulator 2.3)
  - The IBM Jbed does not support the FileConnection because when the LogHandler is created, an exception is thrown: Scheme not found file The problem was solved the problem by removing this handler in the NMApplicationHandlers constructor.
  - The NMAppParser class experiments the usual parsing problems: it is not generic enough. In order to be able to invoke the service with axis generated client code, it was modified in order to get the right result. The corrections applied to the code due to the problems took eight hours. The generated code seems to behave perfectly on J2ME. It cannot be compared with Axis, because there is not a J2ME version of it.

7.2 TCON experience

Xbee Series 2 (ZB OEM) The usage of Limbo is very easy, we just need to specify a set of arguments (e.g. server, client, language, wsdl, etc) and have our web service generated. Some time was spent in order to be able to specify the wsdl for our web service. With Axis the creation of a web service from a Java interface is fast and creates automatically my wsdl file, but we do not have project generation and support for making the device available via UPnP protocol, which Limbo is able to do. We never tested resource consumption when comparing it with an Axis generated service.

7.3 Target platform overview

As shown in Table 1.1 we have been using Limbo tool to generate web services code for a number of devices, and also explored by using it for the development of other Hydra components.

For D1 devices we have been deploying mostly, J2ME code in several Nokia mobile phones. One example is the Nokia 6630 where we have been deploying several services, this phone has 10MB shared memory. One of the services we deployed in this phone was a camera service, which when called takes a picture using its camera and sends the image data to the client. We deployed client code in a Nokia 3110, with 9MB shared memory, and invoked the camera service, when receiving the data image from the server the client builds the image and shows the picture taken from the server. Other service example is a SMS service deployed in a Nokia N80 phone, this service when invoked sends a SMS message to the given number by the client. Performance and resource measurements were made with this service (reference D4.2 (Hansen et al., 2007), Limbo evaluation).

As for D0 devices developers have been generating proxy services for different devices, in Building Automation and Health care scenarios, many of these services were presented in previous reviews.

Currently TID has been working on a J2SE version and J2ME version of the Network Manager and the obtained results in terms of performance and memory consumption show that Limbo is very efficient when comparing it with AXIS generated code.

8 Limbo configuration validation and QoS support for web service generation using OWL/SWRL ontologies

OWL/SWRL ontologies are used in Limbo to formulate the configuration of Limbo, i.e, feature combinations as discussed in deliverables D4.2 (Hansen et al., 2007) and D4.2b (Hansen et al., 2008c). We will improve these former work based on our new findings and will discuss it in Section 8.1. Also to take into consideration of quality of service requirements and different communication technologies employed by embedded devices, we will extend the former work by guiding the limbo generation process using SWRL rules based on the ontologies (QoS ontologies, Device ontologies) as discussed in deliverable D4.8 (Ingstrup and Zhang, 2008).

8.1 Limbo configuration validation

In the former deliverable D4.2b (Hansen et al., 2008c), we have pointed out the problems with the OSGi R4 definition of declarative services, which is using key-value pair to specify component properties:

- *Global constraints are not supported.*
- *Contextual constraints are not supported.*
- *Functional constraints are not supported.*

We also find a problem of OSGi DS implementation as in Eclipse Equinox[1]: component interface mismatches are not sufficiently handled. For example, when there are multiple components with the same name, Equinox just picks the one with lowest bundle (a synonym for *component* in OSGi) identification to execute, and does not care about the exact semantics of the components and their relationships.

Therefore we proposed to enhance the semantics of OSGi declarative services, by adding semantics to the component using OWL[2], in which we developed the OSGi component ontology and a number of configuration rules are developed (Hansen et al., 2008c). The processing of these rules are handled using the developed configuration bundles discussed in Section 8.4. To identify possible interface mismatches, we need also associate the component implementation details with each other in a specific configuration, to validate whether components are correctly referenced with each other. We need also make sure that the Repository architecture style is stilled followed when Limbo components are reconfigured. In this context, and potentials for validating Hydra architecture, and architecture of applications developed using Hydra middleware, a number of OWL ontologies for software architecture assets (including semantic architecture style, semantic OSGi components, semantic connectors) are developed, as part of self-management ontologies, as detailed in deliverable D4.8 (Ingstrup and Zhang, 2008). In this deliverable, we call these semantic software architecture assets SACoCo (Semantic architecture, component, and connectors) ontologies.

[1] http://www.eclipse.org/equinox/
[2] OWL Web Ontology Language http://www.w3.org/TR/owl-features/

SACoCo has a number of ontologies, including one for atomic connectors (AtomicConnetor ontology) and another one for composite connectors (CompositeConnector ontology), an ontology for high level component concepts (ServiceComponent ontology) which imports separate ontologies for specific component models (e.g. OSGi), an ontology for architectural styles (ArchStyle ontology), and additionally we have another ontology to specify architecture constraint rules using SWRL (ArchRule ontology). The relationships between these ontologies in SACoCo are shown in Figure 8.1. For the Limbo case, we only need the OSGi component ontology among all the semantic component models. SACoCo (Semantic architecture, component, and connectors) ontologies can be applied to any other situations for software architecture and configuration validation, other than Limbo.

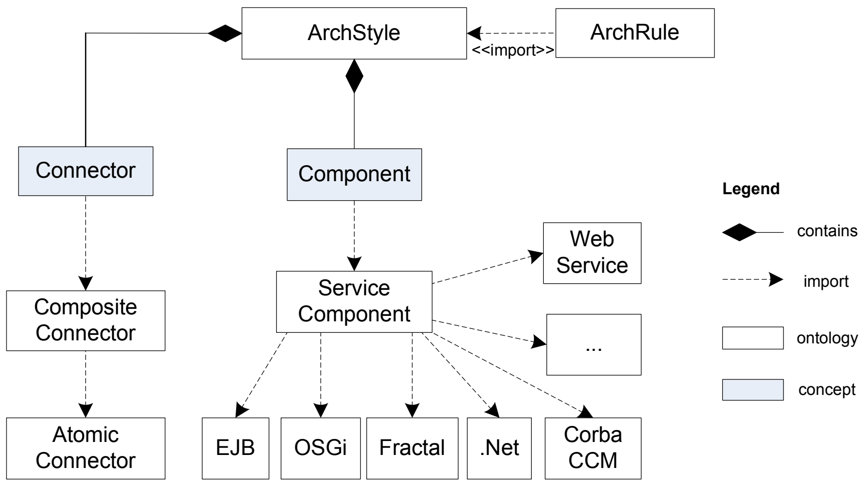


Figure 8.1: SACoCo ontologies structure

Besides the rules as detailed in D4.2b (Hansen et al., 2008c), we need to check that Limbo components (implemented as OSGi components) are correctly referenced, as discussed in the following section. This is an update of the component reference rule as in D4.2b (Hansen et al., 2008c), which did not check the details of component interfaces.

8.1.1 Limbo configuration and validation rules

As introduced in (Hansen et al., 2008a), Limbo components are implemented as OSGi bundles. Component *Limbo* provides a *Generator* service that *Backends* use, and will produce web service stubs and skeletons. At runtime Limbo selects and uses a set of Backends based on its configuration. The Limbo component requires the presence of at least one *Backend* and exactly one *Repository*. SWRL is used to specify architecture constraints for Limbo based on the SACoCo models.

SWRL is a Horn clause rules extension to OWL-DL and shares its formal semantics. A SWRL rule is composed of an antecedent part (body), and a consequent part (head). Both the body and head consist of positive conjunctions of atoms. A SWRL rule means that if all the atoms in the antecedent (body) are true, then the consequent (head) must also be true. In our practice, all variables in SWRL rules bind only to known individuals in an ontology in order to develop DL-Safe rules to make them decidable. In a SWRL rule, the symbol "$\wedge$" means conjunction, and "$?x$" stands for a variable, "$\rightarrow$" means implication, and if there is no

"?" in the variable, then it is an instance. Now we will show the configuration and validation rules, which are the basis for self-configuration rules in D4.8 (Ingstrup and Zhang, 2008).

Check reference relationships

For the OSGi component model and Repository style, the rule "check_OSGi_Reference_noDetails" retrieves all Repository components in the current configuration. If a component has a reference which has cardinality of the form "1." (at least one reference to other service), then there must be a component providing that required service.

Rule: check_OSGi_Reference_noDetails

$archstyle : CurrentConfiguration(?con) \wedge$
$archstyle : hasArchitecturePart(?con, ?comp1) \wedge$
$osgi : componentName(?comp1, ?compname1) \wedge$
$osgi : reference(?comp1, ?ref1) \wedge$
$osgi : cardinality(?ref1, ?car1) \wedge$
$swrlb : containsIgnoreCase(?car1, \text{“1.”}) \wedge$
$osgi : interface(?ref1, ?inter1) \wedge$
$osgi : interfaceName(?inter1, ?name1) \wedge$
$archstyle : hasArchitecturePart(?con, ?comp2) \wedge$
$architectureRole(?comp2, ?role2) \wedge$
$archstyle : archPartName(?role2, ?rolename) \wedge$
$swrlb : equal(?rolename, \text{“Repository”}) \wedge$
$osgi : service(?comp2, ?ser2) \wedge$
$osgi : provide(?ser2, ?inter2) \wedge$
$osgi : interfaceName(?inter2, ?name2) \wedge$
$osgi : componentName(?comp2, ?compname2) \wedge$
$swrlb : equal(?name1, ?name2)$
$\rightarrow sqwrl : selectDistinct(?comp1, ?comp2)$

Identifying valid component references

Assume that we have two Repository components loaded by DS and the two components are implementing two Repository interfaces that differ only with the last operation:

```
URI getHydraOntologyExtension (File wsdlFile);
URI getHydraOntologyExtension (String wsdlFile);
```

Limbo needs to be bound to the Repository interface that has the signature as the first method. The Eclipse Equinox DS bundle binds Limbo to the first Repository component that has the lowest bundle id, without respecting the interface signature it has. If it is bound to the correct Repository, then Limbo can run successfully. But if the Repository with the lowest bundle id is the one that provides the same method but with a String parameter, Limbo will not work.

OWL-S [3] ontologies are simplified to model component behaviors. Here in order to implement SWRL rules (the reason being that current Protege SWRL APIs can not parse rdf:XMLLiteral), we changed the range of the data type property *parameterValue* to *xsd:string* defined in the OWL-S Process ontology. The Limbo Repository component has five operations which are defined as five atomic processes in the Process ontology, where the return types are modeled as process *Output*s, and method signatures are modeled as *Input*s. An instance of *SimpleProcess* is defined which is composed with these five *AtomicProcess*es correspondingly. In these Repository Client components, instances of *SimpleProcess* for each operation are defined in a similar way. If there is a reference from the Repository Client to the Repository component in a method, the atomic process for this method will have a *Participant* instance which should be the *Repository* component.

[3] http://www.w3.org/Submission/OWL-S/

The inputs of this atomic process will contain both the signature of the referenced operation together with its return type, and method signature of itself.

To correctly justify a reference, the component package and component name, method name, method signature including data type and order, and return type, should be consistent in both referencing and referenced components. Using the service details as provided by the ServiceProfile and Process ontologies, we can then retrieve the details of the services and its method signatures. This rule is named “check_OSGi_Reference_Details” as follows.

Rule: check_OSGi_Reference_Details

$$\begin{array}{l}
\mathit{BODY\_OF\_RULE\_check\_OSGi\_Reference\_noDetails} \wedge \\
\mathit{component} : \mathit{componentServiceDetails}(?\mathit{comp}1, ?\mathit{pr}1) \wedge \\
\mathit{service} : \mathit{presents}(?\mathit{pr}1, ?\mathit{prservice}1) \wedge \\
\mathit{profile} : \mathit{has\_process}(?\mathit{prservice}1, ?\mathit{process}1) \wedge \\
\mathit{process} : \mathit{realizedBy}(?\mathit{process}1, ?\mathit{aprocess}1) \wedge \\
\mathit{process} : \mathit{hasInput}(?\mathit{aprocess}1, ?\mathit{input}1) \wedge \\
\mathit{process} : \mathit{parameterValue}(?\mathit{input}1, ?\mathit{ivalue}1) \wedge \\
\mathit{component} : \mathit{componentServiceDetails}(?\mathit{comp}2, ?\mathit{pr}2) \wedge \\
\mathit{service} : \mathit{presents}(?\mathit{pr}2, ?\mathit{prservice}2) \wedge \\
\mathit{profile} : \mathit{has\_process}(?\mathit{prservice}2, ?\mathit{process}2) \wedge \\
\mathit{process} : \mathit{realizedBy}(?\mathit{process}2, ?\mathit{aprocess}2) \wedge \\
\mathit{process} : \mathit{hasInput}(?\mathit{aprocess}2, ?\mathit{input}2) \wedge \\
\mathit{process} : \mathit{name}(?\mathit{aprocess}2, ?\mathit{proname}2) \wedge \\
\mathit{process} : \mathit{hasOutput}(?\mathit{aprocess}2, ?\mathit{proout}2) \wedge \\
\mathit{process} : \mathit{parameterValue}(?\mathit{input}2, ?\mathit{ivalue}2) \wedge \\
\mathit{process} : \mathit{parameterValue}(?\mathit{proout}2, ?\mathit{ovalue}2) \wedge \\
\mathit{swrlb} : \mathit{stringConcat}(?\mathit{str}1, ?\mathit{compname}2, \text{“ + ”}) \wedge \\
\mathit{swrlb} : \mathit{stringConcat}(?\mathit{str}2, ?\mathit{str}1, ?\mathit{proname}2) \wedge \\
\mathit{swrlb} : \mathit{stringConcat}(?\mathit{str}3, ?\mathit{str}2, \text{“\#”}) \wedge \\
\mathit{swrlb} : \mathit{stringConcat}(?\mathit{str}4, ?\mathit{str}3, ?\mathit{ivalue}2) \wedge \\
\mathit{swrlb} : \mathit{stringConcat}(?\mathit{str}5, ?\mathit{str}4, \text{“\$”}) \wedge \\
\mathit{swrlb} : \mathit{stringConcat}(?\mathit{str}6, ?\mathit{str}5, ?\mathit{ovalue}2) \wedge \\
\mathit{swrlb} : \mathit{equal}(?\mathit{ivalue}1, ?\mathit{str}6) \\
\rightarrow \mathit{sqwrl} : \mathit{selectDistinct}(?\mathit{ivalue}1, ?\mathit{comp}1, ?\mathit{comp}2, ?\mathit{str}6) \wedge \mathit{sqwrl} : \mathit{select}(\text{“}\textit{valid references}\text{”})
\end{array}$$

As can be noted from this rule, in a referencing component, the references to another component is modeled in the *hasInput* datatype property in *Process* ontology, in the format as “*component name(including package name)+operation name#input types with orders$return type*”. Then this information is compared with that from the referenced component with respect to a specific interface, which is modeled as an atomic process. If they are exactly matched, then the references are valid.

8.2 Power consumption and QoS considerations of communication technologies for web service generation

We have tested that SOAP over UDP has better throughput over wireless network (e.g. Wifi network), and over the mix between wireless and wired network as shown in D5.9 (Sperandio et al., 2008). Here we summarize our tests in the following tables (Table 8.1, Table 8.2), with additions on the average power consumption for Hama Bluetooth dongle.

Table 8.1: Power consumption for Nokia N95 and Hama Bluetooth dongle

	BTClient	BTServer	Idle
Nokia N95	0.488w	0.2395w	
Hama BT Dongle	0.092w	0.051w	0.037w

Table 8.2: Power consumption for D-link DWL-G122 WIFI Dongle

D-link DWL-G122 WIFI Dongle			
	TCP	client	1.358w
		server	1.356w
	UDP	client	1.36w
		server	1.359w
	idle		1.344w

From Table 8.1 and Table 8.2, we could notice that SOAP over Bluetooth consume much less power than SOAP over UDP or SOAP over TCP. This knowledge will be used to guide the code generation of Limbo when power consumption should be considered in web services.

Table 8.3: Performance tests of TCP/UDP transportation

		Wired Client		Wireless Client
		Wired server	wireless server	wireless server
TCP	RTC	3.5ms	21.5ms	25.99ms
	Throughput	2311.23	401.25	340.73
UDP	RTC	3.44ms	8.16ms	7.5ms
	Throughput	2311.616	954.44	1009.86
	goodput	100%	99.93%	99.90%
BT	RTC			669.62ms
	Throughput			18.01

For Table 8.3, we can conclude that:

- If wireless network is involved in a web service call, SOAP over UDP has better throughput than SOAP over TCP, and of course than SOAP over bluetooth.
- SOAP over UDP has good goodput, which can be used for normal web service calls without high reliability requirements.
- For wired network, SOAP over UDP and SOAP over TCP has equal performance.
- For fast response from a service, i.e. a short RTC (round trip call) time, we need to use the SOAP over UDP.

This knowledge will be considered for guiding the code generation of Limbo, if QoS is required for the generated code:

- SOAP over UDP should be generated if a wireless network is involved in a web service call for good throughput, if reliability is not critical.
- SOAP over TCP should be generated for highest reliability.
- SOAP over Bluetooth should be generated for best power consumption requirement.

- SOAP over UDP should be used in any case (if there is an IP stack and a device has the capability to make use of the TCP/IP connection) for fast response from a web service.

We will use SWRL to implement these conclusions and guide the web service generation process for Limbo by processing these rules. What is more interesting is the usage of this knowledge at runtime to realize the self-management activities according to certain QoS requirements, where a number of services are involved. These requirements are possibly conflict with each other, for example both power consumption and response time should be within certain limits. How to find an optimal solution is a global multi-objective optimization problem, and can be resolved using genetic algorithms as outlined in deliverable D4.8, and remains ongoing work.

As can be noted from above, the generation type of web service for a device depends on:

- Device capabilities: Network capabilities of the device, software capabilities, hardware capabilities (e.g. CPU). The device capabilities are described in Device ontologies (Hardware ontology, Software platform ontology) in deliverable D4.2 (Hansen et al., 2007).
- QoS requirements, including the service response time, throughput, and power consumption requirements, as modeled in QoS ontologies discussed in deliverable D4.8 (Ingstrup and Zhang, 2008).

Therefore now we need to upgrade our generation rules compared to the former deliverables (Hansen et al., 2007), (Hansen et al., 2008c), due to the considerations of QoS parameters for code generation, in which communication protocols are considered.

A first draft of the QoS ontology is briefly discussed in D4.5 (Scholten and Shi, 2008), and an enhanced version is developed in D4.8 (Ingstrup and Zhang, 2008) in which the necessary QoS parameters (These parameters can cover what we needed for guiding the code generation process for Limbo, as analyzed above) that are considered, including:

- Bandwidth, or throughput
- Latency
- ErrorRate
- Availability, including both network availability, and service availability
- Reliability
- Security
- Accuracy (of measurement, operation)
- Speed (of operation, service)
- PowerDrain (of service execution, of operation)
- Cost

Now we will make use of Device ontologies, QoS ontologies and LimboConfig ontology to implement SWRL rules to guide the generation process considering the QoS characteristics of different transportation protocols. This also paves the way for considering other QoS parameters for code generation, for example when the RESTful web service is ready, we can then add rules to choose what kind of web services should be used in what situations, using the same set of ontologies.

8.3 SWRL rules for guiding the Limbo web service code generation

Currently it is the developer's responsibility to chose which communication protocol to use for SOAP transportation. We need to automate this process using SWRL rules, and will implement it as a separate configuration component.

The following rule will make sure that if the throughput requirement for a device is greater than 800 kBps, which does not have a CPU, i.e. that should generate a proxy service for this device, should use the SOAP over UDP as the underlying transportation protocol, as there is no further QoS requirements for reliability.

Rule: genUDP
$device : HydraDevice(?a) \wedge$
$device : hasHarware(?a, ?h) \wedge$
$hardware : primaryCPU(?h, ?c) \wedge$
$hardware : cpuName(?c, ?b) \wedge$
$swrlb : equal(?b, "No") \wedge$
$qosspec : hasDemand(?ser, ?demand) \wedge$
$qos : hasMetric(?demand, ?metric) \wedge$
$QoSmetric : hasParameter(?metric, ?para) \wedge$
$QoSmetric : parameterName(?para, ?pname) \wedge$
$swrlb : equal(?pname, "throughput") \wedge$
$QoSmetric : Value(?para, ?v) \wedge$
$QoSmetric : hasUnit(?para, ?unit) \wedge$
$abox : hasURI(?unit, ?uri) \wedge$
$swrlb : contains(?uri, "kBps") \wedge$
$swrlb : greaterThan(?v, 800) \wedge$
$limbogen : GenerationType(?current) \rightarrow sqwrl : select(?a, ?demand, ?current) \wedge$
$limbogen : genTransportation(?current, limbogen : UDP)$

Similarly, a battery consumption requirement for a device will probably make it necessary to make use of the SOAP over bluetooth, as shown in the following rule.

Rule: genBT
$device : HydraDevice(?a) \wedge$
$device : hasHarware(?a, ?h) \wedge$
$hardware : primaryCPU(?h, ?c) \wedge$
$hardware : cpuName(?c, ?b) \wedge$
$swrlb : equal(?b, "ARM") \wedge$
$qosspec : hasDemand(?ser, ?demand) \wedge$
$qos : hasMetric(?demand, ?metric) \wedge$
$QoSmetric : hasParameter(?metric, ?para) \wedge$
$QoSmetric : parameterName(?para, ?pname) \wedge$
$swrlb : equal(?pname, "batteryDrain") \wedge$
$QoSmetric : Value(?para, ?v) \wedge$
$QoSmetric : hasUnit(?para, ?unit) \wedge$
$abox : hasURI(?unit, ?uri) \wedge$
$swrlb : contains(?uri, "watt") \wedge$
$swrlb : lessThan(?v, 0.5) \wedge$
$limbogen : GenerationType(?current) \rightarrow sqwrl : select(?a, ?demand, ?current) \wedge$
$limbogen : genTransportation(?current, limbogen : BT)$

8.4 Prototype implementation

In accordance with the problem that Equinox is not working well in the open computing environments, the prototype is implemented to improve Eclipse Equinox's capabilities of awareness of component semantics based on SACoCo ontologies, QoS ontologies, Device ontologies, LimboConfig ontology. Architectural styles and component configurations validating are conducted at run time. The architecture of the prototype implementation is shown in Figure 8.2, in which there are EnhancedEquinoxDS bundle, *LimboConfigValidation* bundle.

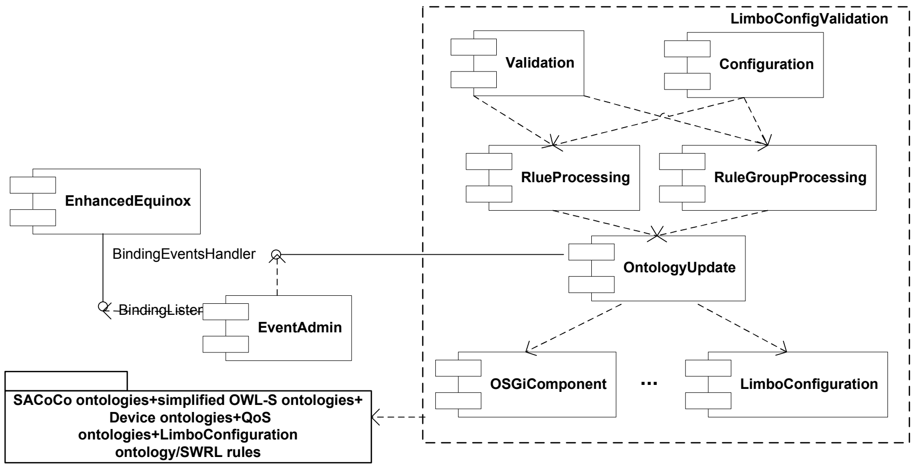


Figure 8.2: Limbo configuration and validation prototype architecture

This *EnhancedEquinoxDS* bundle is used (which is an extension to the original implementation) to discover and maintain a model of the component instance topology, where there is a Binding Listener that knows when services are bound to (and unbound from) components. Whenever such an event happens, the Binding Listener uses the standard OSGi Event Admin that provides a topic-based publish/subscribe service. This enables the Limbo configurator to maintain a model of component instances, their services, and their relationships. The registered services of the component are retrieved from the bundle context of the component.

In order to get detailed information about the services provided by the components that are loaded in the OSGi framework, the *LimboConfigValidation* bundle gets a list of enabled component description properties from the *EnhancedEquinoxDS* bundle (as another extension to the original DS implementation). These component details are then updated into the SACoCo models as a set of component instances to be validated. Based on this information, the *LimboConfigValidation* bundle validates component configurations by executing the related SWRL rules, and inferring whether they are valid or invalid, whether services are matching or not according to semantic constraints specified as outlined in the former sections.

We are using the Protege-OWL/SWRL APIs[4] (which are the only SWRL APIs currently available) for the prototype implementation to validate Limbo architectural style and configurations. From our experience, SWRL rule grouping will enhance the performance of validation (Zhang and Hansen, 2008). Therefore, in the prototype, all rules in a step are executed with the rule grouping features, for example the checking of the Repository style elements can be executed with the key word of “check_RepositoryStyle” from the rule names, and the key words can be combined with boolean operations.

The RuleProcessing component is used to execute a single rule and to retrieve the corresponding results. The RuleProcessing component is responsible for the execution of a rule group. Currently the rule grouping is based on the name of the SWRL rules, which can be logically combined (e.g. *AND*, *OR*). The rule processing features are generic and can be used to process all different kinds of rules, i.e. they are independent of architecture styles, components/connectors.

[4] http://protege.stanford.edu/

The rules for guiding the generation process considering the QoS requirements are ready as shown above, but are not integrated into this prototype yet and will be finished in the near future.

9 Integration between Rule-based configuration management, ASL and Limbo

The Architectural Scripting Language is a set of operations used to manipulate an architecture. Limbo is the Hydra-developed configurable web service compiler for embedded devices. In combination with the rule-based and ontology-based reasoning they form a powerful tool-suite for managing the inherent complexity of building a distributed system for deployment on a heterogeneous set of embedded devices, that is, one with multiple simultaneous target platforms. In this chapter we describe the integration of these tools.

The overall workflow when using these tools to build a system has three steps. First, the rule based reasoning engine tool determines a target configuration. The notion of a target configuration is construed broadly, as a set of architectural entities obeying a set of constraints. The entities are architectural, and could be binary components and target platforms for a deployment view; a set of modules or packages for a module-view; or a set of services and connectors for a runtime view, to name a subset. Second, the configuration is used to generate an ASL script that when executed with the required resources will generate the target configuration. This is described in section 9.2. Third, in order to generate code as specified in the target configuration ASL needs to configure Limbo with the backends capable of producing code for the right platform. The design for enabling ASL to configure Limbo is described in the following section.

9.1 Integration of Limbo with ASL

In one form ASL is a set of tasks that extend the well known configuration management tool ANT. It is described in (Ingstrup and Zhang, 2008), with a tutorial in (Atta Badii and Adedayo Adetoye (Ed), 2008). The functionality of ANT-based ASL is accessed as a set of operations for:

- Starting and stopping devices
- Deploying and undeploying components (bundles for OSGi)
- Starting and stopping services (likewise bundles in OSGi)
- Binding and Unbinding interfaces (service-interfaces in the case of OSGi)

There are in principle two ways to integrate ASL and Limbo. First, since Limbo itself is configurable its configuration can be managed with ASL. Doing so is useful because the configuration of limbo is a natural part of the subsequent compilation, a task in the development process that can also be automated with ANT. Second, ASL can be extended with support for the Limbo target platforms, so that it can be used to deploy the services limbo compiles. Since the second is an ASL specific extension and relies on the first, we have focused on the first case of using ASL to configure Limbo itself.

In this section we describe a design for integrating ASL with Limbo. The design has not been implemented in full at this point, although important parts of it are in place. In particular, ASL already supports the OSGi platform that Limbo is implemented with.

Limbo's architecture consists of the compiler component (bundle) itself, a repository bundle, a number of frontend as well as a number of backend components. These are bound together using the OSGi's notion of service-interfaces. In order set up a configuration of

Limbo, a series of architectural operations are required (the parentheses gives the name of the ASL operation used to accomplish this step:

1. An instance of the OSGi framework/platform must be started (using start_device())
2. The bundle jar files for the components used in the desired configuration must be available (using define_component())
3. The components are deployed to the running instance of the platform (using deploy_component)
4. The components are started in the correct sequence. Once started they announce their service-interfaces so that other bundles can be bound to them. (start_component())
5. The components of Limbo are bound (using bind_interfaces()).

At this point all of the steps are supported in ASL, and the integration only requires modifying the Limbo components so they support the bind_interfaces() operation of ASL.

In Java based OSGi, a service-interface is just a java interface type. A client component has an object (client-object) with a variable that references the server component's implementation of that interface. The server component that provides a service-interface has an object implementing this service-interface, and that is registered with, and obtainable through, the OSGi runtime. The ASL bind operation needs a reference to both the client-object and the server-implementation, because it needs to set the client-object's variable so that it points to the server component's implementation of that interface.

ASL is able to obtain references to these two objects as follows. The server-implementation object is easily obtained through look-up of the service interface in the OSGi runtime. The client-object is obtained using a special interface Bindable, which the client component must implement. The bindable interface has two methods, getBindMethod(String class) and getUnbindMethod(String class). They are used to get, given the class, the name of the method on the client-object that is used to set its reference to the server-implementation.

9.2 Configurations with ASL scripts

An ASL script operates on a configuration of architectural entities, in this case a set of compilation units that should be compiled to generate deployable units, web services. Numerous constraints may exist among devices, web-services, compiled components etc. For instance, a SOAP-over-Bluetooth web service should only be deployed to a device with Bluetooth communication. The generated ASL scriptis only correct if it does not violate such a constraint.

Formally, the execution of an ASL script can be modeled as a trace. An ASL script is a list of ASL operations with specific values as parameters. A script operates on a start-configuration when it is executed. After the first operation, say start_device(), this configuration is modified by having a new active device; as such we have a new configuration after each step in the script execution. The trace of a script execution is a list of such intermediate configurations, and the transition between two adjacent configurations is an invocation of an ASL operation. That is, the call of an ASL operation with specific values assigned to its parameters and to the script-properties it reads.

A script execution is correct if none of the configurations in its trace violate the relevant constraints on the system configuration, and if none of the operation-invocations are illegal for the configuration they operate on. For instance, a device cannot be started before it is defined, and components cannot be deployed onto it if it has not been started beforehand.

9.3 Integration of SWRL reasoning with ASL for the Limbo code generation

The execution of rules *Rule: genUDP* and *Rule: genBT* will tell us that the current Limbo configuration should use the "UDP", and "BT" transportations respectively. These kind of reasoning results can be inserted into an ASL script-template, so they are fed as the corresponding parameters for the Limbo compilation. A script-template is just a script which is incomplete because, in this case, the paramters for Limbo must be added to complete the script. In this case, it is the "-c" parameter for choosing the SOAP transportation approach.

As shown in rules *Rule: check_OSGi_Reference_Details* and *Rule: check_OSGi_Reference_noDetails*, and also the rules shown in deliverable D4.2b (Hansen et al., 2008c), it is possible to tell whether the current set of Limbo components is a valid configuration or not.

Once the target configuration has been decided, an ASL script needs to be generated which can actuate the configuration. This can be done with a simple template that the output form the rule-based semantic reasoning is filled into to generate a complete script. The ASL interpreter can then be used to actuate these Limbo components to generate code, and potentially further to compile the generated code, and make the services runnable automatically. This remains as to be further explored.

10 Conclusion and future work

Limbo is extended with the support of the generation of Hydra specific services, the generation of Eclipse specific project files in order to ease of the usage of the generated pervasive web services for the developer, and the generation of SOAP transportation over different communication protocols including TCP, UDP and Bluetooth, based on study made in D5.9 (Sperandio et al., 2008). Hence now Limbo is capable of considering QoS while generating pervasive web service code. We also add initial support for Limbo to generate RESTful web services.

In the future Limbo will work on automatic selection of the most suitable protocol considering QoS that will take into account several network parameters and give the correct configuration to Limbo. As future work we will add full support in Limbo for RESTful services, we will also work on the issues raised by developers on interoperability of Limbo generated code with AXIS and PHP, and we will work on in the support for .NET services.

Bibliography

Atta Badii and Adedayo Adetoye (Ed) (2008). External developers workshops teaching material. Technical Report D12.5, Hydra Consortium. IST 2005-034891.

Fielding, R. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California.

Hansen, K. M., Soares, G., and Zhang, W. (2007). Embedded Service SDK Prototype and Report. Technical Report D4.2, Hydra Consortium. IST 2005-034891.

Hansen, K. M., Zhang, W., and Fernandes, J. (2008a). Osgi based and ontology-enabled generation of pervasive web services. In *15th Asia-Pacific Software Engineering Conference*, pages 135–142, Beijing, China.

Hansen, K. M., Zhang, W., Fernandes, J., and Ingstrup, M. (2008b). Semantic web ontologies for ambient intelligence: runtime monitoring of semantic component constraints. In *Proceedings of the First International Research Workshop on The Internet of Things and Services*, Sophia-Antipolis, France.

Hansen, K. M., Zhang, W., Fernandes, J., Sperandio, P., and Ferrarini, M. (2008c). Embedded Service SDK Prototype and Report-version 2. Technical Report D4.2b, Hydra Consortium. IST 2005-034891.

Hansen, K. M., Zhang, W., and Soares, G. (2008d). Ontology-enabled generation of embedded web services. In *Proceedings of the 20th International Conference on Software Engineering and Knowledge Engineering*, pages 345–350, Redwood City, San Francisco Bay, USA.

Ingstrup, M., Hansen, K. M., and Zhang, W. (2008). Self-* properties SDK prototype and report. Technical Report D4.3, Hydra Consortium. IST 2005-034891.

Ingstrup, M. and Zhang, W. (2008). Self* properties DDK Prototype and Report. Technical Report D4.8, Hydra Consortium. IST 2005-034891.

Scholten, M. and Shi, L. (2008). Quality-of-Service Enabled HYDRA Middleware. Technical Report D4.5, Hydra Consortium. IST 2005-034891.

Sperandio, P., Antolin, P., and Bublitz, S. (2007). Draft of Wireless Devices Integration. Technical Report D5.4, Hydra Consortium. IST 2005-034891.

Sperandio, P., Bublitz, S., and Fernandes, J. (2008). Wireless Device Discovery and Testing Environment. Technical Report D5.9, Hydra Consortium. IST 2005-034891.

Zhang, W. and Hansen, K. M. (2008). Semantic web based self-management for a pervasive service middleware. In *Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2008)*, pages 245–254, Venice, Italy.

A Published Papers

A.1 Paper 1: Semantic Web ontologies for Ambient Intelligence: Runtime Monitoring of Semantic Component Constraints

please see the appendix of D 4.8 (Ingstrup and Zhang, 2008).

A.2 Paper 2: Towards Self-Managed Executable Petri Nets

please see the appendix of D 4.8 (Ingstrup and Zhang, 2008).

A.3 Paper 3: Semantic Web based Self-management for a Pervasive Service Middleware

please see the appendix of D 4.8 (Ingstrup and Zhang, 2008).

A.4 Paper 4: Towards Self-managed Pervasive Middleware using OWL/SWRL ontologies

please see the appendix of D 4.8 (Ingstrup and Zhang, 2008).

A.5 Paper 5: An OWL/SWRL based Diagnosis Approach in a Pervasive Middleware

please see the appendix of D 4.8 (Ingstrup and Zhang, 2008).

A.6 Paper 6: Flexible Generation of Pervasive Web Services Using OSGi Declarative Services and OWL Ontologies

2008 15th Asia-Pacific Software Engineering Conference

Flexible Generation of Pervasive Web Services using OSGi Declarative Services and OWL Ontologies

Klaus Marius Hansen and Weishan Zhang and João Fernandes
Department of Computer Science, University of Aarhus
Aabogade 34, 8200 Århus N, Denmark
{klaus.m.hansen,zhangws,jfmf}@cs.au.dk

Abstract

There is a growing trend to deploy web services in pervasive computing environments. Implementing web services on networked, embedded devices leads to a set of challenges, including productivity of development, efficiency of web services, and handling of variability and dependencies of hardware and software platforms. To address these challenges, we developed a web service compiler called Limbo*, in which Web Ontology Language (OWL) ontologies are used to make the Limbo compiler aware of its compilation context such as device hardware and software details, platform dependencies, and resource/power consumption. The ontologies are used to configure Limbo for generating resource-efficient web service code. The architecture of Limbo follows the Blackboard architectural style and Limbo is implemented using the OSGi Declarative Services component model. The component model provides high flexibility for adding new compilation features. A number of evaluations show that the Limbo compiler is successful in terms of performance, completeness, and usability.*

1 Motivation and introduction

Web services promise interoperable, composable, and reusable services in architectures in which Service-Oriented Architecture principles are employed. Increasingly, and from a deployment viewpoint, such architectures also includes resource-constrained embedded devices.

On the other hand, we are facing a number of challenges for implementing web services on devices. First, web services should be sufficiently efficient in order to provide usable services on small devices, because embedded devices are constrained in memory, processor and energy resources. Secondly, development of embedded web services must handle variability of hardware and software, and possible dependencies between hardware and software. This is particularly true when different implementation language and communication protocols are involved. And, finally, is the question how to support developers of pervasive web service applications in order to improve productivity.

Code generation is an effective way to improve reuse and to improve the productivity as for the last challenge mentioned. To support this, we have developed an OSGi[1]-based and ontology-enabled web service compiler called *Limbo* to generate resource-efficient code. Limbo is part of a larger project, Hydra[2], in which middleware for networked embedded devices is researched and developed.

The idea behind Limbo is: from the methodological point of view, we propose using OWL[3] ontologies to encode device hardware and software details, including their dependencies, which can then be used as compiling contexts during code generation for a specific device; from the architecture design point of view, we use of the Blackboard architectural style realized through the OSGi platform. This OSGi-based implementation of Blackboard architecture make the adding of compiling for different targeted platform dynamic without affecting existing components. In this way, we liberate the developer the headache of understanding details and dependencies of various platforms, and at the same time can generate resource efficient code according to requirements and the details of a device. This paper describes our experiences with using OSGi while previous papers focus on the use of ontologies [4, 3].

The rest of the paper is structured as follows: in Section 2, we present the design and architecture of Limbo; followed by is the section on the OSGi-based Limbo implementation, illustrated with UPnP[4] plugin, and we describe how to use the generated code for the development of web services. Section 4 discusses ontologies used in Limbo and state machine code generation. Then we present an evaluation of web services generated by the Limbo compiler in

[1]http://www.osgi.org
[2]http://www.hydramiddleware.eu
[3]http://www.w3.org/2004/OWL/
[4]http://www.upnp.org/

Section 5, from the perspective of complexity, usability and performance. We compare our work with related work in section 6. Conclusions and future work end the paper.

2 Limbo design

2.1 Limbo architecture

The Limbo compiler should be flexible in terms of, e.g., supporting different type of communication protocols, service discovery protocols, and also programming platforms. WSDL (Web Services Description Language) [5] is an XML-based language for describing Web services and how to access them, and should be used as central point for web service generation. Such files are provided as input to Limbo and Limbo follows the "Blackboard" architectural patterns [7] in which a central *Repository* stores data (initially WSDL data) related to the transformation process and on which *Frontends* and *Backends* operate to read and write information.

Figure 1 shows the module structure of the Limbo compiler. Frontends generally process source artifacts (in particular web service interface descriptions in the form of WSDL files and ontology descriptions in the form of OWL files). Conversely, Backends produce target artifacts in the form of code (including web service stubs and skeletons, state machine stubs, device and service descriptions) and configuration files.

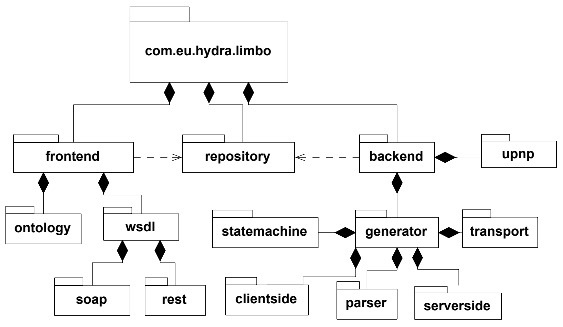


Figure 1. Module structure of Limbo

Backends may have different features. An essential feature is the parser backends with different implementation languages such as Java SE/Java ME (Java Standard Edition/Java Micro Edition). There may also be requirements for generation of client-side stubs and/or server-side skeletons, and for transport code for network communication between client and a server. To provide the possibility of handling dynamicity of device state changes, a state machine backend is designed to generate state machine stubs. At runtime, when executing the service, these state machines are intended to report relevant state changes of the device.

To provide the possibility of finding devices supporting different communication protocol, service discovery backend is also developed. UDDI[6] is designed for wired network for service discovery, which is not suitable for pervasive computing environment. For the Hydra middleware, UPnP is chosen as the main protocol for service discovery, therefore, we also have a UPnP backend, which is used to generate the UPnP device description, and the service description, in order to make both device and service available for UPnP to work with, so that the device can be discoverable by UPnP. This makes use of an ontology description of the device and service. Our use of ontologies is described next.

2.2 Extending WSDL for ontology binding

Originally WSDL has nothing to do with (OWL) ontologies. Recently, there have however been a number of efforts to link OWL with WSDL, for example OWL-S[7] and SAWSDL[8]. OWL-S is targeting automatic selection, composition, and execution of web services, which is quite different from what we require for linking WSDL with web service generation. Also OWL-S is heavy-weight solution in that it requires a major shift from WSDL to describe semantic services. SAWSDL is light-weight and links the service defined in a WSDL file to external sources of semantic description in OWL. This work inspired us to extend the semantics of WSDL binding, through a *hydra:binding*, in order to make use of ontologies during compilation. This new binding must refer to a device instance in our Device ontology (which imports other ontologies). An example of a Hydra ontology binding for thermometer in WSDL would be the following:

```
<hydra:binding device="http://hydra.eu.com/ontology/
                       Device.owl#thermometer"/>
```

Formally, this extension appears as follows in a definition of WSDL:

```
<wsdl:binding name="nmtoken" type="qname">*
<-- extensibility element --> *
<hydra:binding device="uri">?
<wsdl:operation name="nmtoken">*
...
</wsdl:operation>
</wsdl:binding>
```

The Limbo ontology frontend will resolve this URI and retrieve thermometer hardware and software information.

[5]Web Services Description Language 1.1. http://www.w3.org/TR/wsdl

[6]http://www.uddi.org/

[7]http://www.w3.org/Submission/OWL-S/

[8]http://www.w3.org/2002/ws/sawsdl/

2.3 Limbo compilatio process

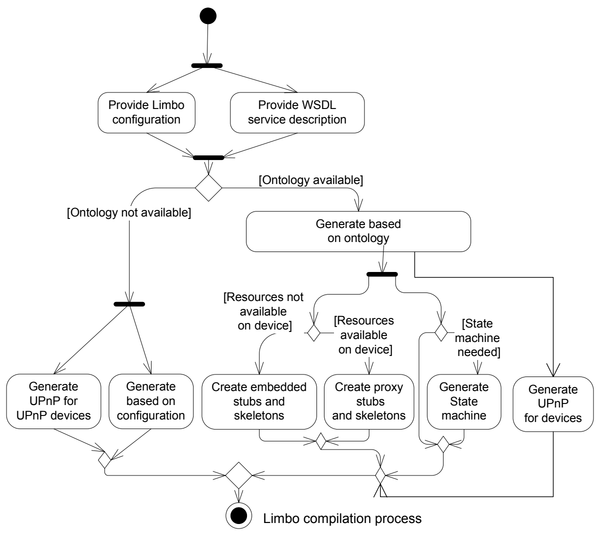

Provide Limbo configuration
Provide WSDL service description
[Ontology available]
[Ontology not available]
Generate based on ontology
[Resources not available on device]
[Resources available on device]
[State machine needed]
Generate UPnP for UPnP devices
Generate based on configuration
Create embedded stubs and skeletons
Create proxy stubs and skeletons
Generate State machine
Generate UPnP for devices
Limbo compilation process

Figure 2. Limbo compiling process

Figure 2 shows the compilation process of the Limbo compiler. In this paper, a "thermometer service" is used to illustrate the compilation and the usage of the generated artifacts. In the example, the service runs on a thermometer device, Pico TH03, and provides a temperature measurement upon request. The following steps are involved:

- *Provide WSDL service description*: The main input for Limbo is a WSDL file, and Limbo also supports that WSDL files reference the Hydra device ontology.

- *Generation based on configuration or ontology.* If an ontology instance for the device is available and referenced, device specific platform information will be used to generate client and/or server code. Otherwise, generation configuration is based solely on developer-supplied parameters.

- *Create embedded/proxy stubs and skeletons.* Stubs and skeletons for the device service are created according to the device capability. If there are not (enough) resources for running code directly on devices, proxy code is generated based on OSGi. For the thermometer, as it does not have any computing capability itself according to the retrieved platform information from the ontology, proxy code will be generated. Currently, Limbo supports Java SE and Java ME code generation.

- *Generate device state machine stubs.* If a device state machine instance is available in the StateMachine ontology which is imported by the Device ontology, and Limbo is configured to generate state machines, then state machine stubs will be generated for a device.

- *Generate UPnP device description and service description.* If a device supports UPnP, then the generation will be based on the default UPnP information provided by the device manufactures, or else the device description will be based on the information in the Device ontology, and UPnP service description will be generated based on WSDL description.

3 OSGi-based implementation of Limbo

3.1 OSGi Declarative Services

OSGi provides a set of services per default [6], one of which is management of *Declarative Services*. OSGi's Declarative Services (OSGi-DS) Specification [6] enables developers on the OSGi platform to declaratively manage service composition at runtime. Concretely, OSGi DS allows OSGi bundle developers to provide a XML-based description of *components* that may be instantiated at runtime to provide and require services. The following list shows an example of such a description which specifies the main component (*LimboComponent*) of the Limbo compiler.

```
<?xml version="1.0" encoding="UTF-8"?>
<component name="com.eu.hydra.limbo">
  <implementation class="com.eu.hydra.limbo.LimboComponent"/>
  <service>
    <provide interface="com.eu.hydra.limbo.generator.Generator"/>
  </service>
  <reference name="BACKEND"
     interface="com.eu.hydra.limbo.backend.Backend"
     cardinality="1..n"
     policy="dynamic"
     bind="addBackend"
     unbind="removeBackend"/>
  <reference name="FRONTEND"
     interface="com.eu.hydra.limbo.frontend.Frontend"
     cardinality="1..n"
     .../>
  <reference name="REPOSITORY"
     interface="com.eu.hydra.limbo.repository"
     cardinality="1..1"
     .../>
</component>
```

3.2 LimboComponent

The LimboComponent provides the Generator service that both Frontends and Backends may use. Frontends process source artifacts whereas Backends produce output artifacts (such as web service stubs and skeletons). Both Backends and Frontends may use a single Repository.

Furthermore, the LimboComponent requires the presence of at least one Frontend and at least one Backend and one Repository. When these services are available, the references are said to be satisfied and the component may be activated. Essentially, OSGi DS provides a way for components to specify provided and required services (in the form of Java interfaces) declaratively so that the OSGi framework can resolve service ependencies dynamically. This kind of constraints are specified with Semantic Web Rule Languge[9] (SWRL) rules and implemented using another OSGi component called LimboConfigurator, details of which are given in [3].

3.3 Limbo Plug-In Model

OSGi provides a service-oriented component-based platform for use by systems that require dynamic updates, and minimal disruptions to the running environment. This provides us more flexibility than the Blackboard architecture itself by allowing dynamically adding new compilation features, for example, the supporting of new service discovery protocols, and new requirement for generating code for different platforms.

A Limbo plug-in is a normal OSGi bundle and as such needs to implement the BundleActivator interface of OSGi. This interface has two methods, `start` and `stop`, which are used to start and stop the bundle respectively. In the method `start`, we register service interface that Limbo provides (the Generator interface).

The Generator interface has the following methods:

```
 public interface Generator {
  void addBackend(Backend backend);
  void removeBackend(Backend backend);
  void addFrontend(Frontend frontend);
  void removeFrontend(Frontend frontend);
  Repository getRepository();}
```

This allow plug-ins to register Backend services (that provide extra generation facilities) and Frontend services (that may add extra preprocessing) and to access the Repository that contains the code generation artifacts.

Furthermore, the backend plug-ins typically implement the Backend interface:

```
public interface Backend {
public void generate(Repository repository) throws Exception;}
```

When compiling, Limbo will invoke the `generate` method on all Backends.

3.3.1 UPnP component

The UPnP component is an example of a plug-in. The component creates device services that are discoverable by the UPnP protocol. The creation involves the generation of UPnP device and service descriptions in XML syntax. The generated code makes these descriptions including the devices' capabilities available on the network, so other devices can learn about the device. The UPnP component has the following description in XML format:

```
<?xml version="1.0" encoding="UTF-8"?>
<component name="com.eu.hydra.upnp.upnpcomponent">
<implementation class="com.eu.hydra.upnp.UPnPComponent"/>
<property name="platform">
JSE
</property>
<service>
<provide interface="com.eu.hydra.limbo.backend.Backend"/>
</service>
<reference name="Repository"
interface="com.eu.hydra.limbo.repository.Repository"
cardinality="1..1"
policy="dynamic"
bind="setRepository"
unbind="unsetRepository"
/>
</component>
```

3.4 Implementing services based on generated code

For the thermometer with a configuration of web services on the OSGi platform, the following classes will be generated:

- *Activator.java*: Defines start and stop methods for the thermometer service bundle.
- *th03OpsImpl.java*: Implementation of the service methods.
- *th03Parser.java*: Parser for SOAP messages.
- *LimboServlet.java*: A service class extends HttpServlet that handles requests and returns the respective results.
- *ThermometerSMStub.java*: State machine code utilized to handle the self-management features in Hydra.

For a thermometer client, the following classes are generated:

- *LimboClient.java*: Client main method to call the thermometer services.
- *LimboClientHeaderParser.java*: HTTP header parser.
- *th03LimboClientParser.java*: Parser for SOAP messages.
- *th03LimboClientPort.java*: Stub methods for the thermometer services.
- *th03LimboClientPortImpl.java*: Implementation for the stub methods.

To make the Thermometer discoverable, UPnP descriptions are generated as followed:

[9] `http://www.w3.org/Submission/SWRL/`

- *PicoTh03.xml*: Thermometer PicoTh03 UPnP device description, including model name, device type and so on.

- *Th03Service_scpd.xml*: Thermometer service as defined in Th03 WSDL file, following the UPnP specification.

The generation configuration can also be for Java ME server (extends the MIDlet class) or Java SE, but the general runtime architecture will remain the same as shown in Figure 3.

Based on the generated artifacts, the device developer needs to implement the device service. The development process includes:

- *Binding the device services to the actual device.* For the thermometer service this would include, e.g., creating a thread that continuously calculates the temperature based on sensor input and stores the temperature in a local variable. The actual service implementation would then read the value of this variable and return the temperature.

- *Sending state notifications.* The statemachine stub needs to be invoked at proper places. In the case of the thermometer, each successive call will at runtime trigger an event being sent through a publish/subscribe system (Figure 3): when the thermometer is started, when it is measuring, and when it stops.

- *Create deployment artifacts.* Next, device and container-specific deployment artifacts (JAR files, OSGi bundles etc.) need to be created in order to be able to deploy the service.

The left part of Figure 3 shows a typical runtime of a deployed Limbo service. The thermometer service is deployed on a Thermometer Device. A service that needs temperature data ("Thermometer Client") then uses the thermometer service through its web service interface. The state changes internally in the device triggers events sent through a publish/subscribe mechanism.

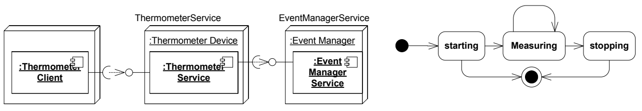


Figure 3. Thermometer runtime and its simplified state machine

4 Ontologies in Limbo

4.1 Motivation for ontology usage in Limbo

There are a number of reasons for us to use ontologies in Limbo:

- *Knowledge reuse.* Some ontologies are used across all parts of the Hydra middleware. For example, the Device ontology is used to model device capabilities, which can be used in Limbo to generate resource-aware code for services and which can also be used in self-management

- *Complexity hiding.* Details for device hardware and software can be encoded in the related ontologies. Web service developers only need to know about the device URI and the service they are implementing, as shown in the Thermometer example. Dependencies between software and hardware platforms are rigorously specified and may be transparent to developers

- *Generating resource efficient code.* In order to generate resource-efficient code, knowledge on device software platform and resource consumption comparisons are built into the related ontologies, and used during the configuration of Limbo for code generation

- *Support for device discovery.* In order to make devices discoverable, UPnP device descriptions should be generated for devices if desired. The ontologies encode knowledge (such as model name and number) which is used in UPnP but is not available in a service description

- *Rigorous configuration of components.* As not all combinations of Limbo components are valid, ontologies can rigorously regulate combinations of Limbo components and resolve dependencies among them

4.2 Details of Ontologies

We have developed the supporting ontologies for Limbo as shown in Figure 4.

The Device ontology is used to define high-level only information of a device, e.g., device type classification (e.g., an alarm device is a sensor).

The HardwarePlatform ontology includes concepts such as CPU and Memory, and also relationships between them, for example "hasCPU". Power consumption concepts and properties for different wireless networks are added to the HardwarePlatform ontology to facilitate power awareness.

The software platform-related ontologies specify resource consumption comparisons for different platforms using object properties such as *requiresMoreMemory*,

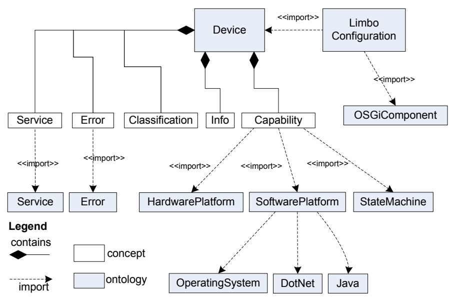


Figure 4. Structure of Limbo ontologies

reuqiresFasterCPU, and platform dependencies, for example, .NET can only run on Windows operating systems if there are no other supporting software packages.

The StateMachine ontology is used to generate state machine stubs which are used for self-diagnosis in the Hydra middleware. The state machine ontology is based on [1] with several improvements, for example we add a data-type property *hasResult* to the *Action* (including activity) concept in order to check the execution result at runtime.

To enhance the semantics of OSGi component, we developed an OSGiComponent ontology based on OSGi-DS where details are given in [3]. Limbo can make use of different backends and frontends for a specific compilation. Not all combinations of frontends and backends in Limbo are valid. For example, for OSGi, there is no need for the Server backend as a web server is built into the OSGi framework. SWRL[10] rules are used specify possible feature combinations and are validated at run time by a configurator component based on the OSGiComponent ontology as shown in [3].

In order to generate resource-efficient code, Limbo will utilize the resource/power consumption knowledge built in the ontologies. Therefore the LimboConfiguration ontology imports the Device ontology, and hence all other ontologies through the ontology import mechanism. Object properties in the LimboConfiguration ontology (*requireCPU*, *requireOS*, *requireVM* and *requireLibrary*) are used to specify a backend's detailed requirements for the CPU, operating system, virtual machine and libraries.

5 Evaluation of Limbo

We have evaluated Limbo according to the evaluation framework of one.world [2]. This includes evaluating: *Completeness*: can useful services be generated;*Performance*: is the generated services sufficiently resource efficient; *Complexity and utility*: how hard is it to create services and can others build upon it. The details of the evaluation are related in [4]. Here we summarize the evaluation results and focus on plug-ins and ontology construction.

[10]http://www.w3.org/Submission/SWRL/

5.1 Completeness

We evaluate this through the generation of services for a set of prototypes for a set of home automation devices. Five services were created in total, four of them by Hydra members that did not participate in the development of Limbo and one by a Limbo developer. The general conclusion was that useful services could be generated and that Limbo was instrumental in supporting this.

The devices for which the services were developed were:

1. *Nokia N80*
2. *HTC P3300 smart phone*
3. *Pico TH03 thermometer*
4. *Grundfos Magna 32 pump*
5. *Abloy EL582 door lock service*

For the two first devices, the service were embedded in the device; for the other devices, an OSGi-based proxy was generated.

Moreover, the generated state machine code is used successfully for the development of self-diagnosis in Hydra, which shows its usefulness in state-based diagnosis and other self-management processes. The developer of the self-diagnosis does not need to have deep knowledge of web services. The generated eventing code, and state machine code helps development. The evaluations of the self-diagnosis show the usefulness of the Limbo compiler.

Furthermore, the following plug-ins have been developed, demonstrating the plug-in facility:

1. *UPnP plugin*. This plug-in has been presented previously in the paper
2. *Probe plugin*. This plug-in builds probing of message sends and message receive into Limbo-generated services. The information is then used for self-management purposes in the Hydra middleware
3. *StateMachine plug-in*. The plug-in generates state machine stubs based on a statemachine description in the ontology of a Device
4. *LimboConfiguration plug-in*. This plug-in implements the semantic-based component configuration approach introduced in [3].

5.2 Performance

Here we summarize the time and memory usage of Limbo-generated services. These are compared with Apache Axis[11]-generated services. Although Apache Axis was designed for server environments, this gives an indication of the level of resources that Limbo-generated services use.

We used a web service implementing an SMS service as a test case. This service was implemented directly on a device (Nokia N80) with Java ME for Limbo and using Java SE for both Apache Axis and Limbo. Here we show the results of making five consecutive invocations of the same service with an Apache Axis client, but the results are further discussed in [4]. The service running on Nokia N80 ("Limbo ME") is significantly slower than the others in startup because of network setup (we use Nokia's Raccoon software[12]) and because it sends an actual SMS.

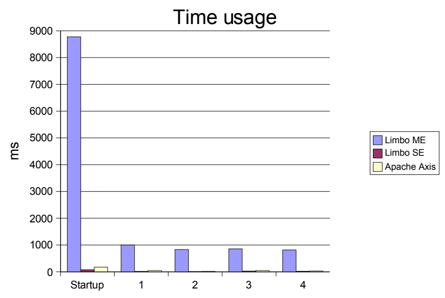


Figure 5. Limbo time usage

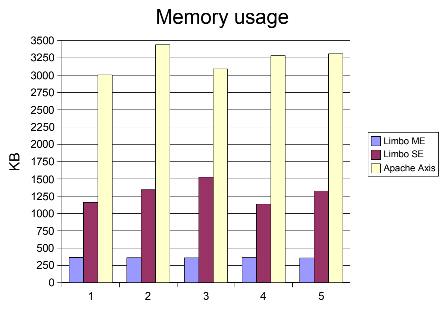


Figure 6. Limbo memory usage

[11]Apache Axis. http://ws.apache.org/axis

[12]Nokia Mobile Web Server. http://wiki.opensource.-nokia.com/projects/Mobile_Web_Server

5.3 Complexity and utility

Our primary means of investigating complexity and utility was through the prototyping of a service for an HTC P3300 smart phone[13]. The prototype was made by Hydra participants not involved in Limbo development. The evaluation was successful.

An ontology engineer, who was unfamiliar with the device, but experienced in ontologies and knowledgeable of the Hydra ontologies was able to construct needed ontologies in a day. The resulting ontology is shown in Figure 7.

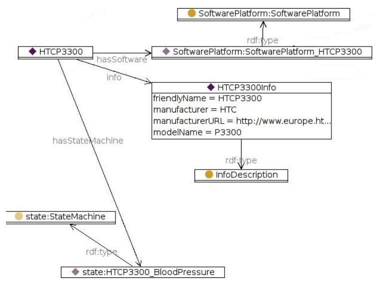


Figure 7. HTC P3300 ontology overview (Partial)

The Limbo compiler has been shown to be useful with low resource consumption of the generated code. The generated code is useful in developing services in Hydra, such as self-diagnosis service, network management service for a peer-to-peer network, and can also be used by other development outside Hydra.

6 Related work

Compared to the former version of Limbo [4], the new version based on OSGi-DS is much more flexible and extensible, where no service tracking for backends and frontends are needed and can be added dynamically, and will be validated by a configurator executing SWRL rules to validate the configurations as shown in [3]. More features are added to Limbo with the improved compilation process.

The existing tools and techniques for the development of (pervasive) web services, such as Microsoft's Web Services on Devices[14], kSOAP2[15], and Fast Infoset[16], fall short of the necessary flexibility of generating different code artifacts for the large variety of devices based on different protocols and quality of service requirements. These tools

[13]`http://www.europe.htc.com/en/products/htcp3300.html`

[14]http://www.microsoft.com/whdc/rally/Rallywsd.mspx

[15]http://ksoap2.sourceforge.net/

[16]https://fi.dev.java.net/

may thus lack the versatility of being used for different embedded devices. We are making use of ontologies to make our tool capable of handling different type of devices, and Limbo can be easily extended to support other service discovery protocols than UPnP.

XML Screamer [5] is an example of a tool that generates specific XML parsers for a specific XML Schema. Our work is clearly related to this in that we generate specific parsers for SOAP XML data described by a WSDL files (that may also contain an XML Schema specification). So we cast our work in the context of web services with the purpose of supporting ressource-constrained devices and use ontologies to guide this process flexibly. XML Screamer focuses purely on time usage.

Apache Muse[17] can be used to build web service interfaces for resources an example of which could be devices. Limbo has a highly flexible architecture which can be easily extended with the generation of code for .NET code, and other specialized platform whereas Apache Muse focuses on specialized specifications. Moreover, we use ontologies and rule languages to specify and regulate the generation process.

7 Conclusions and future work

There is an increasing requirement to run web service for resource-constrained devices in pervasive computing. To develop pervasive web services for embedded devices with high productivity, and to create resource efficient code is a challenge. In this paper, we propose a generation-based pervasive web service development approach, powered by supporting OWL ontologies. This approach is exemplified with a compiler called Limbo for the generation of embedded web services. Limbo has followed the Blackboard architecture style and is implemented as OSGi DS components in Eclipse Equinox, where different frontends and backends can be easily added.

Limbo gets the targeted device information from a Device ontology that imports a hardware platform ontology and software platform related ontologies, where resource consumption comparisons are specified, and used by Limbo to achieve the generation of resource-efficient web services. A StateMachine ontology is used to generate state machine stub code and using a publish/subscribe system to publish state change events. We are using a configuration ontology to rigorously specify the legal feature combinations of the Limbo compiler. To make the embedded devices discoverable, Limbo can generate UPnP device and service descriptions based on the device information in the Device ontology and WSDL descriptions.

Our evaluations show that the design of the Limbo compiler is successful in terms of resource consumption of the generated web services, complexity hiding of both device hardware/software details and the web service itself, and that developers can use Limbo to develop resource efficient web services for a variant of different embedded devices. The Limbo compiler is also used to generate code (event publishing/subscribing, state machine) for the development of self-diagnosis feature in Hydra, which demonstrates its usefulness for pervasive service development.

More backend components supporting other embedded platforms, for example LeJOS and Microsoft .Net, are also under planning. To support more efficient transport of SOAP messages, we are investigating to use UDP as the underlying transportation protocol. To improve the usability of the Limbo tool, an integrated development environment using Eclipse is scheduled to help user make decisions on choosing the generating types, for example software platforms.

[17]Apache Muse project. `http://ws.apache.org/muse/`

Acknowledgements

The research reported in this paper has been supported by the Hydra EU project (IST-2005-034891). Thanks to Goncalo Soares for the implementation of the first version of Limbo and performance measurements.

References

[1] P. Dolog. Model-Driven Navigation Design for Semantic Web Applications with the UML-Guide. *Engineering Advanced Web Applications, In Maristella Matera and Sara Comai (eds.)*, Dec. 2004.

[2] R. Grimm, D. Wetherall, J. Davis, E. Lemar, A. Macbeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, and S. Gribble. System support for pervasive applications. *ACM Transactions on Computer Systems (TOCS)*, 22(4):421–486, 2004.

[3] K. M. Hansen, W. Zhang, J. Fernandes, and M. Ingstrup. Semantic web ontology for ambient intelligence: Runtime monitoring of semantic component constraints. In *The Internet of Things and Services, 1st International Research Workshop*, Sophia Antipolis, French, Sept. 2008. To appear.

[4] K. M. Hansen, W. Zhang, and G. Soares. Ontology-enabled generation of embedded web services. In *The 20th International Conference on Software Engineering and Knowledge Engineering*, pages 345–350, Redwood City, San Francisco Bay, USA, Jul. 2008.

[5] 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.

[6] OSGi Alliance. OSGi Service Platform – Service Compendium. Technical Report Release 4, Version 4.1, OSGi, April 2007.

[7] M. Shaw. Some Patterns for Software Architectures. *Pattern Languages of Program Design*, 2:255–269, 1996.