Contract No. IST 2005-034891

Hydra

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

D4.2 Embedded Service SDK 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 **December 28, 2007- version 1.4**
Lead Contractor: University of Aarhus

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

Dissemination Level: Confidential

Document File: D4.2 Embedded Service SDK Prototype and Report
Work Package: 4
Task: 2
Document Owner: University of Aarhus

Document history:

Version	Authors	Date	Changes Made
1.0	Klaus Marius Hansen and Weishan Zhang	2007-10-06	First draft of deliverable based on papers
1.1	Klaus Marius Hansen	2007-10-25	Update of outline
1.2	Klaus Marius Hansen and Weishan Zhang	2007-12-16	Details and revisions. Ready for internal review
1.3	Klaus Marius Hansen	2007-12-17	Additional material on Limbo evaluation. Ready for internal review
1.4	Klaus Marius Hansen and Weishan Zhang	2007-12-22	Updates based on internal review. Ready for submission to commission

Internal review history:

Reviewed by	Date	Comments
Siegfried Bublitz, SAG	2007-12-18	Approved with comments
Peter Rosengren, CNET	2007-12-18	Approved with comments

Contents

List of Figures

1 Introduction

1.1 Purpose, context, and scope of this deliverable

The purpose of this deliverable is to present *Limbo* and associated technology which is an integral part of the Hydra SDK.

Generally, a Software Development Kit (SDK) contains runtime and development-time components as well as resources that support developers in building applications. In the context of WP4 and embedded services, this consists of the following components:

- *Development-time components.* The primary output of WP4 here is a *compiler* for semantic, embedded web services, *Limbo*, which is the main focus of this deliverable. Other typical components in this category which are not presently considered in Hydra are *linkers* and *assemblers*.
- *Runtime components.* A main part of an SDK are runtime *libraries* of which a major category in Hydra are managers. These are described in more detail in the WP3 deliverables and the Diagnostics Manager, *Flamenco* is described in D4.3 (Hansen and Zhang, 2007). Here we also describe how Limbo *plug-ins* may be created. Further components typically in this category are *virtual machines* and *debuggers*.
- *Resource components.* This document is part of the *documentation* of the SDK (and in particular Limbo). Furthermore, Limbo is documented in code (using JavaDoc), and the prototype also contains a user's guide. Finally, the ontologies used in Hydra (some of which are discussed in this report), are a part of the resources for developers.

The other parts of this deliverable are the first Limbo prototype and its associated documentation:

- Limbo prototype
  - `https://hydra.fit.fraunhofer.de/svn/trunk/sdk/limbo`
- Limbo tutorial
  - `https://hydra.fit.fraunhofer.de/confluence/x/NQUv`
- Limbo ontologies (used with Limbo1, no rules and LimboConfiguration ontology)
  - `https://hydra.fit.fraunhofer.de/svn/trunk/sdk/limbo/resources`

Furthermore, the following is needed in order to run Limbo services with state machines:

- Hydra event manager
  - `https://hydra.fit.fraunhofer.de/svn/trunk/middleware/eventmanager`

This material is also provided for review in a ZIP file as part of this deliverable.

1.2 Structure of the deliverable

The rest of this deliverable is structured as follows:

- First, Section 2 (page 8) presents the design goals of Limbo and how they relate to the Hydra requirements.
- Then Section 3 (page 10) explains the design of Hydra in relation to the goals.
- Section 4 (page 17) goes into detail with the use of ontologies in Limbo.
- In Section 5 (page 33) we present the evaluations of the current version of Limbo.
- Finally, Section 6 (page 39) concludes the deliverable

2 Design goals of Limbo

A number of goals have been driving the design of the Limbo compiler:

1. *Resource efficiency*. Web services developed based on artefacts produced by the Limbo compiler must be efficient in terms of memory usage, processing power, and network use. Effectiveness of communication is to some extent given by the protocols used: SOAP, e.g., is very resource consuming in terms of network usage whereas a REST-based protocol is more efficient although it still uses HTTP.

2. *Complexity hiding*. Artefacts generated by Limbo should hide the complexities of web services meaning that Limbo needs to take device characteristics into consideration during compilation. Limbo should help a web service developer to understand the intricacies of the devices the web service is developed for, and should reduce the burden of understanding device details.

3. *Dynamicity support*. We are considering devices in dynamic environments in which devices are moved, network connectivity is changing, and resources available vary. Therefore it is very important to know the states of devices in a system from the point of view of an application using device web services. This includes supporting context awareness from the level of devices.

4. *Platform decision support*. The heterogeneity of devices brings a large number of variants in software and hardware platforms. Some software platforms consume more memory and require more powerful CPUs than others. Limbo should help a web service developer to make decisions on the code produced and choosing appropriate platforms accordingly.

5. *Rigorously regulate compilation feature combinations.* Besides the hiding of complexity of targeting device software and hardware platform, the compiler should also be user-friendly to the web-service developer in terms of hiding compiler details, especially different parsers for different platforms and their inter-dependencies.

Furthermore, the following WP3, WP4, and WP6 requirements (which are the ones most related to Limbo) need to be taken into account:

HYDRA-017 When applicable, middleware interfaces are exposed by WSA-compatible services

HYDRA-019 Support of low-end devices

HYDRA-031 An easy-to-use programming framework should be provided

HYDRA-112 Dynamic Web Service Generation

HYDRA-114 Semantic enabling of device web services

HYDRA-151 Devices send events when their status changes

HYDRA-337 There should be a procedure/strategy for interfacing with non-HYDRA devices

HYDRA-366 Services should run on embedded devices

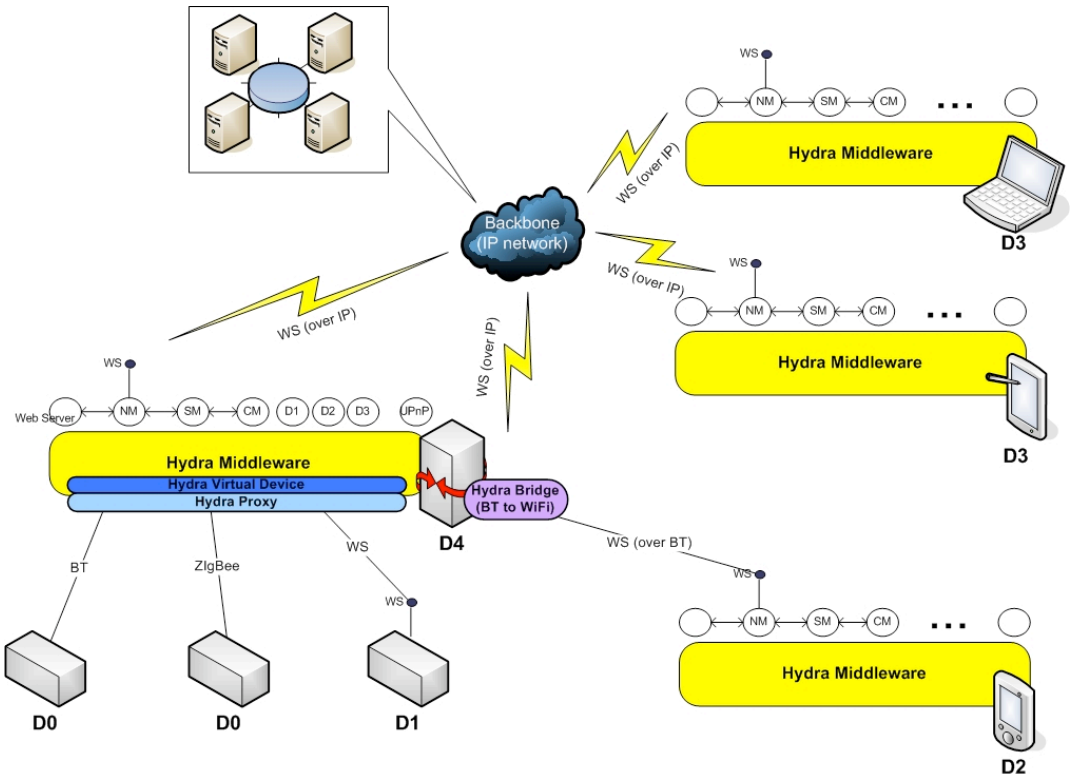

Figure 2.1: Device integration in Hydra

Furthermore, Limbo should in general support the Hydra middleware. The IP-based device integration in Hydra is illustrated in Figure 2.1 (taken from WP5). Limbo should directly support the generation of web services for D1 devices (see D5.4 “Draft of Wireless Devices Integration” for details), but should also support the generation of web services for D3 devices (which may, e.g., support Java ME).

3 Design and architecture

In order to exemplify the design in the following, we use a simple example of a service, a *thermometer service*, that is able to deliver a temperature when requested. In Java-like code, the service would have this interface:

```
public interface ThermometerService {
   double getTemperature();
}
```

3.1 The compilation process

Figure 3.1 shows the compilation process of the Limbo compiler. A device service developer initiates the process and the Limbo compiler generates service implementation artefacts. In

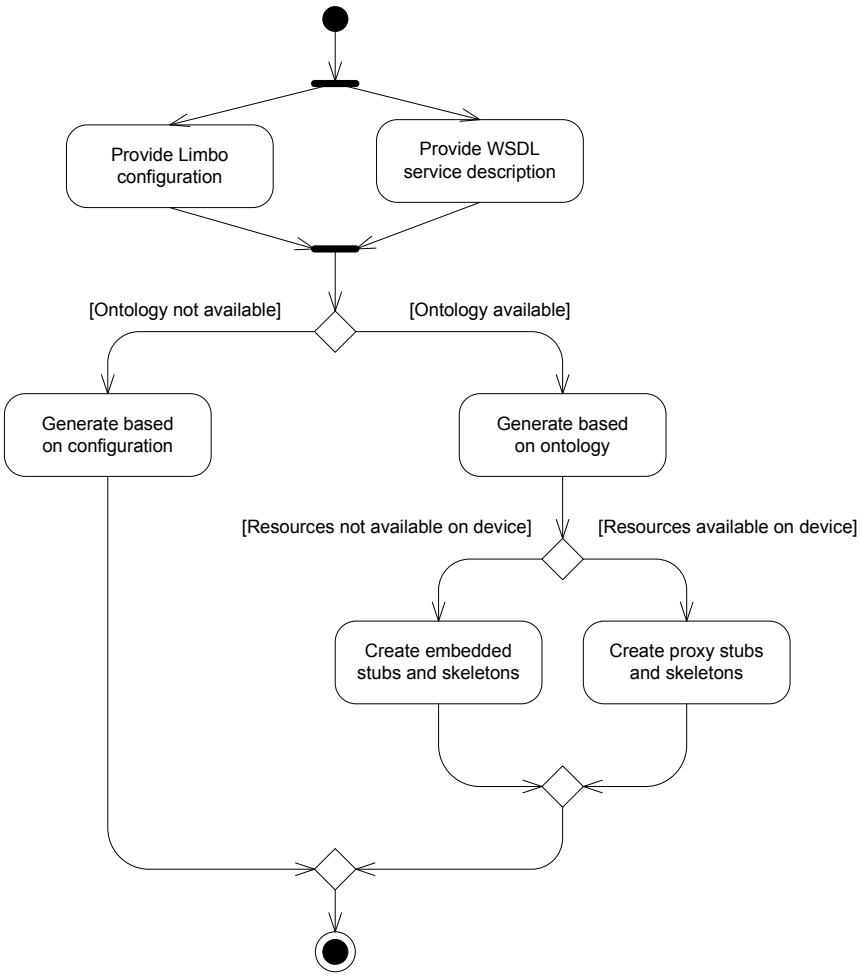

Figure 3.1: Limbo compilation

the following, we explain each of these steps[1]:

- *Provide WSDL service description*: The main input for Limbo is a service description written in the Web Service Description Language (WSDL). In addition to normal WSDL constructs, Limbo also supports that WSDL files reference the Hydra device ontology. For the thermometer example, this may look like:

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

  Formally, this extension appears as follows in a WSDL file specification:

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

  where

  ```
  xmlns:hydra="http://hydra.eu.com/"
  ```

  This may be seen as an extension of the WSDL-S semantic extension of WSDL files in that WSDL-S is not concerned with WSDL bindings whereas the Hydra extension is. Limbo resolves the URI and uses the device-specific information in the compilation process.

- *Provide Limbo configuration*: Limbo may be configured through a command-line configuration. In this way the device service developer may determine, e.g., that proxy code should always be generated

- *Generate based on configuration.* If no ontology is available (or specified) for the service, the generation of code artefacts is based solely on the developer-specified configuration to Limbo. (See “Create embedded stubs and skeletons” below for an explanation of generated artefacts)

- *Generate based on ontology.* If an ontology for the device is available this is used to guide the compilation. The ontology input consists of two main parts:

  1. A platform description. An excerpt of an example platform description for the thermometer example is shown in Figure 3.2.
  2. A state machine description. An example (simplified) state machine for the thermometer example is shown in Figure 3.3.

  Details on the use of ontologies are provided in Section 4.

[1] The Limbo manual, `http://...`, provides detail on how Limbo is concretely used

```
<hydra_device rdf:ID="thermometer">
  <info rdf:resource="#info_description_thermometer"/>
  <agent_compliancy rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean"
  >false</agent_compliancy>
  <hw_properties>
    <hw_description rdf:ID="hw_description_thermometer">
      <numOfChannels rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >3</numOfChannels>
      <connection rdf:resource="#Serial_connection_1"/>
      <cpu rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
      >PIC16C54C</cpu>
    </hw_description>
  </hw_properties>
  ...
</hydra_device>
```

Figure 3.2: Thermometer ontology excerpt

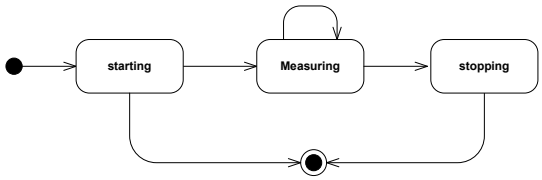

Figure 3.3: Thermometer state machine

- *Create embedded stubs and skeletons.* In both this and the “Create proxy stubs and skeletons” activity, stubs and skeletons for the device service are created. Stubs are client-side implementations of a service interface and skeletons are server-side partial implementations. Both of these are provided in the target language (Limbo currently only supports Java). A crucial part of the Limbo generation is generating a service-specific parser. Since XML grammars are quite simple (Murata et al., 2005) this can be done in an efficient way.
- *Create proxy stubs and skeletons.* In this case, OSGi-based artefacts are created. Instead of generating a simplified web server, the built-in HTTP Service of OSGi is used

3.2 Implementing services

For the thermometer with a configuration of a standalone server written in Java Standard Edition, the following classes will be generated. One can choose also to have an OSGi configuration (JSE) or a Java ME server, and finally one can generate clients either for Java ME or Java SE. The output for OSGi is standard (i.e. an Activator instead of an EndPoint, a th03Servlet instead of a th03Service).

- *EndPoint.java* - Abstract class that defines the EndPoints (i.e. services) that are provided by the server

- *th03OpsImpl.java* - Implementation of the service methods
- *th03Service.java* - Service class that handles the request and returns the respective result
- *LimboServer.java* - Limbo Server main class
- *StringTokenizer.java* - Hydra version of String Tokenizer, which was created in order to face the difficulties of this class being non-existent in JAVA Micro Edition

Figure 3.4: Limbo command line

Currently, the developer has to use a command line (see Figure 3.4) to generate services. Based on the generated artefacts, the device developer needs to implement the device service. This entails:

- *Binding the device services to the actual device.* For the thermometer service this would entail, e.g., creating a thread that continuously calculates the temperature for the thermometer and storing the value in a local variable. The actual service implementation would then read this local variable and return the temperature value.
- *Ensure state notifications are sent.* The state machine stub needs to be invoked at proper places. In the case of Figure 3.3 this is when the thermometer is started, when it is measuring, and when it stops (if possible). Each successive call will at runtime trigger an event being sent through the Hydra middleware.
- *Create deployment artefacts.* Currently, the device developers have to create deployment artefacts (JAR files, OSGi bundles etc) based on the generated code in order to be able to deploy the service

3.3 The runtime of services

Figure 3.5 shows a typical deployment of a Limbo service. The thermometer service is deployed on a Thermometer Device (either directly or by proxy as explained previously). A service that needs temperature data (“Thermometer Client”) then uses the thermometer service through its web service interface. Finally, the state changes internally in the device triggers events sent through the Event Manager of Hydra. Currently, the device service developer is responsible for implementing code that locates the Event Manager service, but the next version of Limbo should support that the generated services use the Hydra discovery protocol.

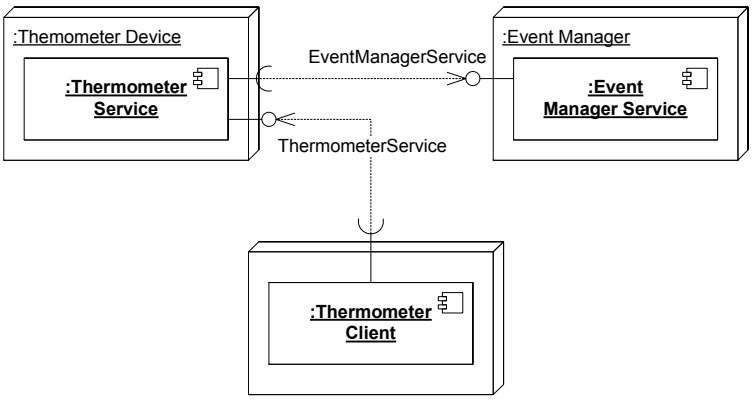

Figure 3.5: Thermometer runtime

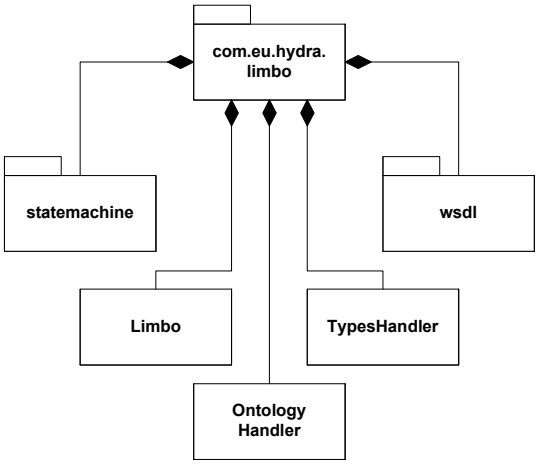

Figure 3.6: Limbo module structure

3.4 Limbo architecture

Figure 3.6 shows (part of) the current module structure of Limbo. The main responsibilities of the modules (packages and classes) are:

- *Limbo:* overall control and configuration handling
- *OntologyHandler:* general referencing of device ontology
- *statemachine:* classes for generation of device state machine based on the device ontology
- *wsdl:* classes for generation of service-specific parsers

The current implementation of Limbo is the one that has been implemented and evaluated (see Section 5, page 33). Although this evaluation can be said to be succesful it has been decided to redesign Limbo for the next Hydra iteration. Based on the experience in implementing Limbo and the evaluation, the following shortcomings have been identified:

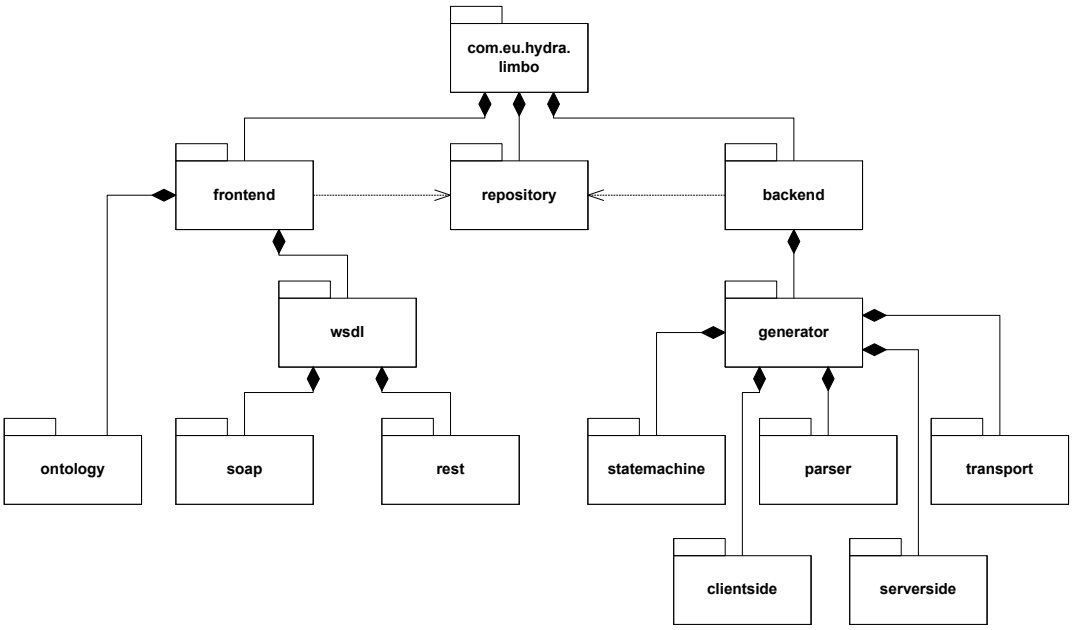

Figure 3.7: Limbo module structure

- The architecture is inflexible with respect to extensions. Adding a new application-level communication protocol such as REST (REpresentational State Transfer) is hard
- The architecture makes reasoning over configurations hard since it is not sufficiently modularized. The generation of server- and client-side code is, e.g., localized in the same module. Thus the reasoning on ontologies as presented in Section 4 is primarily implemented in a proof-of-concept version not integrated in the first version of Limbo.
- The missing modularization also makes flexible reconfiguration hard since modules cannot be freely combined in a meaningful way. It is not possible, e.g., to decouple the choice of a SOAP protocol from the choice of a specific serverside implementation

This has lead to the redesign of Limbo (called “Limbo2” in the rest of this deliverable) presented in Figure 3.7 which is now being implemented. The three main modules are:

- *frontend:* encapsulates processing of input artefacts. In our case this is *ontologies* and service descriptions (*wsdl*). The result of processing these are written into a *repository*
- *repository:* this encapsulates information in the compilation process
- *backend:* encapsulates the generation of output artefacts such as *statemachine*s, *parser*s, *clientside* artefacts, *serverside* artefacts, and *transports*

For the concrete realization of modules, an implementation based on OSGi bundles which will be annotated with ontology references (as described in Section 4.3.2, page 22) is planned. This allows for runtime configuration of the compiler modules based on ontological reasoning.

3.5 Limbo design vs. design goals

In this section, we briefly summarize how the design of Limbo attempts to meet the design goals:

1. *Resource efficiency*. The web services generated by Limbo may use a special-purpose web server and use a specialized XML parser for a specific service (see Section 3.1). Furthermore, the architecture of Limbo2 enables flexible choice of (more) resource-efficient service implementations such as REST.
2. *Complexity hiding*. The device service developer needs only to be concerned with target language artefacts. If the language is Java, the developer needs to override Java methods and invoke Java methods of the generated state machine. Ontology details, e.g., are thus hidden from the developer (if the developer wants so).
3. *Dynamicity support*. Devices report their state through the use of events. Currently, Limbo supports reporting of basic state information (i.e., that the device is in a given state), allowing state actions to be bound to service operation. But it is planned to make the state machine translation more complete (e.g., consideration of transition conditions).
4. *Platform decision support*. Platform decision is supported through configuration and through the use of ontologies (see Section 4) for more detail.
5. *Rigorously regulate compilation feature combinations*. Limbo2 supports automatic feature combination again through the use ontologies (see Section 4).

Furthermore, we briefly summarize the status with respect to the listed Limbo-related requirements.

HYDRA-017 When applicable, middleware interfaces are exposed by WSA-compatible services. Limbo attempts to generate compliant service, but compliance to any specific specification is at the discretion of the device service developer

HYDRA-019 Support of low-end devices. Limbo employes techniques to reduce resource utilization at service runtime

HYDRA-031 An easy-to-use programming framework should be provided. We attempt to use well-known programming frameworks (direct methods calls with target language objects) instead of being forced to use

HYDRA-112 Dynamic Web Service Generation. Limbo could be used as a component in the middleware runtime even though this has not been the focus thus far

HYDRA-114 Semantic enabling of device web services. We use semantics to guide the translation and generated services may, e.g., be used for semantic service composition.

HYDRA-151 Devices send events when their status changes. Limbo supports this through the state machines that are generated.

HYDRA-366 Services should run on embedded devices. This is in line with HYDRA-019

HYDRA-337 There should be a procedure/strategy for interfacing with non-HYDRA devices. This is currently done through the use of OSGi-based proxies that can be generated by Limbo

4 Ontologies in Limbo

4.1 Ontology structure

Ontologies are very important in the Hydra project to achieve context-awareness and fulfill the self-recovery requirements when there are failures, and also for achieving the design goals of the Limbo compiler. Corresponding to the design goal of complexity hiding, all details for device hardware and software are encoded in the related ontologies. Web service developers only need to know about the device URI and the service they are implementing. The device URI is the resource pointed to a specific device instance in the Device ontology. To model the dynamicity of a device, an state machine ontology is used to model device state and transition changes and it is linked to the device URI.

In order to provide resource-efficient code generation capabilities to Limbo (especially for Limbo2, as Limbo1 is using only the CPU information at present stage), knowledge on device software platform classification and resource consumption comparisons is built into related ontologies, and can be accessed through the ontology frontend of Limbo.

To this end, we design several OWL-DL ontologies including Device, Hardware, SoftwarePlatform, OperatingSystem, StateMachine and so on. The ontologies related to the Limbo compiler are shown in Figure 4.1.

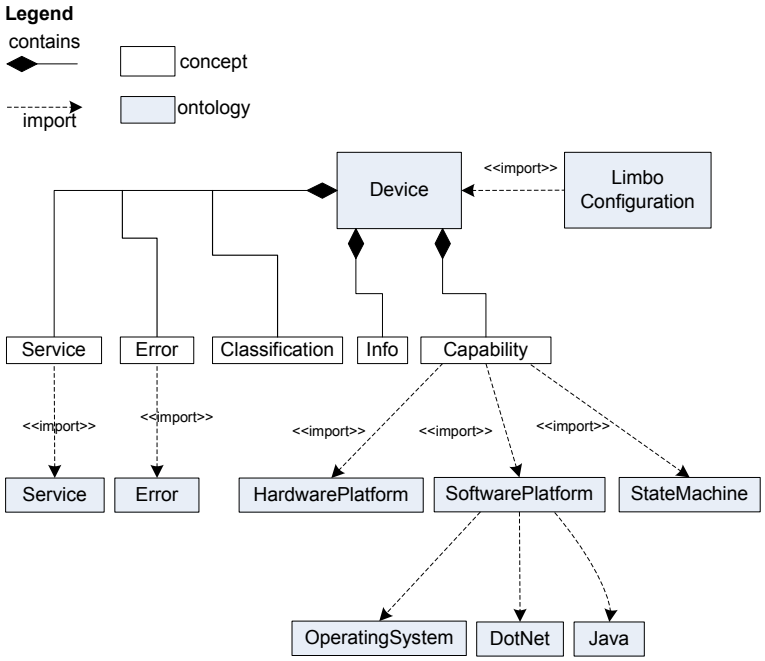

Figure 4.1: Structure of ontologies used in Limbo

The Device ontology, including its associated ontologies, are also overview-ed in D6.2 and some other details are given in D6.3, in the perspective of ontology model driven architecture (MDA). Here we discuss these ontologies in the perspective of Limbo design goals in the scope of WP4. And our work in the ontology based web service generation with Limbo,

is also part of the MDA work.

The usage of the shown ontologies are summarized as followed:

1. *LimboCongiguation ontology*. It is used to configure the Limbo2 components, in order to provide optimized valid combinations of different front end and back ends. Details for the configuration are shown in Section 4.3.2. Limbo1 does not need this ontology.
2. *Device ontology and associated hardware platform and software platform ontologies*. These ontologies are used to retrieve device specific information in order to generate resource-awareness code for a certain device. Details are also shown in Section 4.3.2. Limbo1 is using only the CPU information to decide the code generation type.
3. *StateMachine ontology*. This ontology is used to generate state and transition related code. Currently only state based code is used, and action in states is linked with actual service. Both Limbo1 and Limbo2 are using this ontology.

Service ontology and Error ontology are not used by Limbo.

4.2 Device Ontology and its associated ontologies

4.2.1 Device ontology

The Device ontology is used to define high level only information of a Hydra device, for example device type classification(e.g. mobile phone, sensor), and run time properties such as device error and its resolution information in order to support these self-recovery requirements. The Device ontology is based mainly on the concepts in AMIGO project ontologies(IST Amigo Project, 2006).

From now on, the legend for showing ontologies is shown in 4.2.

Figure 4.2: Legend for ontology

Part of the Device ontology is shown in Figure 4.3, where the Hardware and .NET ontologies are shown at the same time because of the import mechanism. This figure lists the device types used in the first Hydra prototype, i.e. Pump, Lock, MobilePhone and PDA.

Using the OWL *import* mechanism, ontologies for hardware and software information together with state machines for devices are linked under the Capability concept. The Hardware ontology is based on the hardware description part from W3C deliveryContext ontology[1]. It includes concepts such as Bluetooth, Network, CPU, Memory, Camera and so on, and also relationships between them, for example ”hasCPU”.

To correspond to the review comments on the power consumption: We add some improvements in order to manage power consumption issues, for example the concept of *Power* and datatype properties such as *voltage*, *current* and *frequency* of the power used by a device. More information on network connections are added including the *Bandwidth*, and its datatype properties such as *averageDataRate* and *maxDataRate*. *WorkingMode* concept (sleep mode, awake, using the radio) and its datatype *powerUsed*. All these improvements are helping us to achieve power-awareness as needed.

[1] Delivery Context Overview for Device Independence. http://www.w3.org/TR/di-dco/

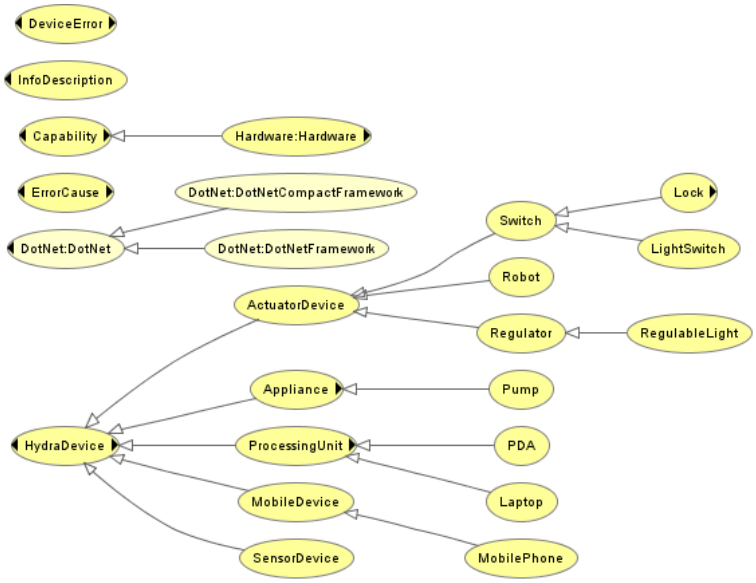

Figure 4.3: Device ontology

4.2.2 Software platform related ontologies

The software platform related ontologies are important for the Limbo compiler. They should provide the following functionalities in order to support the Limbo requirements:

- *Software platform classification*. As there are many concepts involved in the development of web services for embedded device, it is very important to provide a clear classification of these software in order to understand them. And more interestingly is to classify related concepts in order to support the reasoning on type of a concept which is not superficial. We will show examples later.
- *Rigorous specification of platform dependencies*. As there are lots of variants for software platform, dependencies between them should be rigorously defined. For example some programming languages need certain operating system support.
- *Resource consumption relationships*. As mentioned before, there are resource restraints for embedded devices, therefore it is important to know what kind of software platform is more/less resource consuming than others, in order to choose an appropriate platform for the corresponding requirements.

Several ontologies are designed to model the software related concepts and their relationships, including a SoftwarePlatform ontology, an OperatingSystem ontology, a Java ontology and a .NET ontology. The software platform related ontologies are developed based on (Wagelaar and Jonckers, 2005), where classifications and relationships among concepts are re-arranged and re-organized. And more importantly the improvements make the specification of resource consumption comparisons easier. Platform dependencies are also added, for example, .NET can only run on Windows operating systems if there are no other supporting software packages.

The SoftwarePlatform ontology defines VirtualMachine, Library, Middleware and object properties such as *requiresMoreMemory*, *reuqiresFasterCPU*, and their reverse properties *requiresLessMemory* and *requiresSlowerCPU*, provides and requires (certain feature or library). SoftwarePlatform ontology imports the OperatingSystem ontology, the Java ontology and the .NET ontology as shown in Figure 4.4.

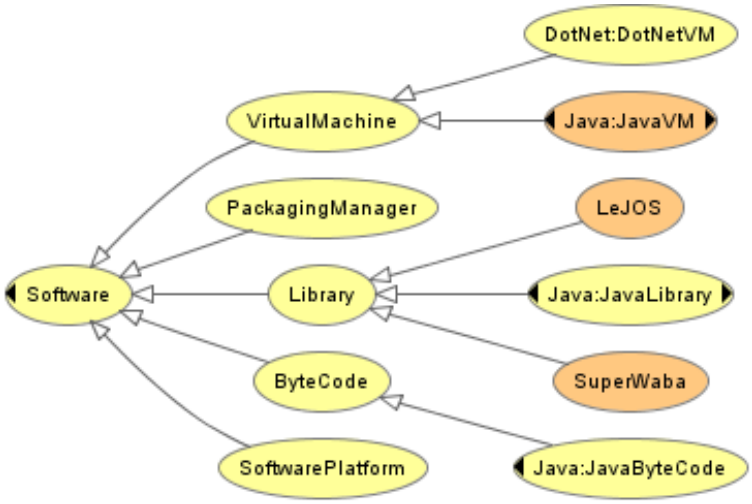

Figure 4.4: SoftwarePlatform Ontology

In the Java ontology we define concepts like JavaVM, JavaByteCode, JavaLibrary and specify that a specific Java platform (e.g. CLDC) provides which Library, Rendering Engine, etc. The classification of different versions of Java, such as J2EE, J2SE and MIDP1/MIDP2 is also presented in the ontology as shown in Figure 4.5.

The OperatingSystem ontology as shown in Figure 4.6 is used to specify operating system related concepts and properties. The classification of an operating system is based on its characteristics and version for example Win32/Win16. Such kind of classification could facilitate the restrictions on which operating system consumes more memory than others.

The overall view of the device ontology showing details is shown in Figure 4.7. This figure lists the device types used in the first Hydra prototype, i.e. Pump, Lock, MobilePhone and PDA.

4.3 Ontology reasoning in Limbo

Pervasive web services for networked and embedded devices should consider many aspects, for example power consumption and resource usage. These information may be not superficial, and sometimes very complex when combining with different wireless communication technologies, different hardware and software dependencies. Developers face the problem of appropriate platform selection and lost in details like details of hardware/software, communication power consumption details , etc. To help make reasonable decisions, ontology reasoning is used in Limbo, especially in Limbo2.

Ontology reasoning can be used to provide additional and helpful facts about devices and its hardware and software. These mainly include:

Figure 4.5: Java ontology

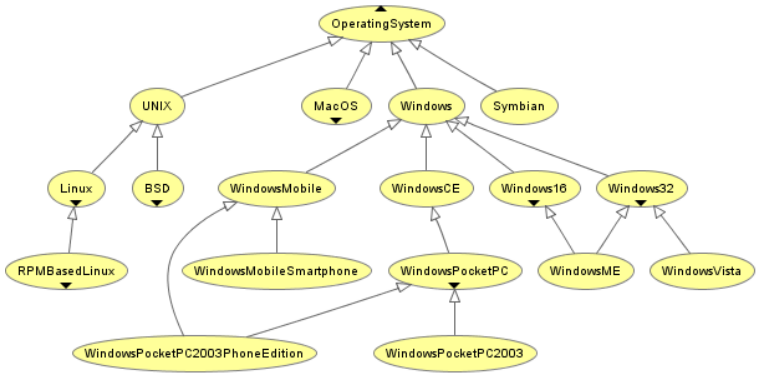

Figure 4.6: Operating system ontology

- Providing new classification information based on the subsumption reasoning of description logic (Nardi and Brachman, 2003). An example is:
  $JavaVM \sqsubseteq VitualMachine$
  $JavaVM \sqsubseteq Middleware$
  $JavaVM \equiv \exists runs.JavaByteCode$
  $SuperWaba \sqsubseteq Library$
  $SuperWaba \sqsubseteq \exists runs.JavaByteCode$
  Then $SuperWaba \sqsubseteq JavaVM$

JavaVM is defined as something that can run JavaByteCode, and SuperWaba is necessarily to run JavaByteCode, therefore SuperWaba is a subclass of JavaVM. This is helpful for someone who is not familiar with SuperWaba. The same kind of reasoning will classify LeJOS [2] as a kind of Java virtual machine.

- Choose the appropriate software platform according to the resources available. For example, *requiresMoreMemory* is defined as a transitive property.
  $requiresMoreMemory(CDC\ CLDC)$
  $requiresMoreMemory(J2SE\ CDC)$
  Then $requiresMoreMemory(J2SE\ CLDC)$
  As *requiresMoreMemory* is transitive, this is used to derive that J2SE requires more memory than CLDC. Here CDC, CLDC and J2SE are instances of its corresponding class.

- Formally regulated component composition. The rigorous run time configuration is conducted with LimboConfiguration ontology. Invalid configuration violating the built-in rules will be reported at run time by a backend reasoning tool, in our case, RacerPro.

4.3.1 Getting device information

With the help of an ontology frontend, the Limbo compiler can obtain general device information, such as the device name, model name and device type from the Device ontology. The CPU and memory information is obtained from the Hardware ontology. Detailed information on operating system is obtained from the OperatingSystem ontology. And Java virtual machine, Java library information is retrieved from the Java ontology. Other software packages information is got from the SoftwarePlatform ontology for example third party software packages available for the device, for example SuperWaba.

4.3.2 Formal configuration with LimboConfiguration ontology

Limbo2 can have different backends and frontends. Not all the combinations of the frontend and backend in Limbo are valid. Either the SOAP frontend or REST frontend can be chosen at one run. There exist complex options for the backends. The parser generator backend can choose one from several different programming platforms, for example JavaSE, JavaME, .NET or LeJOS. Besides the parser generation, other generation work can also be conducted at the same time, for example the generation of client side stub and server side skeleton, and transport between the client side and a server side.

We define a *configuration* for Limbo as the valid combinations of frontend, repository and backends that could be used to compile and generate correct web services according to the specific device information, such as CPU and operating system. A feature model(Czarnecki and Eisenecker, 2000) can be used to visually show the configurations of Limbo (Figure 4.8). But the feature model could not easily show some of the hidden dependencies between backends. For example, client side and server side (including server, skeleton, controller) generation, and also state machine generation should be consistent with the chosen parser language, i.e. they should use the same language in one Limbo compiling run. For OSGi, there is no need for the Server generator as a web server is built in the OSGi framework.

Therefore it is very important to rigorously regulate the valid combinations of different Limbo components and resolve dependencies among them, no matter combinations are

[2] LeJOS homepage. http://lejos.sourceforge.NET/

explicit in the feature model or implicit. As the Hydra project is using ontologies to share knowledge across all the software to be developed, this naturally motivates us to the exploration of using OWL to achieve validation and verification of these configurations. As Wang proposed in (Wang et al., 2005), we use OWL DL ontologies to formally specify what the legal feature combinations are.

The basic idea is: for every node in the feature diagram, to use a companion rule class to specify its regulations including how each of its child features are related to this node, as well as restrictions for its combination with all other features. The conjunction/disjunction of "someValuesFrom" restrictions over *hasFeatureX* (FeatureX stands for a feature node in the feature diagram, for example, *hasRESTFrontend*) object properties can be used to model different type of features. For example the "Alternative" feature for the WSDL frontend is specified with the following axioms:
$WSDLFrontendRule \sqsubseteq (\exists hasRESTFrontend.$
$RESTFrontend) \sqcup (\exists hasSOAPFrontend.SOAPFrontend)$
$WSDLFrontendRule \sqsubseteq \neg(\exists hasRESTFrontend.$
$RESTFrontend) \sqcap (\exists hasSOAPFrontend.SOAPFrontend))$

This means that either REST Frontend or SOAP frontend will be used, but not both. And coming to the OSGi controller as an example, we use the following to restrict its exclusive relationship with the .NET parser backend:
$OSGiControlerRule \sqsubseteq \neg(\exists hasDotNetParserBackend.$
$DotNetParserBackend)$

We can then use this ontology to verify whether a feature combination is valid or not. As an example in Figure 4.9, if both an OSGi controller and a .NET parser backend are chosen for a configuration:
$E\_Config \sqsubseteq \exists hasBackend.Backend$
$E\_Config \sqsubseteq \exists hasDotNetParserBackend.$
$DotNetParserBackend$
$E\_Config \sqsubseteq \exists hasFrontend.Frontend$
$E\_Config \sqsubseteq \exists hasOSGiControler.OSGiControler$
$E\_Config \sqsubseteq \exists hasRepository.Repository$

Then RacePro[3] will report that this configuration of Limbo is inconsistent, because the combination of the OSGi controller and the .NET backend contradicts to the rules in the Limbo configuration ontology which result in an invalid configuration.

Why SWRL rules

For situations where some complex rules are needed to specify certain restrictions, we are using SWRL[4] to develop these rules, and the SWRL APIs from Protege-owl[5] for the implementation.

"SWRL is intended to be the rule language of the Semantic Web. SWRL is based on the OWL Web Ontology Language. It allows users to write rules to reason about OWL individuals and to infer new knowledge about those individuals. A SWRL rule contains an antecedent part, which is referred to as the body, and a consequent part, which is referred to as the head. Both the body and head consist of positive conjunctions of atoms. SWRL does not support negated atoms or disjunction. Informally, a SWRL rule may be read as

[3]RacerPro homepage: http://www.racer-systems.com/

[4]SWRL: A Semantic Web Rule Language Combining OWL and RuleML. http://www.w3.org/Submission/SWRL/

[5]protege homepage. http://protege.stanford.edu/ and Protege-SWRL tab: http://protege.cim3.net/cgi-bin/wiki.pl?SWRLTab

meaning that if all the atoms in the antecedent are true, then the consequent must also be true."

SWRL is built on OWL DL and shares its formal semantics. It is more expressive than OWL DL alone. And all variables in SWRL rule bind only to known individuals in an ontology in order to develop a DL-Safe rules to make them decidable.

Configuration algorithm for Limbo

To be practical for Limbo to use the configuration techniques proposed in (Wang et al., 2005), we go a step further by importing the Device ontology in order to configure Limbo compiler according to the hardware and software details of a specific device. Object properties (*requireCPU*, *requireOS*, *requireVM* and *requireLibrary*) are also added to the configuration ontology in order to specify a backend's requirements for the detailed CPU, operating system, virtual machine and libraries. The Limbo configuration algorithm is shown as an UML activity diagram in Figure 4.10. We are not showing the handling of third party libraries which can be handled in the same way as explained later.

Step 1 Checking CPU/OS/VM details

When a compiling task is needed for a certain device, first the detailed software and hardware information, especially CPU, operating system, virtual machine will be retrieved using the ontology frontend. In SWRL rules, the symbol $\wedge$ means conjunction, and $?x$ stands for a variable, and symbol $\rightarrow$ means implication.

Take the Nokia N80 as an example, its operating system information is checked using the following SWRL query:
$device : MobilePhone(device : NokiaN80) \wedge device :$
$hasSoftware(device : NokiaN80, ?s) \wedge$
$SoftwarePlatform : hasVirtualMachine(?s, ?vm) \wedge$
$Java : hasMIDP2version(?vm, ?version) \rightarrow$
$sqwrl : select(device : NokiaN80, ?s, ?vm, ?version)$

Here "device" and "SoftwarePlatform" are namespaces for the Device ontology and the SoftwarePlatform ontology respectively. The execution of this query will give us these results:
$(?s = SoftwarePlatform : SoftwarePlatform\_N80)$ and $(?vm = Java : MIDP2\_2)$, $(?version = 2)$. Software platform instance for N80 is defined with *SoftwarePlatform_N80*, and N80 virtual machine is *MIDP2_2*, which means that it supports MIDP2 based on CLDC1.1 (*MIDP2_1* is defined as the MIDP2 based on CLDC1.0).

A general rule for checking both mobile phone CPU and virtual machine is as followed:
$device : MobilePhone(?device) \wedge device : hasHarware(?device, ?hardware) \wedge Hardware : primaryCPU(?hardware, ?cpu) \wedge Hardware : cpuName(?cpu, ?cpuname) \wedge device : hasSoftware(?device, ?soft) \wedge SoftwarePlatform : hasVirtualMachine(?soft, ?vm) \rightarrow sqwrl : select(?device, ?cpuname, ?vm)$

Step 2 Iteratively checking the backends' required CPU/OS/VM

After the detailed information on CPU, operating system and virtual machine has been obtained from related ontologies, then this information will be checked iteratively for

whether this version of CPU, operating system and virtual machine are required for the leaf backends in the feature diagram. This kind of information is stored within instances of the leaf backends associated with the requireCPU, requireOS and requireVM object properties. The main loop control variable is the list of all leaf backends in the feature model. An example for the N80 mobile phone is:
$JavaMEParserBackendRule(?r) \wedge requireVM(?r, ?vm)$
$\wedge\ Java : hasMIDP2version(?vm, ?version)$
$\rightarrow sqwrl : select(?r, ?vm, ?version)$

It returns $(?r = JavaMEParserBackendRule\_1)$ and $(?vm = Java : MIDP2.0)$, $(?version = 2)$. We define MIDP2.0 as the MIDP2 which can be based on CLDC1.0 or CLDC1.1. From these two queries, we can see that N80 needs a JavaMEParserBackend.

Step 3 Resolving choices using user preferences

There are situations where we can get multiple options for backends. For example, Motorola MPx220 has Windows Mobile as its operating system, but at the same time it has J2ME MIDP2_1 (MIDP 2.0, CLDC 1.0), which can be got from the following two query rules.
$device : MobilePhone(device : MotorolaMPx220) \wedge$
$device : hasSoftware(device : MotorolaMPx220, ?s) \wedge SoftwarePlatform : hasOperatingSystem(?s, ?os)$
$\rightarrow sqwrl : select(device : MotorolaMPx220, ?s, ?os)$

$device : MobilePhone(device : MotorolaMPx220) \wedge$
$device : hasSoftware(device : MotorolaMPx220, ?s) \wedge SoftwarePlatform : hasVirtualMachine(?s, ?vm) \rightarrow sqwrl : select(device : MotorolaMPx220, ?s, ?vm)$

Now we have enough information to compare with the end user group preferences. Then the generation can go ahead with the chosen Net CF or J2ME platform.

Step 4 Resolving choices based on CPU/Memory usage
We are using RacerPro as the backend ontology reasoning tool. Therefore, some of the reasoning are implemented with nRQL(new Racer Query Language). For situations where memory and CPU usage should be decided, for example J2SE, CDC and CLDC as options, we will choose the one that consumes less memory and requires a slower CPU for small devices as default. For example, we could retrieve all requireMoreMemory pairs with the following nRQL query.
$(retrieve\ (?x\ ?y)(?x\ ?y\ requireMoreMemory))$

And we could also retrieve only the inferred requireMoreMemory pair with the following nRQL query.
$(retrieve\ (?x\ ?y)(and(?x\ ?y\ requireMoreMemory)(neg$
$(and(?x\ ?y\ requireMoreMemory)(neg\ (project - to$
$(?x\ ?y)(and(?x\ ?z\ requireMoreMemory)$
$(?z\ ?y\ requireMoreMemory))))))))$

Here the requireMoreMemory should be the URI containing the requireMoreMemory object property in the SoftwarePlatform ontology and we deliberately show only requireMoreMemory itself in order to save space.

Step 5 Resolve options based on power/energy policy. These rules may decide whether to choose a proxy implementation or running embedded web service directly on device. The energy rules have big consequences in this respect. An example rule to check the battery level is like this:
$device : MobilePhone(?device) \wedge device : hasHarware(?device, ?hardware) \wedge Hardware : primaryBattery(?hardware, ?battery) \wedge Hardware : batteryLevel(?battery, ?level) \rightarrow sqwrl : select(?device, ?level)$

Similarly, we can check the power consumption of various bearers supported by a mobile phone, and choose a corresponding bearer according to the power consumption expectation. $device : MobilePhone(?device) \wedge device : hasHarware(?device, ?hardware) \wedge Hardware : supportedNetworkBearers(?hardware, ?bearers) \wedge Hardware : networkPowerUsed(?bearers, ?power) \rightarrow sqwrl : select(?device, ?bearers, ?power)$

4.4 State Machine

4.4.1 State machine ontology

A common conception of embedded devices is that they are usually designed and operated as state machines. In line with this, we make use of a state machine for a device to achieve a number of objectives.

- Generation of stub code for the state transitions.
- Specification of complex context rules and diagnosis rules based on states and running result of an action.
- Supervising the health status of a running system.

A failure can be caused by an earlier state transfer failure. Therefore some transitions and states can be used to explain the observed malfunctions and effectively identify faults in the device, which is of great value for helping the self-diagnosis and self-recovery.

4.4.2 Development of the state machine ontology

To achieve these goals, we develop a state machine ontology based on Dolog's work (Dolog, 2004). To accomplish the notification of state changes and actions execution results, a publish/subscribe event manager is used.

We show the state machine ontology for the thermometer (UML state machine shown in Figure 3.3) in Figure 4.11.

We changed Dolog's work in several aspects: first, we add a datatype property *isCurrent* in order to indicate whether a state is current or not with the publish/subscribe mechanism; secondly, we add a *doActivity* object property to the *State* in order to specify the corresponding activity in a state and this makes the state machine complete; third, we add a datatype property *hasResult* to the *Action* (including activity) concept in order to check the execution result at run time. The action results are also updated using the publish/subscribe mechanism just like the updating of states. This facilitates the specification of diagnosis rules based on both the state and activity results. We also add properties to store 3 historical results and their time stamp.

In order to keep track of the action result and make appropriate diagnosis, we have one state machine instance for every device. But the generated state machine stub should be suitable for every device, or one type of device. Therefore, we add a dummy state machine for every device type and use this generic state machine to generate code for this type of device, which shares this state machine. The generated state machine stub is as follows. The measuring state of the thermometer is linked to the action service:

```
package com.eu.hydra.flamenco.statemachine;
...

public class StateMachineStub_Thermometer{
    ...

    public void  ThermometerStopping(){
      event ev = new event();
      ev.parts_add(new part("DeviceType", "Thermometer"));
      ev.parts_add(new part("State", "ThermometerStopping"));
      ev.parts_add(new part("StateAction", "ThermometerStop"));
      ev.parts_add(new part("DeviceID", this.deviceID));
      eventManager.publish("statemachine/statechange", ev);
    }

    public void  ThermometerStarting(){
      ...
    }

    public void  ThermometerMeauring(){
      ...
    }

    ...
}
```

4.4.3 SWRL rules for diagnosis and complex context specification

We are using SWRL together with the state machine ontology to achieve the former mentioned objectives. For example, rules with SWRL for checking device status and diagonosis can be written as followed.

- Checking how many states an instance of the state machine for a device has:
  $StateMachine(?sm) \wedge hasStates(?sm, ?s) \rightarrow sqwrl : select(?sm) \wedge sqwrl : count(?s)$
- Checking the current state that *isCurrent* is set to ”true”.
  $StateMachine(?sm) \wedge hasStates(?sm, ?s) \wedge$
  $isCurrent(?s, "true") \rightarrow sqwrl : select(?sm, ?s)$
- Diagnosis on the fly of the device working condition. For example, we can check the *validImage* action result in order to decide the working status of the camera with the following rule:
  $StateMachine(?sm) \wedge hasStates(?sm, ?s) \wedge$

$doActivity(?s, ?ac) \wedge actionResult(?ac, ?r) \rightarrow$
$sqwrl : select(?sm, ?s, ?ac, ?r)$

If the returned image is valid, then the camer is working well. This checking can only be executed when *processingImage* state is the current state.

The SWRL provides builtin math, string, comparisons that can be used to specify extra contexts, which are not possible or very hard to achieve by the present OWL itself. For example, we can specify a *farAwayFromHome* context (e.g. 5 miles away from home using the GPS distance calculation formula [6]) with the following two rules:
$distanceFromHome(?p, ?distance) \wedge$
$swrlb : greaterThan(5, ?distance)$
$\rightarrow farAwayFromHome(?p, true)$ ...(1)

$person : Person(?p) \wedge hasHome(?p, ?h) \wedge$
$hasLocation(?h, coord : LocationCoords_1) \wedge$
$coord : latitude(coord : LocationCoords_2, ?lan2) \wedge coord : latitude(coord : LocationCoords_1, ?lan1) \wedge$
$swrlb : subtract(?rsublan, ?lan1, ?lan2) \wedge swrlb : multiply(?squaresublan, ?rsublan, ?rsublan) \wedge$
$coord : longitude(coord : LocationCoords_2, ?long2) \wedge coord : longitude(coord : LocationCoords_1, ?long1) \wedge$
$swrlb : subtract(?rsublong, ?long1, ?long2) \wedge swrlb : multiply(?squaresublong, ?rsublong, ?rsublong) \wedge$

$swrlb : add(?add, ?squaresublong, ?squaresublan) \wedge$
$swrlm : sqrt(?distance, ?add)$
$\rightarrow distanceFromHome(?p, ?distance)$ ...(2)

As the current SWRL APIs are not supporting OWL description language expressions as SWRL specification promised, and it can not work on the inferred model, we will extend our work after the new SWRL APIs released which will support all those missing functionalities and ease the design of general rules at the class level. We are also working together with the APIs developer to improve the stability of these APIs.

This part of work will be elaborated in D4.3.

4.5 Ontology tools experience

We are using a number of ontology related tools in WP4.

4.5.1 Ontology development tools

For the development of ontologies, we are Protege3 and Protege4, together with XMLSpy in order to sort out some problems for example hidden datatype redefinitions. Protege 3 is becoming cumbersome because its protege-owl is based on the Frame implementation. However, Protege 3 has many plugins and is the most popular tool for ontology development. Furthermore, it has good user support through its mailing list. Protege 4 has a better architecture but is not functionally complete. XMLSpy can be used to filter out some XML-related editing and validation which is not conveniently provided by Protege.

[6] How to calculate the distance between two points on the Earth. http://www.meridianworlddata.com/Distance-Calculation.asp

4.5.2 Ontology reasoning tools

RacerPro which is free for academic usage, with built-in support for Location relationship reasoning; Pellet, open source, and builtin support from Protege4 without the need of DIG. The DIG interface has many problems at present stage and there seem to be no promises for a new version in the near future.

4.5.3 Ontology programming APIs

We are using the Jena, Protege-owl and SWRL APIs in WP4. Protege-owl is chosen because it is necessary for SWRL, and Jena is used by Protege as its backend for ontology loading and saving.

4.5.4 Rule engine

Jess is providing education license although it is not open source. Jess is used because SWRL currently needs it to translate SWRL rules back and forth to Jess rules.

4.5.5 Tool experiences

- Too often Protege (v3.3 or before, even v3.4) will throw an error and the work is not able to be saved anymore.
- Too often the value for a string can not be saved, and one has to use Protege 4 or a text editor to finish this input.
- The DIG interface is very often not stable for large ontologies.

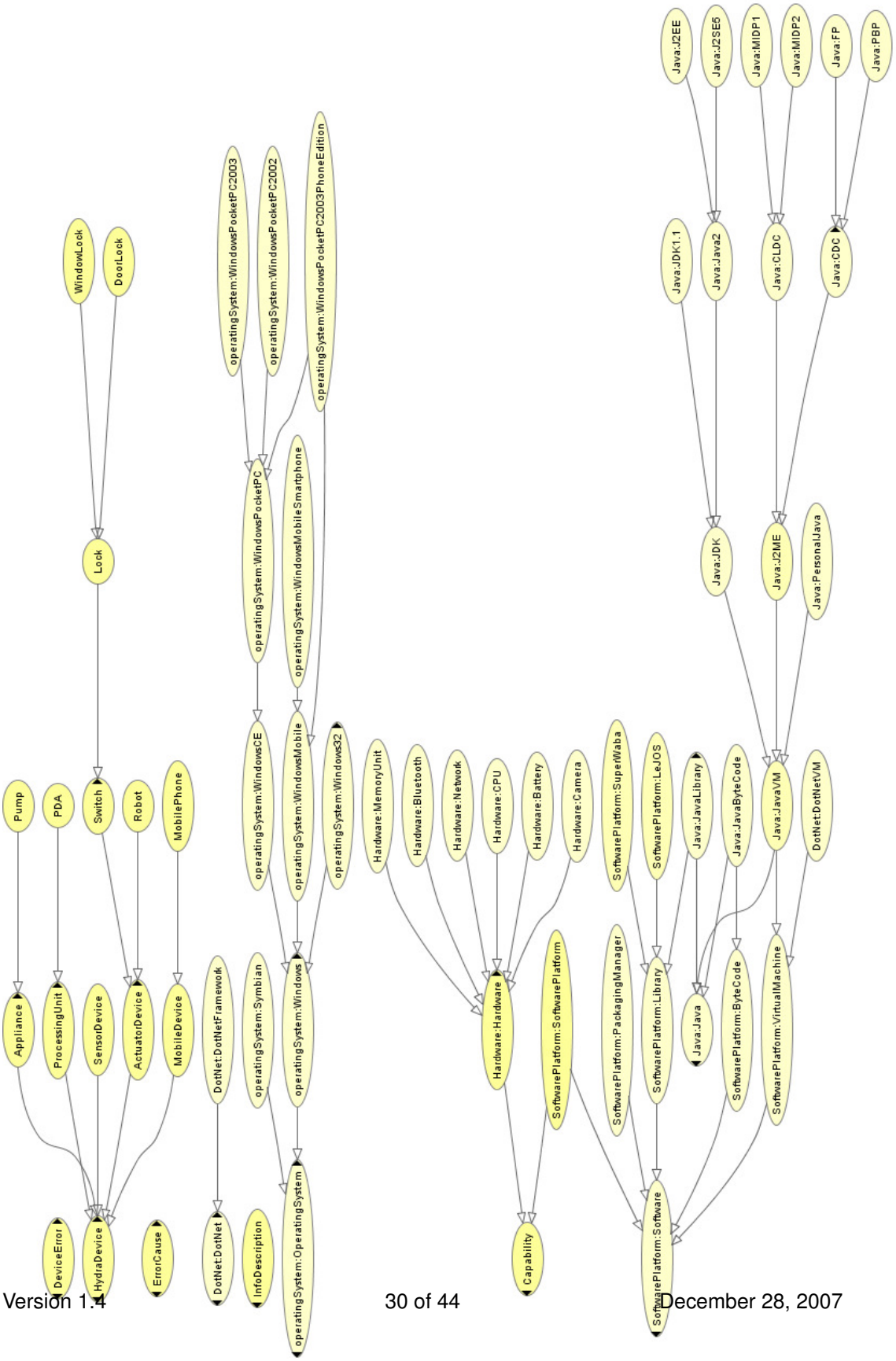

Figure 4.7: Device ontology details

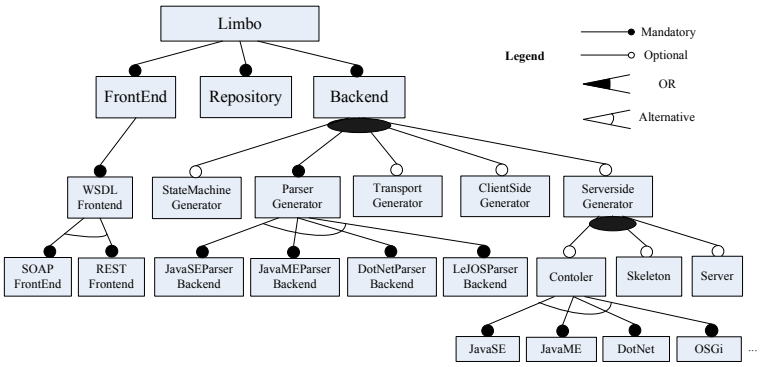

Figure 4.8: Feature model for Limbo

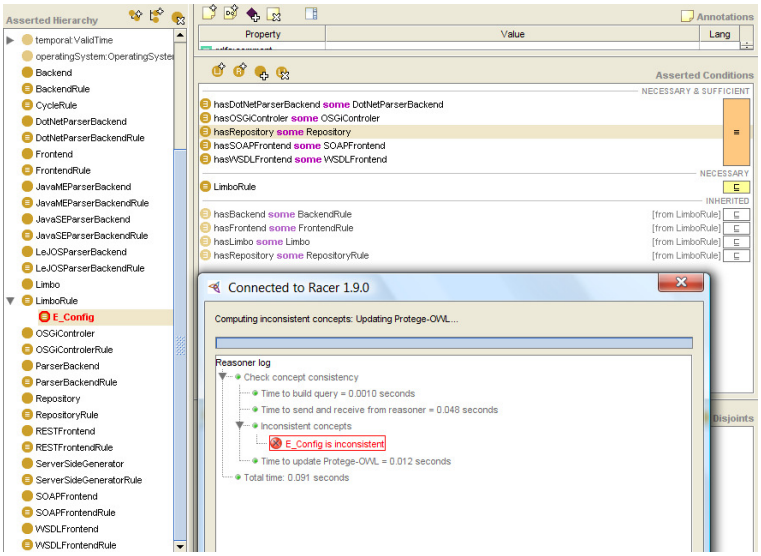

Figure 4.9: Limbo configuration reasoning

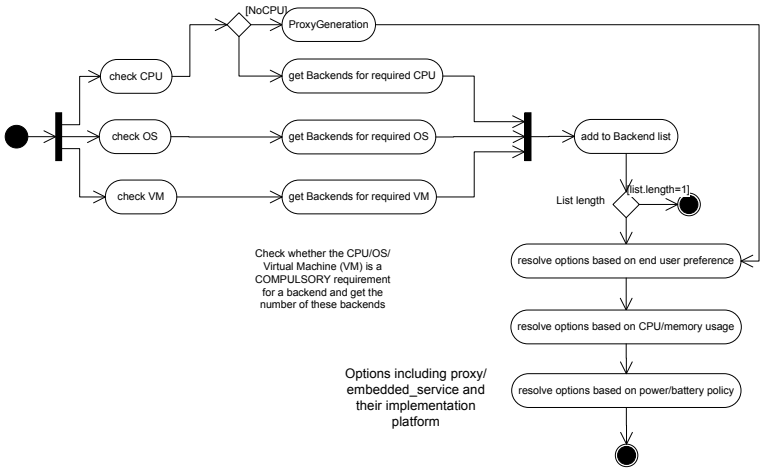

Figure 4.10: Limbo configuration algorithm

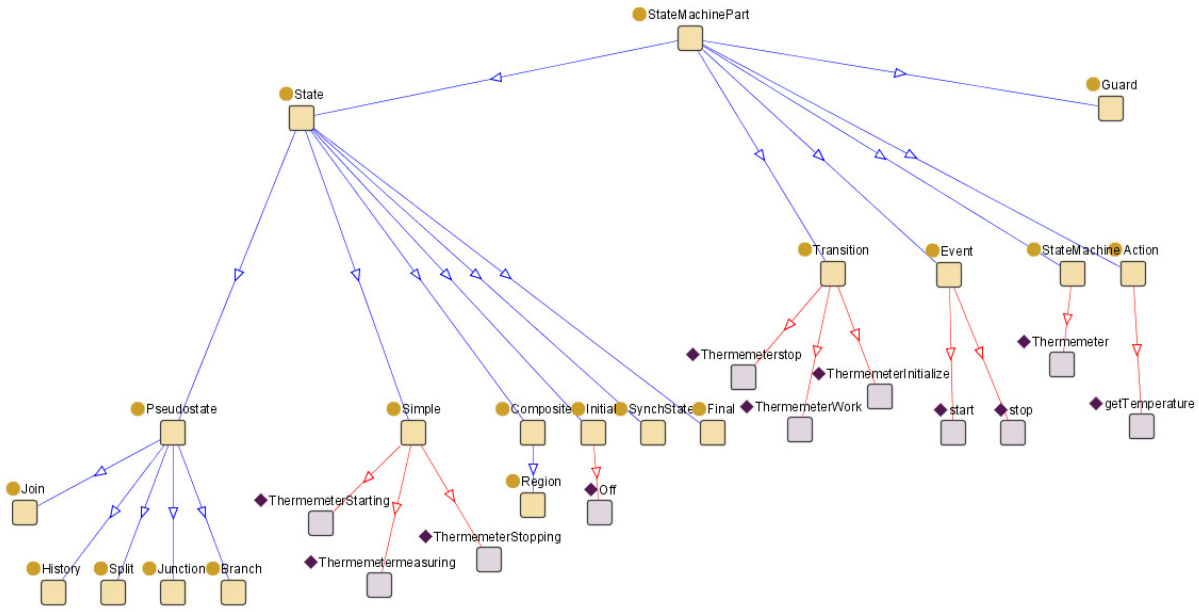

Figure 4.11: State machine Ontology

5 Evaluation of Limbo

We have evaluated the current version of Limbo according to the evaluation framework of one.world (**?**). This entails evaluating

1. *Completeness*: can useful services be generated?
2. *Performance*: is performance acceptable, i.e., in the Hydra case, are the generated services sufficiently ressource efficient?
3. *Complexity and utility*: how hard is it to create services and can others build upon it?

We now turn to each of these.

5.1 Completeness

Our main vehicle for evaluating "completeness" in the sense of whether useful services can be generated has been through the generation of services for the first Hydra prototype. Here services were primarily created by a member of the UAAR Hydra team who has not participated in the development of Limbo (4 services) and by a UAAR member of the Limbo compiler team (1 service). The generated services were:

- *Nokia N80 SMS service*. This service used the Nokia Raccoon gateway[1] (but not the Apache web server) for Nokia S60 mobile phones. The service uses Limbo's midlet generation option and runs a Limbo-generated web server
- *Pico TH03 thermometer service*. This and the following services run as proxies on an OSGi gateway and interface with devices via device-specific protocols
- *Grundfos Magna 32 pump service*
- *Abloy EL582 door lock service*

For all services, Hydra helped in hiding web service complexity and in generating efficient web services. State machine generation was not used in this case since it was not implemented. State Machine generation has been evaluated later with the Pico TH03 thermometer, see Deliverable 4.3 (Hansen and Zhang, 2007) for more detail.

5.2 Performance

The performance evaluation of Limbo has been targeted on the resource utilization of generated web services to that web services generated by other compilers. Here we report on time and memory usage measurements compared to Apache Axis[2]. The purpose was not to compare to Apache Axis per se since this was designed for a multithreaded server environment, but rather to see that Limbo-generated services used significantly fewer resources than a popular web service framework.

For measuring resource utilization, we used a setup with a SOAP-based web service implementing an SMS service. This web service was requested by a Limbo-generated

[1] Nokia Mobile Web Server. http://wiki.opensource.-nokia.com/projects/Mobile_Web_Server
[2] Apache Axis. http://ws.apache.org/axis

client and implemented using Apache Axis and using Limbo on both Java SE and Java ME (on a Nokia N80 mobile phone). For the Apache Axis and Limbo SE implementations a PC (an Apple Mac Book Pro 15" with a 2.33 GHz Intel Core 2 Duo processor, 2 GB 667 MHz DDR2 SDRAM memory, MAC OS X Version 10.4.10 as operating system). Communication was done over a local area network for the PC case and using Nokia's Raccoon software as a gateway to overcome firewall problems in the Nokia N80 case.

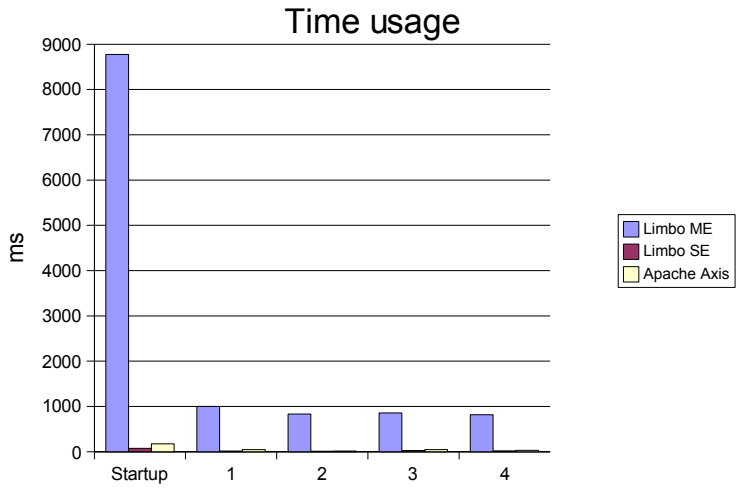

Figure 5.1: Limbo time measurements

Figure 5.1 shows the result of our time measurements. The figure shows the total execution time for five consecutive calls made to the SMS web service. For all implementations there is a high start-up cost due to the establishment of sockets – in particular so in the Java ME case. The Limbo ME implementation is also orders of magnitudes slower than the SE implementations, a fact that is due to the network setup of the Nokia N80 – and to the fact that the ME implementation actually sends an SMS – since the Limbo SE and Apache Axis implementations are comparable with respect to time usage.

Figure 5.2 shows the memory measurements of Limbo and Apache Axis. Both the Limbo SE and the Limbo ME versions use significantly less memory than Apache Axis. In the SE cases, the measurements were made using a JMX agent to measure the maximum amount of memory used during processing of requests excluding the Java environment itself. In the ME case, we measured maximum memory with SUN's Wireless Tool Kit (Version 2.5) so that the service ran in an emulator for the memory measurements. On average, the Limbo ME service used 362.4 Kb memory. In conclusion, the resource usage of Limbo generated services is significantly smaller than that for Apache Axis-generated services.

5.3 Complexity and utility

Complexity and utility were evaluated by members of the Hydra consortium that had not participated in the development of Limbo. Two partial evaluations were made:

1. Evaluation of ontology construction

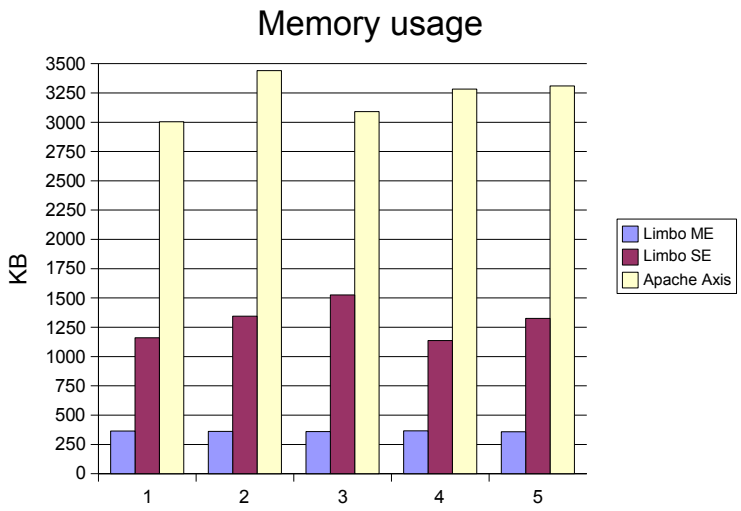

Figure 5.2: Limbo memory measurements

2. Evaluation of code generation

For both, a case of implementing a blood pressure service on an HTC P3300 smartphone[3] was executed. A blood pressure measurement device is imagined to be connected to the smartphone. In the concrete evaluation, we focussed on the smartphone and a blood pressure measurement device was not connected.

In Java-like pseudocode, the intended service would look like the following:

```
public interface SmartphoneBPService {
   String getSmartphoneBP(String monitorId);
}
```

The associated WSDL file is shown in Appendix A. The associated state machine is shown in Figure 5.3.

5.3.1 Complexity and utility of ontology construction

It was possible to create a model of the HTC P3300 device including a state machine within a day of work for an ontology engineer unfamiliar with the device and the associated service. The ontology was created using the Topbraid Composer ontology tool. The additions to the Hydra device ontology are shown in Appendix B. An overview is shown in Figure 5.4 on page 38.

Generally, the ontologies for software and hardware device properties need to be very well documented. The model is highly technically specific. For a developer, who is not familiar with the domain area, it is very hard (or in some cases even impossible) to create suitable representation of device properties, without precise specification of meaning of the concepts. Thus these models need to be created by someone who is both familiar with ontologies and the specific device. Possibly tool support (e.g., through a specific IDE for this) could be used.

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

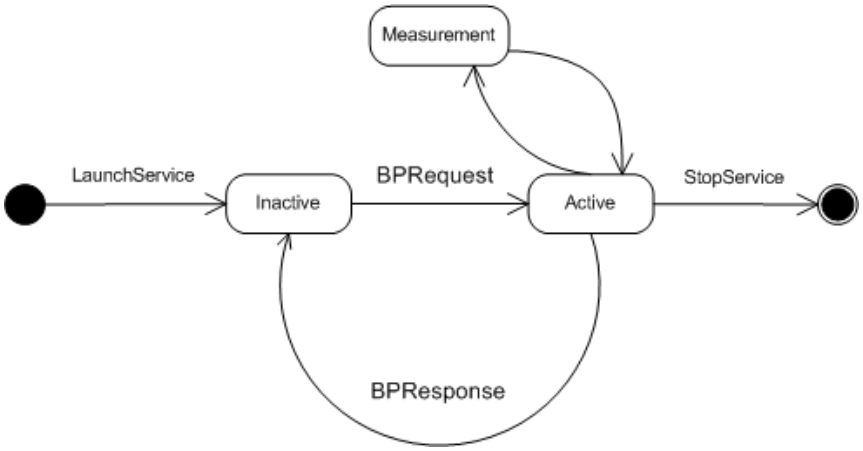

Figure 5.3: Blood pressure service state machine

The comments, which have arisen from the modelling process of the HTC P3300 device, can be summarized as follows:

- *Better classification is needed of some concepts*. It was, e.g., needed to create the new operating system (OS) instance for Windows Mobile 5.0 PocketPC. The attempt to classify the new operating system to the existing OS hierarchy showed that with the actual OS hierarchy (concretelly for WindowsMobile) it was hard to decide whether to choose some existing OS class or if it is better to create the new OS class. Finally, the new OS class was created (as the existing OS concepts seemed not be suitable) with the new OS instance. It would be helpful to create an OS hierarchy reflecting, e.g., the types of OSs, which could contain instances representing the versions of specific OSs. An OS hierarchy should maybe also reflect the various hardware platforms, on which the OSs can run. The suitable classifications can be developed with the help of experts
- *High-quality documentation of concepts is needed.* The range of some properties, e.g., require better comments of usage. For example, properties displayHeight and displayWidth have range defined as xsd:string. Thus, the developer is not able to decide, if the value should be e.g. 100 or 100px. As far as this information could be potentionally important, for example, to make decisions of device capability to display images (or other media) of several size (or resolution), usage of such properties should be well defined
- *Parts of the model is conceptually confusing*. For example, the hierarchy of concepts in the StateMachine ontology contains the formulation that every concept, including the StateMachine class, is the subclass of StateMachinePart concept. Thus, it can be inferred, for example, that StateMachine is the StateMachinePart.

This will be taken into account in the next iteration of developing the Limbo tool.

5.3.2 Complexity and utility of code generation

The Limbo compiler has been successfully used to generate small applications to test their compatibility with Windows Mobile-Based Smartphone and EclipseME-generated classes.

The next step will be using Limbo in specific applications for the second iteration prototype.

The following comments have arisen through the code generation process:

- Input indications are required in some sections. There are different classes in which the developer must enter new lines or modify the existing ones to create his own midlets, it would be very useful to at least mark them for a faster application development.
- Code comments are needed. The generated classes could include a more detailed comments structure. In this way it will be easier for the developer to understand each step and the whole application
- JAD and JAR manifest configuration assistance would be good. There are different versions of specific JME components which can make a midlet work or not work in different devices. For example, MIDP2.1 does not work with current HTC P3300 Virtual Machine, so a change in the manifest or the virtual machine must be made. Knowing what exactly works in which environment makes code generation faster.
- EclipseME plugin and Limbo integration would be good. This is not a high priority requirement, but it would be very useful to integrate the JME Eclipseme plugin and Limbo-based class generation in next versions.

5.4 Evaluation conclusions

The Limbo compiler has been shown to be useful, performant and of utility in the contexts in which it has been evaluated. Clearly there is a need for future work based on the evaluation:

- Better documentation is needed both for the compiler per se but also for the ontologies used in the compiler
- Further work is needed on the structure and the concepts of the ontology
- The structure of generated code could be improved and further documented

Further, combined with the requirements and design goals for Limbo, additional work is required in the area of

- Fully utilizing ontologies and reasoning at compile time
- Implementing the Limbo2 architecture
- Further compilation targets including .NET and REST-based targets
- Improved support for the Hydra middleware architecture including the UPnP-based discovery algorithm

Overall we believe that the first version of Limbo is a step in a right direction towards facilitating the development of embedded web service.

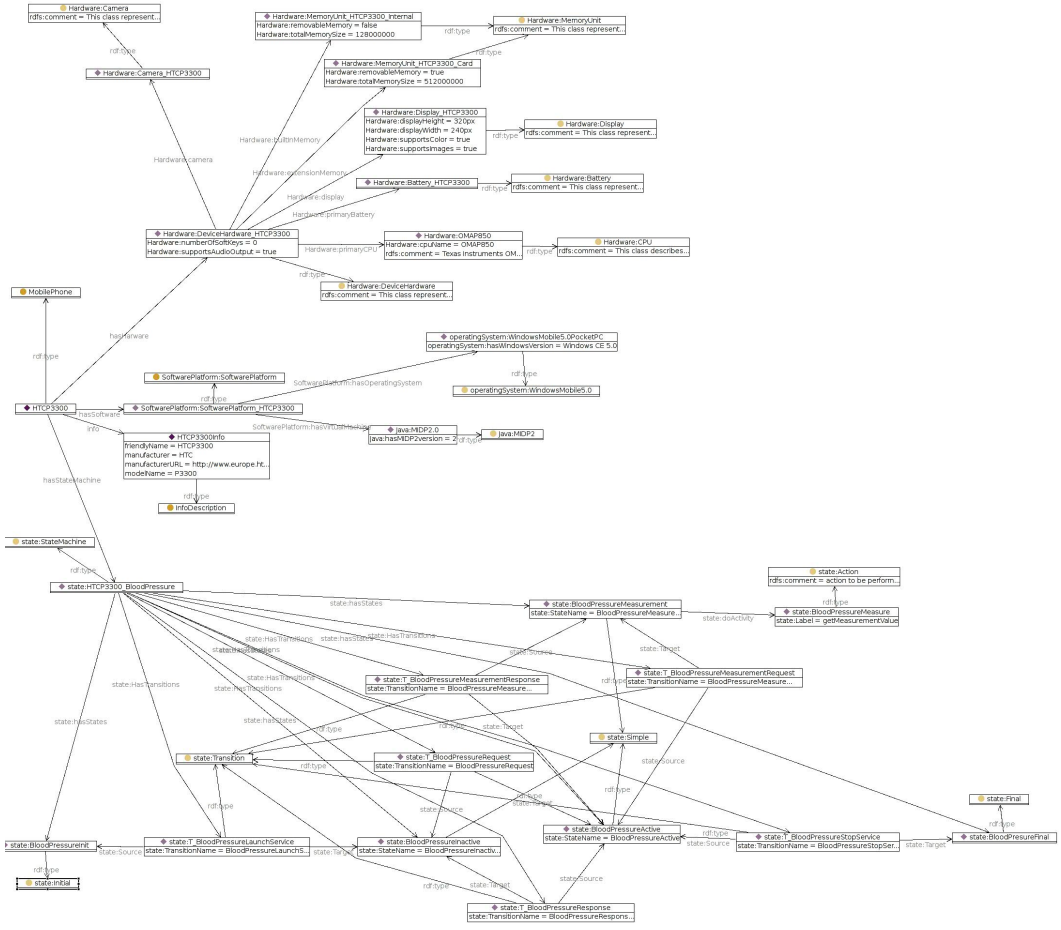

Figure 5.4: HTC P3300 ontology overview

6 Conclusion and future work

This deliverable is a companion to the first version of the Hydra Limbo tool. We have presented the design goals and architecture of Limbo as an important part of the Hydra SDK. On the other hand, the deliverable has focused on ontologies and shown how the usage of ontologies can add value to Limbo in the terms of both configuration and runtime semantic support.

Furthermore, the deliverable has presented results from evaluating Limbo in several ways (by building a range of services, by evaluating performance, and by evaluating the usage of ontologies). The evaluations have proved that the design of Limbo is successful, and Limbo is useful for supporting the development of (embedded) services based on the Hydra middleware.

For the next iteration on Limbo (due month 24), there are a number of directions we are pursuing:

- Implementing the Limbo2 architecture and its related semantic reasoning
- Further alignment of Limbo-generated services with the Hydra middleware including the Hydra discovery protocol, network architecture, and semantics approach when decided
- Investigating the generation of services for more platforms (as needed by the Hydra prototypes). This pertains to both hardware platforms (currently, services are created only for hardware platforms that support Java SE and Java ME) and software platforms (such as Microsoft .NET)
- Investigating the usage of WS-*-compatible interfaces in the Limbo-generated services. This is the road that, e.g., Devices Profile for Web Services [1]

[1] `http://schemas.xmlsoap.org/ws/2006/02/devprof/`

A HTC P3300 Blood Pressure service WSDL file

```
<?xml version="1.0" encoding="utf-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
   xmlns:hydra="http://hydra.eu.com"
   targetNamespace="http://hydra.eu.com">

<hydra:message name="sendBPRequest">
   <part name="monitorID" type="xs:string"/>
</hydra:message>

<message name="sendBPResponse">
   <part name="BloodPressure" type="xs:string"/>
</message>

<portType name="BPServicePort">
   <operation name="getSmartphoneBP">
      <input message="hydra:sendBPRequest"/>
      <output message="hydra:sendBPResponse"/>
   </operation>
</portType>

<binding name="BPServiceBinding" type="hydra:BPServicePort">
   <soap:binding style="rpc"
                 transport="http://schemas.xmlsoap.org/soap/http"/>
   <operation name="getSmartphoneBP">
      <soap:operation soapAction="http://hydra.eu.com/SmartPhone/
          getSmartphoneBP" style="rpc"/>
      <input>
         <soap:body use="literal"/>
      </input>
      <output>
         <soap:body use="literal"/>
      </output>
   </operation>
</binding>

<service name="SmartphoneBPService">
   <port name="BPServicePort" binding="hydra:BPServiceBinding">
      <soap:address location="http://hydra.eu.com/SmartPhone"/>
   </port>
</service>
</definitions>
```

B HTC P3300 Ontologies

B.1 Device ontology

```
<InfoDescription rdf:ID="HTCP3300Info">
  <modelName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
     P3300
  </modelName>
  <manufacturerURL rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
     http://www.europe.htc.com/en/products/htcp3300.html
  </manufacturerURL>
  <manufacturer rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
     HTC
  </manufacturer>
  <friendlyName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
     HTCP3300
  </friendlyName>
</InfoDescription>

<MobilePhone rdf:ID="HTCP3300">
  <hasStateMachine
     rdf:resource="file:./resources/StateMachine.owl#HTCP3300_BloodPressure"/>
  <hasHarware
     rdf:resource="file:./resources/Hardware.owl#DeviceHardware_HTCP3300"/>
  <hasSoftware
     rdf:resource="file:./resources/SoftwarePlatform.owl#SoftwarePlatform_HTCP3300"/>
  <info rdf:resource="#HTCP3300Info"/>
</MobilePhone>
```

B.2 State Machine ontology

```
<StateMachine rdf:ID="HTCP3300_BloodPressure">
   <hasStates rdf:resource="#BloodPressureInactive"/>
   <HasTransitions
      rdf:resource="#T_BloodPressureMeasurementResponse"/>
   <hasStates>
      <Initial rdf:ID="BloodPressureInit"/>
   </hasStates>
   <HasTransitions>
      <Transition rdf:ID="T_BloodPressureMeasurementRequest">
        <Target rdf:resource="#BloodPressureMeasurement"/>
        <Source rdf:resource="#BloodPressureActive"/>
        <TransitionName
           rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >BloodPressureMeasurementRequest</TransitionName>
      </Transition>
   </HasTransitions>
   <HasTransitions rdf:resource="#T_BloodPressureResponse"/>
   <hasStates rdf:resource="#BloodPressureMeasurement"/>
   <HasTransitions rdf:resource="#T_BloodPressureRequest"/>
   <hasStates rdf:resource="#BloodPressureActive"/>
```

```
    <HasTransitions>
      <Transition rdf:ID="T_BloodPressureLaunchService">
        <TransitionName
           rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >BloodPressureLaunchService</TransitionName>
        <Target rdf:resource="#BloodPressureInactive"/>
        <Source rdf:resource="#BloodPressureInit"/>
      </Transition>
    </HasTransitions>
    <HasTransitions rdf:resource="#T_BloodPressureStopService"/>
    <hasStates rdf:resource="#BloodPresureFinal"/>
</StateMachine>
```

B.3 Hardware ontology

```
<DeviceHardware rdf:ID="DeviceHardware_HTCP3300">
    <extensionMemory>
      <MemoryUnit rdf:ID="MemoryUnit_HTCP3300_Card">
        <totalMemorySize rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >512000000</totalMemorySize>
        <removableMemory rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean"
        >true</removableMemory>
      </MemoryUnit>
    </extensionMemory>
    <builtInMemory>
      <MemoryUnit rdf:ID="MemoryUnit_HTCP3300_Internal">
        <totalMemorySize rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >128000000</totalMemorySize>
        <removableMemory rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean"
        >false</removableMemory>
      </MemoryUnit>
    </builtInMemory>
    <display>
      <Display rdf:ID="Display_HTCP3300">
        <supportsImages rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean"
        >true</supportsImages>
        <supportsColor rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean"
        >true</supportsColor>
        <displayWidth rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >240px</displayWidth>
        <displayHeight rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >320px</displayHeight>
      </Display>
    </display>
    <camera>
      <Camera rdf:ID="Camera_HTCP3300"/>
    </camera>
    <primaryBattery>
      <Battery rdf:ID="Battery_HTCP3300"/>
    </primaryBattery>
    <primaryCPU>
      <CPU rdf:ID="OMAP850">
        <rdfs:comment>Texas Instruments OMAP850 200 MHz processor</rdfs:comment>
```

```
        <cpuName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >OMAP850</cpuName>
      </CPU>
    </primaryCPU>
    <supportsAudioOutput rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean"
    >true</supportsAudioOutput>
    <numberOfSoftKeys rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
    >0</numberOfSoftKeys>
</DeviceHardware>
```

B.4 Software platform ontology

```
<SoftwarePlatform rdf:ID="SoftwarePlatform_HTCP3300">
   <hasVirtualMachine rdf:resource="file:./resources/Java.owl#MIDP2.0"/>
   <hasOperatingSystem
       rdf:resource="file:./resources/OperatingSystem.owl#WindowsMobile5.0PocketPC"/>
</SoftwarePlatform>
```

Bibliography

Czarnecki, K. and Eisenecker, U. (2000). *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley. NY, USA.

Dolog, P. (2004). Model-driven navigation design for semantic web applications with the umlguide. *Engineering Advanced Web Applications, In Maristella Matera and Sara Comai (eds.)*.

Hansen, K. M. and Zhang, W. (2007). Self-* properties SDK prototype and report. Technical Report D4.3, Hydra Consortium. IST 2005-034891.

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

Murata, M., Lee, D., Mani, M., and Kawaguchi, K. (2005). Taxonomy of XML schema languages using formal language theory. *ACM Transactions on Internet Technology (TOIT)*, 5(4):660–704.

Nardi, D. and Brachman, R. (2003). The Description Logic Handbook-Theory, Implementation and applications.

Wagelaar, D. and Jonckers, V. (2005). Explicit platform models for mda. In *8th International Conference on Model Driven Engineering Languages and Systems*, pages 367–381, Montego Bay, Jamaica. LNCS 3713, Springer-Verlag.

Wang, H., Li, Y. F., Sun, J., Zhang, H., and Pan, J. (2005). A Semantic Web Approach to Feature Modeling and Verification. In *1st Workshop on Semantic Web Enabled Software Engineering (SWESE'05)*, Galway, Ireland. LNCS, Springer-Verlag.