Ontology-Enabled Generation of Embedded Web Services

Klaus Marius Hansen and Weishan Zhang and Goncalo Soares
Department of Computer Science, University of Aarhus
Aabogade 34, 8200 Århus N, Denmark
{klaus.m.hansen,zhangws,afonso}@daimi.au.dk

Abstract

Web services are increasingly adopted as a service provision mechanism in pervasive computing environments. Implementing web services on networked, embedded devices raises a number of challenges, for example efficiency of web services, handling of variability and dependencies of hardware and software platforms, and of device state and context changes. To address these challenges, we developed a web service compiler, Limbo*, in which Web Ontology Language (OWL) ontologies are used to make the Limbo compiler aware of its compilation context, such as targeted hardware and software. At the same time, knowledge on device details, platform dependencies, and resource/power consumption is built into the supporting ontologies, which are used to configure Limbo for generating resource efficient web service code. A state machine ontology is used to generate stub code to facilitate handling of state changes of a device. A number of evaluations show that the design of the Limbo compiler is successful in terms of performance of the generated web service, completeness in being applicable to a variety of embedded devices, and usability for developers in creating new services.*

1 Motivation and introduction

Pervasive computing is becoming a reality. Because of their increasing ubiquity in business environments, web services are increasingly needed to be adopted as service provision mechanisms in pervasive computing environment. Consequently, in a number of applications, web services are deployed on resource-constrained embedded and networked devices. Implementing web services on embedded devices raises a number of challenges. First, embedded devices are constrained in memory, processor and energy resources. The web services should be sufficiently resource efficient in order to provide usable services. Second, development environments for embedded web services must be able to handle the variability of hardware and software, power supply, and possible dependencies between platform properties. At the same time, pervasive computing environments are highly dynamic, with, e.g., device statuses changing very often; something that affects end user applications.

A number of tools and approaches focusing on making web services available on small embedded platforms exist. One example is Microsoft's Web Services on Devices[1], and Fast Infoset[2]. Fast Infoset is not a web service technology per se, but provides a binary encoding of XML that may be used to make web services more efficient in the sense that they use less bandwidth in communication. These tools, however, fall short in the flexibility of code generation and complexity hiding of device details and web service details for the developer. At the same time, they lack the extensibility for using new protocols and technologies, when considering the huge variance of embedded and networked devices.

To address these issues, in this paper, we present *Limbo*, an ontology-enabled compiler for the generation of embedded web services. A number of Web Ontology Language (OWL[3]) ontologies are used to encode device details, platform dependencies, resource/power consumption, and valid Limbo components combinations, which are used to make the Limbo compiler aware its compilation context, such as the appropriate hardware and software for a given service. Runtime states of a device are handled with a state machine ontology and stub code is generated to support reporting device state changes.

The development of Limbo is part of a large, European research project, Hydra[4] that develops secure, service-oriented, and self-managed middleware for pervasive computing application scenarios.

The rest of the paper is structured as follows: in Section 2, we present the design and implementation of Limbo; followed by is the section on how to use the generated code

[1] http://www.microsoft.com/whdc/rally/Rallywsd.mspx
[2] https://fi.dev.java.net/
[3] http://www.w3.org/2004/OWL/
[4] http://www.hydra.eu.com/

for the development of web services. Section 3 discusses ontologies used in Limbo. In Section 4, we present the configuration algorithm used in Limbo. Then we evaluate the Limbo compiler in Section 5, from the perspective of complexity, usability and performance. We compare our work with related work in section 6. Conclusions and future work ends the paper.

2 Limbo design, implementation, and usage

2.1 Limbo design and implementation

Figure 1 shows the module structure of the Limbo compiler. The software architecture of Limbo follows the "Repository" architectural pattern [5] in which a central *Repository* stores data related to the transformation process and on which *Frontends* and *Backends* operate to read and write information. Frontends process source artifacts (in particular web service interface descriptions in the form of WSDL [5] files and ontology descriptions in the form of OWL files). Conversely, Backends produce target artefact's in the form of code (primarily state machine stubs, web service stubs and skeletons) and configuration files.

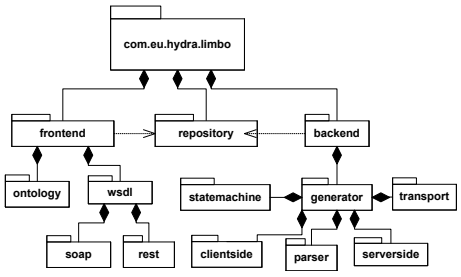

Figure 1. Module structure of Limbo

Backends implement different features. An essential feature is the parser backends with different implementation languages such as Java SE and Java ME (Java Standard Edition/Java Micro Edition). An example of generation can be the generation of client-side stubs and/or server-side skeletons or 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 generates state machine stubs. Figure 2 shows the compilation process of the Limbo compiler. 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:

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

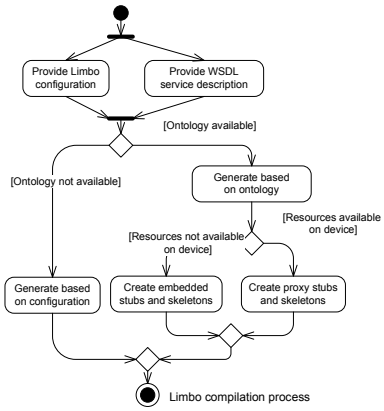

Figure 2. Limbo compiling process

- *Provide WSDL service description*: The main input for Limbo is WSDL file, and Limbo also supports that WSDL files references the Hydra device ontology. An example of a Hydra ontology binding for the thermometer in WSDL would be the following:

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

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

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

- *Create embedded/proxy stubs and skeletons.* Stubs and skeletons for the device service are created according to the device's capabilities. If code cannot be directly embedded on the device, proxy code is generated based that will run on OSGi[6]. For the thermometer, as it does not have any computing capability itself, according to the the retrieved platform information from the ontology, proxy code will be generated using OSGi.

2.2 Implementing services based on generated code

For the thermometer with a configuration of a standalone server using Java SE, the following classes are generated:

[6]`http://www.osgi.org`

- *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* - A service class that handles requests and returns the respective results
- *LimboServer.java* - Limbo server main class
- *StateMachineStub_Thermometer.java* - State machine stub for thermometer

An OSGi configuration (Java SE) or a Java ME server can be chosen, and Limbo can also generate clients either for Java ME or Java SE. The generation of OSGi code is following the OSGi specification (e.g., an Activator instead of an EndPoint, a servlet instead of a "th03Service"). Classes are also generated to support this in the form of a state machine stub that will allow the service developer to model and notify upon state changes. The following code is generated for the thermometer state machine shown in Figure 3, and the measuring state of the thermometer is linked to the getTemperature service.

```
public class StateMachineStub_Thermometer {
 public void  ThermometerStopping() {  ...  }
 public void  ThermometerStarting(){  ...  }
 public void  ThermometerMeauring(){
  event ev = new event();
  ...
  ev.parts_add(new part("Result",
    "" + service.getTemperature(this.deviceID)));
  eventManager.publish("/statemachine/statechange", ev);
 }
 ...
}
```

Based on the generated artifacts, 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 include, e.g., creating a thread that continuously calculates the temperature 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 state machine 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 the event manager (Figure 3), when the thermometer is started, when it is measuring, and when it stops as shown in the thermometer state machine in the lower part of Figure 3.
- *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 upper 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. Thermometer state changes trigger events sent through a publish/subscribe mechanism.

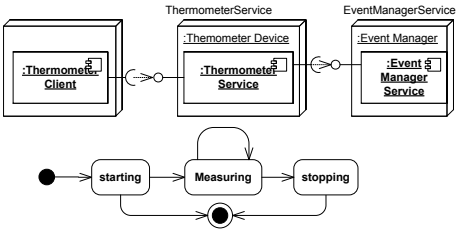

Figure 3. Thermometer runtime and thermometer state machine

3 Ontologies in Limbo

There are a number of reasons for us to use ontologies in Limbo: first, details of device hardware and software, and possible dependencies between them, are hidden 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. Second, in order to generate resource-efficient code, knowledge on device software platform and resource/power consumption comparisons are built into the related ontologies, and used during the configuration of Limbo for code generation.

We have developed the supporting ontologies for Limbo as shown in Figure 4. The usage of these ontologies can be summarized as follows:

- *LimboConfiguation ontology*. Not all the combinations of the frontends and backends in Limbo are valid. For example, for OSGi, there is no need for the Server generator as a web server is built into OSGi frameworks. Therefore it is very important to regulate the valid combinations of different Limbo components and resolve dependencies among them, whether combinations are explicit in the feature model or implicit. As proposed in [6], we develop a LimboConfiguration ontology to formally specify what the legal feature combinations are.
- *Device ontology and associated hardware platform and software platform ontologies*. These ontologies are used to retrieve device specific information in order to generate resource/power-awareness code for a certain device. The Device ontology is used to define high

level only information of a device, for example device type classification (e.g., an alarm device is a sensor).

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

The SoftwarePlatform ontology defines VirtualMachine, Middleware and object properties such as *requiresMoreMemory*, *reuqiresFasterCPU*, and their reverse properties. In the Java ontology we define concepts such as JavaVM, JavaByteCode and specify that a specific Java platform (e.g., CLDC) provides a certain Library or Rendering Engine etc.

The OperatingSystem ontology provides a classification of an operating system based on its characteristics and version for example Win32/Win16, which can facilitate the restrictions on which operating system consumes more memory than others.

- *StateMachine ontology*.

  For every type of device in the Device ontology, there is a corresponding state machine instance in the StateMachine ontology. This state machine instance is used to generate state machine stubs.

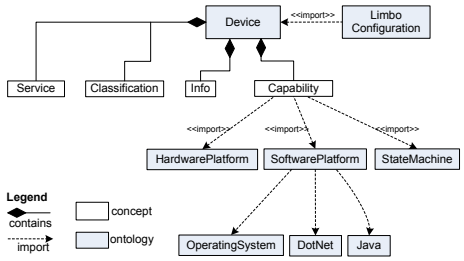

Figure 4. Structure of ontologies used in Limbo

4 Limbo configurations with ontologies

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. The Limbo configuration algorithm is shown as a UML activity diagram in Figure 5 and described next.

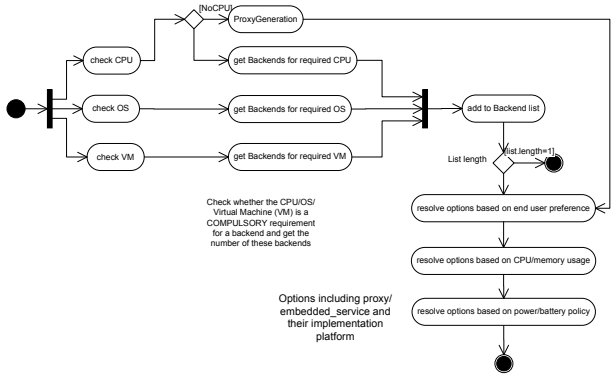

Figure 5. Limbo configuration algorithm

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.

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, this information will be checked iteratively for whether this version of CPU, operating system and virtual machine are required for the backends. This kind of information is stored within instances of the backends associated with the *requireCPU*, *requireOS*, *requireVM* object properties.

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, which will be compared with end user preferences. Then the generation can go ahead with the chosen platform.

Step 4. Resolving choices based on CPU/Memory usage 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.

Step 5. Resolve options based on power/energy policy The power consumption of various bearers supported by a device is checked, and choose a corresponding bearer according to the power consumption expectation.

In our implementation of the above algorithm, we are using SWRL[7] to resolve options for multiple platforms as detailed in [2].

5 Evaluation of Limbo

We have evaluated Limbo according to the evaluation framework of one.world [1]. 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.

5.1 Completeness

We evaluate this through the generation of services for a set of prototypes for a home automation scenario to testify whether useful services can be generated by Limbo. Here services were primarily created by a member of the Hydra team who has not participated in the development of Limbo (four services) and by a member of the Limbo compiler team (one service). For all services, Hydra helped in hiding web service complexity and in generating efficient web services. The generated services were:

- *Nokia N80 SMS service*. The service uses Limbo's Midlet generation option and runs a Limbo-generated web server.
- *Pico TH03 thermometer service*, *Grundfos Magna 32 pump service* and *Abloy EL582 door lock service*. These services run as proxies on an OSGi gateway and interface with devices via device-specific protocols.

5.2 Performance

Here we report on time and memory usage measurements compared to Apache Axis[8]. The purpose is not to compare to Apache Axis per se since it was designed for a multi-threaded 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 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 with a 2.33 GHz Intel Core 2 Duo processor, 2 GB DDR2 SDRAM, MAC OS X 10.4.10). The left part of Figure 6 shows the result

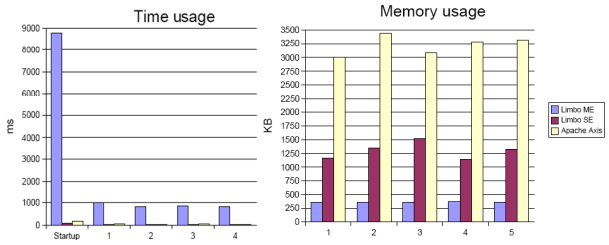

Figure 6. Limbo time and memory usage measurements

of our time measurements with 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.

The right part of Figure 6 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. In the ME case, we measured maximum memory with SUN's Wireless Tool Kit (Version 2.5). 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 project that had not participated in the development of Limbo. Two partial evaluations were made on evaluation of ontology construction and code generation. For both, a case of implementing a blood pressure service on an HTCP3300 smartphone[9] was used.

It was possible to create a model of the HTCP3300 device including a state machine within a day of work for an ontology engineer unfamiliar with the device and the associated service. The Limbo compiler has been successfully used to generate small applications to test their compatibility with Windows Mobile Smartphone and Eclipse ME-generated classes.

[7] SWRL homepage. http://www.w3.org/Submission/SWRL/
[8] Apache Axis. http://ws.apache.org/axis
[9] `http://www.europe.htc.com/en/products/htcp3300.html`

5.4 Evaluation conclusions

The Limbo compiler has been shown to be useful with good resource consumption of the generated code. Clearly there is a need for better documentation for both Limbo and the used ontologies, and Windows Mobile concepts of the OperatingSystem ontology need to be improved.

6 Related work

As said in the introduction, existing tools such as Microsoft's Web Services on Devices and Fast Infoset, fall short of the necessary flexibility of generating different code artifacts for the large variant of devices based on different protocols. These tools lack the versatility of being used for different embedded devices.

In Limbo, we translate WSDL files into a local Regular Tree Grammar (RTG) [4] that describes allowed SOAP envelopes as defined by the WSDL files. Though some frameworks can produce grammar-specific parser of XML data such as done by, e.g., XML Screamer [3], our work leverages this work but casts it in the context of web services, where ontologies are used to support the needed configuration during the generation process. The ontologies are helping to achieve generation-context-awareness and help to make decisions on the targeted platform, with the objective of generating resource efficient code.

Apache Muse[10] can simplify the building of web service interfaces for manageable resources. While Muse has a very specialized objective for the targeted specifications, Limbo has a highly flexible architecture which can be easily extended with the generation of code for .NET code, and other specialized platform such as LeJOS[11]. And more importantly is that we are using ontologies and rule languages to rigorously regulate and instruct the compilation, which can bring us some wiser decisions that is not easily achieved by Apache Muse and other existing approaches.

7 Conclusions and future work

There is an increasing requirement to run web services on resource constrained devices in pervasive computing. In this paper, we present an ontology-enabled compiler called Limbo for the generation of embedded web services. Limbo has followed the Repository architecture style where different frontends and backends can be easily added.

Limbo gets device information from the targeted device in compilation from a Device ontology that imports HardwarePlatform ontology and software platform related ontologies, where resource/power 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 an event mechanism to publish the state change events. We are using a LimboConfiguration ontology to rigorously specify the legal feature combination of Limbo compiler.

Our evaluations through the first Hydra prototype show that the design of the Limbo compiler is successful in terms of resource consumption of the generated web services, complexity hiding of the web service itself and that developers can use Limbo to develop resource efficient web services for a variant of different embedded devices.

A more flexible implementation using OSGi is under development. Web service code generation for .Net platform is planed. And more other hardware platform for example LeJOS is also under exploration.

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

[11] LeJOS homepage. http://lejos.sourceforge.net/

Acknowledgements

The research reported in this paper has been supported by the Hydra EU project (IST-2005-034891).

References

[1] 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.

[2] K. M. Hansen, G. Soares, and W. Zhang. Embedded service sdk prototype and report. Technical Report D4.2, Hydra Consortium, Dec. 2007. IST 2005-034891.

[3] 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.

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

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

[6] H. Wang, Y. F. Li, J. Sun, H. Zhang, and J. Pan. A Semantic Web Approach to Feature Modeling and Verification. In *1st Workshop on Semantic Web Enabled Software Engineering*, Galway, Ireland, Nov 2005. LNCS.