SLL: Running My Web Services on Your WS Platforms

Donald Kossmann
ETH Zürich
8092 Zürich, Switzerland
donald.kossmann@inf.ethz.ch

Christian Reichel
University of Heidelberg, Germany
& Siemens AG, Germany
christian.reichel@email.de

ABSTRACT

Today, the choice for a particular programming language limits the alternative products that can be used to deploy the program. For instance, a Java program must be executed using a Java VM. This limitation is particularly harmful for the emergence of a new programming paradigm like SOA and Web Services because platforms for new innovative programming languages are typically not as stable and mature as the established platforms for traditional programming paradigms. The purpose of this work is to break the strong ties between programming languages and runtime environments and thus make it possible to innovate at both ends independently. Thereby, the specific focus is on Web Services and Service-Oriented Architectures; focusing on this domain makes it possible to achieve this goal with affordable efforts. The key idea is to introduce a Service Language Layer (SLL) which gives a high-level abstraction of a service-oriented program and which can easily and efficiently be executed on alternative Web Services platforms.

Categories and Subject Descriptors

D.2 [Software Engineering] : Software Architectures – Languages; D.2 [Software Engineering] : Design – Representation; D.3 [Programming Languages] : Processors – Code Generation

General Terms: Languages

Keywords: service language layer, decoupling, web services, XML-based service language, XML, transformation

1. INTRODUCTION

The W3C and OASIS have defined many standards in order to enable Service-Oriented Architectures (SOA) and the emergence of Web Services (WS). However, there is no standard programming language to implement WS. As a result, many different languages are used for this purpose; e.g., Java, C# / .NET, BPEL [1], and a battery of Workflow and other domain-specific languages. Unfortunately, the choice for a particular programming language limits the options for platforms to deploy the WS. For example, if C# is used, then the services only run on Microsoft Windows boxes. If BPEL is used, one of the BPEL engines must be installed. Depending on the application server and tools used (e.g., WebLogic or WebSphere), there is a strong dependency between the programming environment and execution platform even within the Java world. This situation is very unfortunate because one goal of the WS vision is to decouple software components and allow best-of-bread development and evolution of the *whole* IT infrastructure.

WWW 2005, May 10-14, 2005, Chiba, Japan.
ACM 1-59593-051-5/05/0005.

This work proposes a *Service Language Layer (SLL)* with the goal to decouple the WS programming model from the execution platform. The idea is to translate programs that define a WS into an intermediary language, called *xSL*, to carry out transformations from xSL to xSL, and to map the resulting xSL programs for deployment onto one of the available platforms (e.g., Java VM, .NET, a BPEL engine, or another special-purpose platform). There are several advantages to such an approach:

Best of bread: Developers can choose the best programming model and platform for deployment independently.

Reduced Vendor Dependency: Developers can implement their applications in the programming language they wish without the fear that they will be tied to a specific vendor for all times. The increased portability of programs is also beneficial if programs need to be run on different devices (e.g., mobile phones, PDAs).

Management and Administration: Companies that have installed several platforms over the years can consolidate their IT landscape and reduce the number of platforms that need to be maintained. In addition to savings in administration costs, this consolidation can result in significant performance improvements.

What is special about the approach proposed in this work is that xSL, the proposed intermediary language, is high-level and can thus be mapped very efficiently to typical WS platforms which also support a high-level programming interface. In other words, rather than using a low-level intermediary language such as three-address code [2], the SLL uses a high-level, XML-based encoding of a WS that can easily be mapped to different target WS engines.

2. SERVICE LANGUAGE LAYER

Figure 1 shows how a Service Language Layer (SLL) can be used to decouple programming models and their execution models:

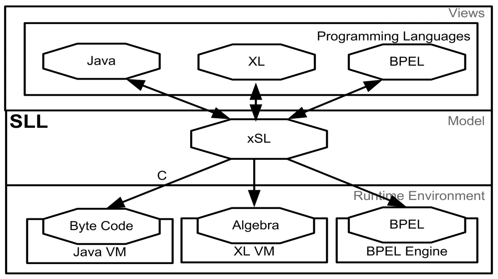

Figure 1: The Service Language Layer (SLL)

This approach removes the direct link between traditional programming languages and platforms within the service domain. SLL defines an XML-based Service Language (xSL) which can be seen as a central model for service-oriented software. Programs in various (service-oriented) programming languages can be transformed into xSL programs. In a second step, xSL programs can be transformed so that they can be deployed and executed on different WS engines. In **Figure 1**, for instance, a BPEL program could be transformed into an xSL program which in turn could be transformed into Java Byte Code for execution on a Java VM. Likewise, a Java program could be executed using the XL VM [3], a special-purpose engine for the execution of WS. There might be limitations in practice (for instance, the current implementation of transformations in our prototype does not support the execution of any arbitrary Java program on a BPEL engine), but in principle any combination is possible as long as the engines in the runtime environment are Turing complete.

As shown in **Figure 1**, it is also possible to transform xSL programs back into programs of a traditional programming language. This way, the SLL can be used as a vehicle for cross-compilation. Another way to look at **Figure 1** is that xSL describes a program at an abstract level which is easy to process for machines, but difficult to read and manipulate for human beings. The syntax of modern programming languages such as Java can be seen as a way to define views on such abstract programs. In other words, the SLL can also be used to decouple the way that programs are represented internally (in xSL) and the way they are presented to programmers in their IDE (e.g., Java or C#). In this regard, the SLL approach works along the lines of the work proposed by Gregory Wilson [4], thereby extending Wilson's work to be applied to the execution of programs, too.

2.1 Overview of xSL

An xSL language that can be used for the SLL must be powerful enough to represent fundamental concepts of programming languages for WS. For instance, using three-address code is not a viable option because it is very difficult to execute that efficiently on a typical WS engine. Furthermore, xSL must be extensible so that it can evolve as the field matures. In addition, xSL must be simple and it must be possible to automate transformations.

In order to support transformations, the current version of xSL is fully XML based. As a result, transformations can be expressed using XSLT or XQuery. XML also makes xSL extensible: new concepts can be represented using new elements. In order to balance expressive power and simplicity, the current version of xSL provides elements for the following concepts (a detailed description and examples can be found in [5]):

- *Service Definition:* The name, version and other non-functional properties such as imports, partner bindings, resources and semantic (QoS, scope, etc.) can be expressed.
- *Service Body:* A representation of the clauses (e.g., triggers, contexts, event handling), operations, statements (e.g., loops and conditionals, transaction and security handling, etc.) and statement combinators (e.g., sequence and data flow) that define the behavior of the WS.
- *Wrapper:* Native functionality of a programming language (e.g., Java library calls of obfuscated code) can be packed into special wrapper elements. These parts can only be executed on the platform that natively supports this functionality.

2.2 Layer Implementation

To implement the SLL, two kinds of transformations are required.

a) **Programming Language (e.g. Java) → xSL.** These transformations start with the language grammar, use parser generators such as ANTLR [6] to generate the Abstract Syntax Tree (AST) and afterwards tree-walkers to get an XML-based representation of the original source code document (e.g. xJava [7]). Additionally, transformations (XSLT) are used to provide a template-oriented mapping between the XML-based source code representation and xSL.

b) **xSL → target technology (e.g. BPEL).** These transformations are completely XSLT based and can provide a direct Mapping (e.g. plain-text) or mappings via intermediate models (e.g. xJava).

As part of the FXL project at Siemens AG and ETH Zurich, several transformations have already been implemented. These include bi-directional mappings from Java to xJava [7] and from xJava to xSL and back to Java, thereby using Axis and Glue in order to implement WS invocation in Java. Furthermore, languages that were specifically designed for WS such as BPEL [1] and XL [3] can be translated back and forth into xSL using FXL. **Table 6** shows the complexity of these transformations.

Table 6: Transformation Complexity [KB]

	to xSL	From xSL
XL	~59 kByte [ANTLR]	~83 kByte [XSLT]
BPEL	~49 kByte [XSLT]	~58 kByte [XSLT]
Java (Axis)	~31 kByte [ANTLR]	~43 kBype [XSLT]

3. Lessons Learned

We have used SLL and xSL for several experiments [5]. In one experiment, we used an online bookshop application that consisted of three WS. One of them was implemented in BPEL and the other two WS were implemented in XL. The orchestration was carried out in BPEL. As a baseline, we executed that application in the traditional way (using Collaxa as a BPEL engine and the XL platform for XL). Furthermore, we used SLL and xSL in order to execute the whole application on the XL platform and alternatively to execute the whole application using a Java VM; thereby not changing a single line of (BPEL and XL) source code. Surprisingly, executing the application entirely on the XL platform was about a factor of 2 faster than in the traditional way. Executing the application on a Java VM was a little slower than on the XL platform.

In another experiment, we used SLL as a means for cross-compilation and compiled the BPEL services from BPEL to XL and back to BPEL and to XL and back and so on. Surprisingly, the size of the code grew only marginally due to cross-compilation and the performance did not degrade with every round of cross-compilation.

4. REFERENCES

[1] Tony Andrews, et al., Business Process Execution Language for Web Services Version 1.1, Mai 2003. http://www-106.ibm.com/developerworks/library/ws-bpel/

[2] A. Aho, R. Sethi, J. Ullman, Compilers: Principles, Techniques and Tools, 1986.

[3] Daniela Florescu, Andreas Grünhagen, Donald Kossmann: XL: a platform for Web Services. CIDR 2003

[4] G. V. Wilson, Extensible Programming for the 21st Century. Jan 2004. http://pyre.third-bit.com/~gvwilson/xmlprog.html

[5] Service Language Layer (SLL) Specification, v1.0, Sept 2004. http://www-dbs.informatik.uni-heidelberg.de/projects/fxl/

[6] Terence Parr. ANTLR, ANother Tool for Language Recognition, 2004. http://www.antlr.org/.

[7] C. Reichel, R. Oberhauser, XML-based Programming Language Modeling: An Approach to Software Engineering, Proceedings of SEA 2004, MIT Cambridge, MA, USA