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@informatik.uni-heidelberg.de

Abstract

Today, the choice for a particular programming language limits the alternative products that can be used to deploy the program. 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. While this goal has been pursued in previous work, the specific focus of this work is on Web Services and Service-Oriented Architectures (SOAs); focusing on this domain and its particular properties 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.

1. Introduction

The W3C and OASIS have defined many standards in order to enable Service-Oriented Architectures (SOA) and the emergence of Web Services; example standards are XML, XML Schema, SOAP, WSDL, WS-Policy [1] and the WS-Security stack [2]. However, there is no standard programming language to implement Web Services. As a result, many different languages are used for this purpose; e.g., Java, C# / .NET, BPEL [3], and a battery of Workflow and other domain-specific languages.

By choosing a programming language, developers limit their options for platforms to deploy their Web Services. 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 (e.g., Collaxa [4]). 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 Web Services vision is to decouple software components and allow best-of-bread development and evolution of the *whole* IT infrastructure.

This paper proposes a *Service Language Layer (SLL)* with the goal to decouple the Web Service programming model from the execution platform. The idea is to translate programs that define a Web Service 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, Collaxa, 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 and laptops).

Management & 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 (section 3).

The idea of an intermediary language is not new: such an approach is used in classic „text book" compiler construction in order to make compilers work for many different target architectures [5]. Furthermore, .NET makes use of a special internal representation of programs in order to support several different programming languages (C#, Visual Basic, etc.) within one common runtime environment (i.e., the CLR). 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 Web Services 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 [5] which can easily be mapped to different target hardware architectures, the SLL uses a high-level, XML-based encoding of a Web Service with the purpose that this

encoding can easily be mapped to different target Web Service engines.

The remainder of this paper is structured as follows. Section 2 presents the main contribution of the paper: the Service Language Layer and its intermediary XML-based service language xSL. Furthermore, Section 2 shows how this SLL was implemented as part of the FXL project which is joint work of Siemens AG and ETH Zurich. Section 3 gives an example, an order processing Web Service from the TPC-W benchmark [6]. The TPC-W benchmark models a book shop and the order processing service is a composition of several Web Services implemented in different programming languages. Section 3 shows how the FXL prototype can be used to cross compile and execute the various Web Services of this example on different Web Service engines. Furthermore, Section 3 contains the results of performance experiments that study the overheads and potential benefits of this approach. Section 4 discusses related work. Section 5 contains conclusions and avenues for future work.

2. Service Language Layer

To reach the aforementioned goal of decoupling between programming models and their execution models a new Service Language Layer (SLL) is introduced (see **Figure 1**).

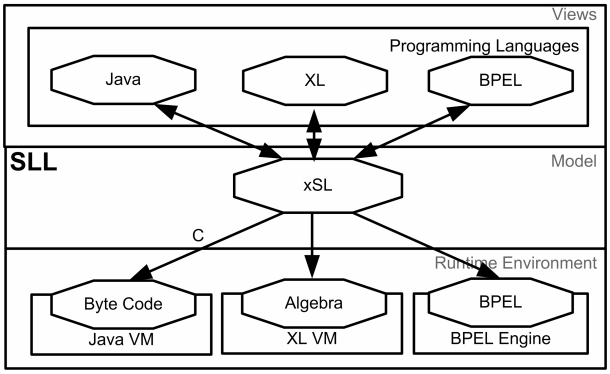

Figure 1: The Service Language Layer (SLL)

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 modified and transformed so that they can be deployed and executed on different Web Service platforms. 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 [7], a special-purpose engine for the execution of Web Services. There might be limitations in practice (for instance, the current implementation of transformations in FXL does not support the execution of an arbitrary Java program on a XL 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. Cross-compilation has been the focus of previous work in the programming languages community and is not the main focus of this work. 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 (i.e., programmers). 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 in order 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 [8], thereby, extending Wilson's work to be applied to the execution of programs, too. For this reason, the SLL also comprise a plain-text language (SL) and graphical representation which can be seen as special views at the XML-based service syntax.

2.1 Requirements

In order to provide the glue between programming and execution model and to reach decoupling, xSL, the internal representation of programs that define Web Services, must fulfill the following requirements:

a) *Expressiveness:* xSL must be powerful enough to represent fundamental concepts of programming languages for Web Services. It must be Turing complete, compatible with existing (Web Service) standards, and at a high-level of abstraction, the same (or higher) level of abstraction as programming languages used for Web Services today. For instance, using three-address code is not a viable option because it is very difficult to execute that efficiently on a typical Web Service engine.

b) *Simplicity:* xSL should be as simple and lean as possible. That way, it is easier to define transformations in the SLL.

c) *Extensibility:* As the field matures, programming languages and platforms for Web Services are extended by new concepts and features (e.g., WS semantic). Correspondingly, it might be necessary to extend xSL so that such extensions can be represented natively and need not be mapped to a sequence of lower-level constructs.

d) *Automation:* Tools should be available to define transformations and to process xSL programs.

There is an apparent conflict between the first two goals: expressiveness and simplicity. As will be shown in the next sub-section, this conflict can be resolved by focusing on one particular domain, Web Service applications. Furthermore, xSL need not be human-readable so that it requires less syntactic sugar.

In order to achieve the last two goals (extensibility and automation), XML was used in order to represent programs in xSL. This way, xSL is naturally extensible (by extending the XML Schema) without invalidating existing xSL programs. Furthermore a whole battery of XML processing tools can be exploited in order to transform xSL programs.

2.2 Layer Specification

As mentioned above, the current version of the xSL language is completely XML based and its specification includes a description of the xSL syntax and semantic. The XML Schema (XSD) file for xSL is approximately 100 KB large and consists of approximately 200 rules and type definitions. The complete XML Schema and a full description of this definition are given in [9]. For brevity, only a high-level overview of the main concepts and a small example is given in this paper. The example shows how the small Java program of **Figure 2** is represented using xSL.

```
class ExampleService {
  public int foo(int param){
    return boo(param, param);}

  public int boo(int param1, int param2){
    return (param1 * param2);}
}
```

Figure 2: The Java ExampleService

An xSL service specification document typically comprises two main parts: the service <definition> and the service <body> element. The following sub-sections describe these two parts.

2.2.1 Service <definition>

The service <definition> contains a description of the service (e.g., meta-data), a listing of all required resources and partners, as well as a declaration of namespaces and implemented interfaces. Elements are:

- *URI:* The URI of the service
- *Name:* Name of the service
- *Version:* Version of this incarnation of the service
- *Namespace:* Target namespace, etc.
- *Interface:* A reference to WSDL port type/interface descriptions
- *Category:* A classification of the service; e.g., as a print service for black and white documents
- *Location:* The physical location of the service
- *Scope:* Physical (geographic) locations from where the service is available
- *Provider:* Information about the author and provider of the service (e.g., name, email, etc.)
- *Conditions:* (WS-)Policies in order to use that service
- *QoS Specs:* Specifications for availability, fault tolerance, and performance guarantees (e.g., throughput and response time)
- *Other:* resource files, service partner, etc.

All of these mentioned elements with the exception of "Name" are optional and most existing programming languages will specify only a small sub-set of these properties. In fact, **Figure 3** shows that for the Java snippet of **Figure 1** only the mandatory "Name" property can be automatically determined; all other properties would be undefined in the xSL representation of that Java program.

```
<service>
  <definition>
    <name>ExampleService</name>
  </definition>
…
</service>
```

Figure 3: xSL ExampleService: <definition> element

Nevertheless, it is important to provide a rich set of properties to describe meta-data of a Web Service. Many modern Web Service languages define such properties and they are important for the deployment of a Web Service. In fact, this set of properties is going to grow with the emergence of standards that formalize the description of the semantics of a Web Service such as OWL-S [10] and WSMF [11]. As mentioned earlier, extensibility is one important reason why we chose XML in order to represent xSL programs.

2.2.2 Service <body>

The service <body> defines implementation specific concerns (e.g., "application logic") of a service specification. For instance, the body contains the definitions of pre- and post conditions, conversation patterns for Web Services that support correlated

messages, partner bindings, time outs, invariants, error handling and compensation actions, triggers, global variable declarations, functions and operations. Functions and operations in turn define the types of parameters, types of results, local variable declarations, expressions, statements (e.g., while loops, if-then-else), and statement combinators (e.g., sequence, parallel, data flow etc.). A complete list of statements and a description of the semantics is described in [9].

The body of the xSL representation for the small Java program of **Figure 1** is given in **Figure 4**. Obviously, the representation is much larger and not intended for readability by humans. However, this representation captures the logic of the Java program and it serves the purpose of being transformable and, thus, executable by different platforms.

When translating to and from Java, C# or another programming language to xSL special attention must be given to the type system. The type system of xSL is XML Schema [12] and the W3C XML Query and Xpath data model [13]. Accordingly, the Java type "int" is represented by the XML Schema type "integer". In general, subtle semantic differences between the original program and the corresponding xSL program can occur due to mismatches in typing; e.g., due to different value ranges of corresponding types. If they are relevant, then those mismatches need to be dealt with manually. In other words, types that cannot be represented by XML Schema must be wrapped, as described in the next sub-section.

```
<service>
…
 <body>
  <operations>
    <operation>
      <name>foo</name>
        <modifiers><public/></modifiers>
        <parameter>
          <variable>
            <type><atomic>
                <ns>xs</ns>
                <name>integer</name>
            </atomic></type>
            <identifier>
              <name>param</name>
            </identifier>
          </variable>
        </parameter>
        <returns>
            <type><atomic>
              <ns>xs</ns>
              <name>integer</name>
            </atomic></type>
        </returns>
        <clauses/>
        <statements>
          <return>
            <expression>
              <operation>
               <identifier>
                 <name>boo</ name>
               </identifier>
               <parameters>
                 <parameter>
                   <variable>
                    <identifier>
                     <name>param</name>
                    </identifier>
                   </variable>
                 </parameter>
                 <parameter>
                   <variable>
                    <identifier>
                     <name>param</name>
                    </identifier>
                   </variable>
                 </parameter>
               </parameters>
              </operation>
            </expression>
          </return>
        </statements>
      </operation>
      <operation>
        <name>boo</name>
        <modifiers><public/></modifiers>
        <parameters>…</parameters>
        <returns>…</returns>
        …
      </operation>
    </operations>
 </body>
</service>
```

Figure 4: xSL ExampleService: <body> element

2.2.3 Wrapping Functionality

xSL must be able to provide a complete representation of any program in any programming language. In theory, this is always possible because xSL is a Turing-complete language. However, often it is not desirable to translate all constructs of a programming language into xSL. For instance, special operators might not be directly represented in xSL because it would be too much effort to *break them down* and represent them using the operators of xSL. Type mismatches can be another reason to sustain from mapping certain parts of a program to xSL.

In order to deal with such situations, xSL provides special *wrapper* elements that allow to represent code that cannot or should not be mapped to native xSL constructs. **Figure 5** shows the corresponding xSL code wrapper element:

```
<wrapper> {
  <language>e.g. Java</language>
  <version> e.g. 1.4.2 </version>
  <mapping> e.g. Castor-based</mapping>
  …
  <![CDATA[
        //wrapped code  ]]>
</wrapper>
```

Figure 5: The xSL code wrapper element

Obviously, using the xSL elements of **Figures 3** and **4** is a much better way to represent the Java program of **Figure 1** because it allows to directly execute the program on any platform for which xSL transformations have been defined. Nevertheless, xSL programs with *wrapper* elements can also be executed. In addition to the XSLT script, that carries out the transformation of the xSL program for the target platform, it must be specified how wrapped code can be executed. For instance, Xquery expressions can be enclosed and it could be specified that these expressions are evaluated by an open-source product such as Galax [14] or Saxon [15]. Furthermore, certain

Java 1.4 elements such as regular expressions can also be excluded from a direct transformation process and executed by a Java VM.

2.3 Layer Implementation

To implement the SLL, two basic 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 [16] 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 [17], an XML-based representation of Java). Additionally, transformations (XSLT [18]) 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 [17] and from xJava to xSL and back to Java, thereby using Axis[19] and Glue in order to implement Web Service invocation in Java. Furthermore, languages that were specifically designed for Web Services such as BPEL [3] and XL [7] can be translated back and forth into xSL using FXL.

Clearly, transforming programs from and to xSL is a complex task. Fortunately, this task must be done only once per programming language and platform. **Table 1** shows the complexity of the XSLT scripts and ANTLR grammars that FXL currently uses for this purpose. Transformations for BPEL and XL are fairly simple because both of those languages have been designed for the same domain as xSL. Not surprisingly, complete transformations for Java are significantly more complex, however the implementation currently consists of core Java + special toolkits such as AXIS [23]. Essentially, the transformations do the same kind of *tweaking* (e.g., adding Java library calls) that a Java programmer would do when using Java to develop Web Services.

Table 1: 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]

In all, it took one Master's student less than six months to learn xSL and the FXL Core Framework and to implement the basic transformations. Obviously, the availability of excellent tools in order to process XML data (as required for xSL) was crucial for this task. There are various XSLT engines (Xalan [20], Saxon [15], etc.) and APIs (DOM4J, JDOM, etc.) and tools which enable the execution of XSLT-based transformations. In addition, the FXL Editor provided an optimized IDE which automates the creation of XML-based source code representations and their transformation and validation within complex pipelines.

3. Example

This section shows the SLL and its implementation in FXL at work. As an example, a trimmed order processing service (*OrderService)* from the TPC-W benchmark [5], which models an online book reseller, is used. This service is composed of other services provided by suppliers and a third party. The workflow of *OrderService* is shown in **Figure 6**.

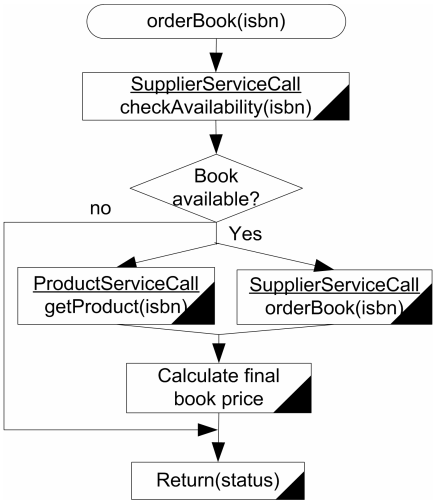

Figure 6: The orderBook operation workflow

As input, this service receives the ISBN of a book that is ordered by a customer. In the first step, the *SupplierService* is called in order to determine the availability of the ordered book. If the book is not available, then the order cannot be executed and the customer is informed accordingly. If the book is available, then the *ProductService* (third party) and the *SupplierService* of the supplier are called concurrently in order to execute the order. The *ProductService* is called in order to get detailed information about the book, including pricing. The *SupplierService* is called in order to initiate the delivery of the book from the supplier to the reseller. From the product information and the shipping information, the final price of the book is computed and a

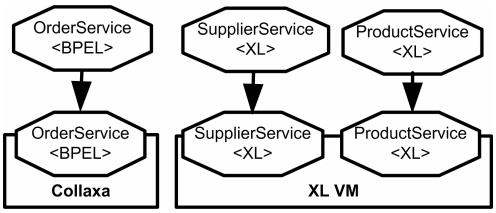

Figure 7: Scenario I – Traditional; BPEL and XL

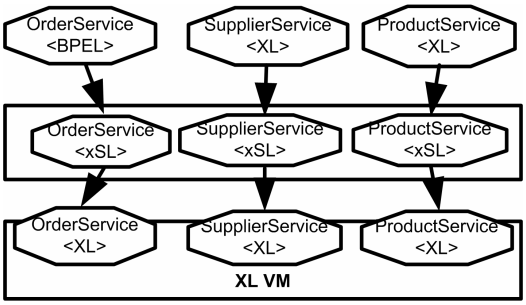

Figure 8: Scenario II – All XL

receipt is computed and returned to the customer. In this example, the *OrderService* of the reseller was implemented using BPEL; it would naturally be deployed on a BPEL engine such as Collaxa [4] (now known as the Oracle BPEL Process Manager). The *SupplierService* of suppliers is implemented in XL. Furthermore, the *ProductService* which is provided by a third party is also implemented in XL. Both of them would naturally be deployed using the XL platform [7]. The following sub-sections describe experiments to execute and cross-compile the three services on different platforms using the SLL approach described in the previous section. All experiments were carried out on a machine with a 3 GHz Pentium 4 processor, 512 MB RAM, and Windows XP-SP2 as an operating system. All Web Services were executed on the same machine so that network latency and congestion did not affect the results.

3.1 Execution Times

Using the SLL implementation of FXL, three scenarios were studied:

Scenario I - Traditional: As a baseline, the *OrderService* was executed on the Collaxa. Both the *SupplierService* and *ProductService* were deployed on the XL platform. This scenario represents the „state-of-the-art" way in order to deploy Web Services: Each Web Service is deployed on the platform that was specifically designed for the programming model in which the Web Service was implemented. FXL and the SSL were not needed for this scenario. **Figure 7** depicts this scenario.

Scenario II – All XL: All three services were deployed on the XL platform. In other words, the definition of all three services was transformed into xSL and, then, the resulting xSL program was transformed so that it could be executed using the XL platform. This scenario is depicted in **Figure 8**. Note that this approach involved an unnecessary mapping and transformation of the *SupplierService* and *ProductService* from XL to xSL and back to XL. (It would have also been possible to execute those native XL services directly on the XL platform. However, as will be seen in the next subsection, the extra transformations did not affect the running times in this example.)

Scenario III – All Java: All three services were deployed on a Java application server using Apache Axis [19] in order to effect SOAP messages for service invocations. Again, the definition of all three services was transformed into xSL; in contrast to Scenario II, however, FXL transformed the xSL to Java and executed it by a Java application server.

The most important result of this experiment was that all three scenarios could be realized, without modifying any source code of the services or tweaking the platforms (Collaxa, XL, and Java application server). The second observation was that the running times of executing the *OrderProcess* workflow did vary. **Table 2** shows the running times in milli-seconds to execute the whole workflow in these three scenarios.

Table 2: Running time of the OrderProcess service

	Traditional	All-XL	All-Java
Running Time	39.3 msec	22.1 msec	24.5 msec

The traditional (non-FXL) way to execute the Web Services showed the worst performance, whereas the All-XL and All-Java scenarios almost had the same running time. The reason for this difference is that the Collaxa BPEL engine is simply not as well-tuned as the other two platforms. Note that in the All-XL and All-Java scenarios, further optimizations are possible because all three Web Services are deployed on the same platform. For instance, SOAP messages in order to implement the invocations of the *SupplierService* and *ProductService* could be replaced by simple method calls or by using more efficient internal protocols of the platforms. Such optimizations can have a huge impact on the performance and are not possible in the traditional, non SLL-enabled way to deploy Web Services. Applying such optimizations makes the use of FXL be even more favourable, but is beyond the scope of this paper.

3.2 Overheads and Risks of SLL Approach

There are potential overheads and risks that come by adding an additional layer. First, there are overheads in order to translate a program into xSL and in order to translate it again for deployment on an existing Web Service platform. These overheads grow linearly with the size of the program. In this example, the transformation time overhead was 24 milli-seconds for the *OrderProcess* service. This overhead must only be paid once, at compilation time (and/or deployment time) of the application.

Second, there are risks: the programmer may have „hand-tuned" the original program in a particular way and this optimization which is probably specific to that particular programming model may be lost due to the translation and mapping. While such effects are well conceivable in a general computing environment, these effects are not likely to matter much for the particular application domain for which Web Services are used. At least, they were not observable in the TPC-W example application. **Table 3** shows the size of programs and the execution time of programs when FXL is used as a cross-compiler.

Table 3: Cross-compilation of Programs: (a) Size of programs [KB] and (b) running time [ms]

Iteration	XL		BPEL		XL	
	Size	**Run. Time**	**Size**	**Run. Time**	**Size**	**Run. Time**
I	1.3	25.5	2.5	39.3	1.5	25.5
II	1.5	25.5	2.7	39.3	1.5	25.5
III	1.5	25.5	2.7	39.3	1.5	25.5

That is, the *OrderProcess* service is cross-compiled from XL to BPEL, back to XL, then to BPEL again, and so on. We carried out three iterations after which the XL and BPEL programs did not change anymore. As shown in **Table 3**, the size of the programs increases slightly, but the running times are not affected. In other words, the number of iterative transformation cycles doesn't worsen the running time which is more dependant to the platform used to execute the program than on the particular implementation of the service.

4. RELATED WORK

The decoupling of programming model and execution model is an active field in current research works and industrial activities. In the industry, several approaches address the creation of intermediate code or models, runnable on a specific platform or transferable to specific target software. As a specific flavor of Model Driven Software Development (MDSD), the Model Driven Architecture (MDA) [21] approach of the Object Management group relies on XML and customizable code generation. It distinguishes between the Platform Independent Model (PIM) and Platform Specific Model (PSM). In order to remove the burden of restricted interoperability of UML tools, OMG's XMI [22] standard was introduced. It specifies an open information interchange model and brings consistency and compatibility to applications created in heterogeneous environments. In contrast, Microsoft's Common Language Runtime (CLR) [23] is designed to be a target platform for multiple languages, ranging from imperative languages, object-oriented languages, scripting languages to declarative languages. While both approaches have certain similarities to xSL, e.g. using an intermediate format (XMI), decoupling strategies (MDA) or targeting a programming language to a specific runtime (CLR), a couple of major differences exist. XMI and the CLR Instruction Set, as well as Java or BPEL, don't provide a specific high-level, XML-based intermediary language optimized for Web Service encoding with the purpose of a flexible mapping to different target Web Service engines. Furthermore, in the case of MDA, current target software transformations are restricted to top-down scenarios and often rely on proprietary languages. In the case of CLR, integration for various languages is offered, but no focus on integration outside the CLR world exists.

[8] says that next-generation programming systems will store source code as XML, so that programmers can represent and process data and meta-data uniformly. In a special sense, this is exactly the idea of xSL. As a central XML-based model and high-level intermediate layer, xSL can represent Web Service descriptions of different programming languages (Java, XL [7], BPEL [3], etc.) and can decouple views and platforms. With regard to traditional compiler principles [5], xSL opens the tight coupling between front-end and back-end and introduces an XML-based intermediate language, e.g., instead of ASTs. Specific transformations can now be realized via standardized technologies (e.g. XSLT) and tools.

Within the field of Web Services various higher-level languages emerged and are still emerging, such as XL [7], OWL-S [10] and BPEL[3]. Their contribution to the WS community is important and accepted. xSL embraces this work and provides a proposal for SLL, which allows a better grade of integration between languages and platforms. The focus and entitlement of this paper, is a flexible decoupling mechanism for WS related applications. It can't provide a complete solution for all aspects (e.g., WS Semantic or WS Policy), but is open for subsequent extensions and standardization efforts.

5. Conclusion

This work showed how a Service Language Layer (SLL) can be used in order to decouple programming models from execution platforms. This way, programmers can select the most appropriate programming language for their problem without the need to worry about the runtime environment or the fate of vendors of runtime environments. In addition, administrators have much more flexibility to consolidate and optimize their IT infrastructure. The use of an SLL makes it much easier to discontinue the service of an out-dated, inefficient, or expensive platform. Furthermore, the SLL makes it easier to deploy programs on different hardware platforms and devices.

The key idea of SLL is to represent programs, coded in different programming languages, in a uniform way, i.e., in a language called xSL. xSL was specifically designed for Web Service applications. It is extensible, easily processable by machines due to its XML syntax, and it has high-level constructs for the most important concepts used in programming languages for the definition of Web Services. As part of the FXL project, transformations are implemented for three important programming languages; Java, BPEL and XL. The approach worked very well for a typical WS application, the TPC-W benchmark which models an online book store. While it is difficult to quantify improvements in administration and software management costs, runtime experiments showed the SLL approach can result in performance improvements.

Since the first results and experiences are very encouraging, we plan to continue this work. One avenue for future work is to write transformations from/to xSL for more programming languages and platforms; C#/.Net are on the top of the list for that kind of work. Furthermore, we plan to continuously extend xSL, as the Web Services technology matures. As a third thread for future work, we will study the potential for performance improvements in more detail and devise new optimization techniques that are enabled by the SLL approach.

6. Acknowledgements

We would like to thank Roy Oberhauser and Ulrich Dinger for their support.

7. References

[1] Siddharth Bajaj, et al., Web Services Policy Framework (WSPolicy), September 2004. ftp://www6.software.ibm.com/software/developer/library/ws-policy.pdf

[2] Bob Atkinson, et al., Specification: Web Services Security (WS-Security), v 1.0, April 2002. http://www-128.ibm.com/developerworks/webservices/library/ws-secure/

[3] 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/

[4] Oracle BPEL Process Manager, 2004. http://www.oracle.com/technology/products/ias/bpel/index.html

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

[6] Wayne D. Smith, TPC-W*: Benchmarking An Ecommerce Solution, 1.2. http://www.tpc.org/tpcw/TPC-W_wh.pdf

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

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

[9] XML-based Service Language (xSL) Specification, Version 1.0, September 2004. http://www.fxl-project.com

[10] OWL-S 1.0 Release, 2004. http://www.daml.org/services/owl-s/1.0/

[11] D. Fensel, C. Bussler, The Web Service Modeling Framework. http://www.swsi.org/resources/wsmf-paper.pdf

[12] Henry S. Thompson, et al., XML Schema Part 1: Structures Second Edition, http://www.w3.org/TR/xmlschema-1/

[13] Mary Fernández, et al., XQuery 1.0 and XPath 2.0 Data Model, W3C Working Draft, April 2002. http://www.w3.org/TR/2002/WD-query-datamodel-20020430/

[14] Galax: An Implementation of XQuery, 2004. http://www.galaxquery.org/

[15] SAXON: The XSLT and XQuery Processor, 2004. http://saxon.sourceforge.net/

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

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

[18] James Clark, XSL Transformations (XSLT), November 1999. http://www.w3.org/TR/xslt

[19] Apache <Web Services /> Project – AXIS, 2004. http://ws.apache.org/axis/

[20] The Xalan-Java is an XSLT processor, 2004. http://xml.apache.org/xalan-j/

[21] OMG Model Driven Architecture, 2004. http://www.omg.org/mda/

[22] OMG XML Metadata Interchange (XMI) Specification, v 1.2, 2002. http://www.omg.org/docs/formal/02-01-01.pdf

[23] About the Common Language Runtime (CLR), 2004. http://www.gotdotnet.com/team/clr/about_clr.asp