Distributed, parallel web service orchestration using XSLT

Peter M. Kelly, Paul D. Coddington, and Andrew L. Wendelborn
School of Computer Science
University of Adelaide
South Australia 5005, Australia
{pmk,paulc,andrew}@cs.adelaide.edu.au

2nd October 2005

Abstract

GridXSLT is an implementation of the XSLT programming language designed for distributed web service orchestration. Based on the functional semantics of the language, it compiles programs into dataflow graphs which can be efficiently executed across a collection of machines in a cluster or grid environment. Calls to web services can be made using the standard function call semantics provided by the language, and occur in parallel using the dataflow model of computation. The programmer is not required to explicitly specify the parallelism, as the details of how programs are scheduled and executed in a distributed environment are abstracted away by the runtime engine. XSLT provides a higher level programming model than many other approaches to web services composition; we explore its use here as a means of easing the task of orchestrating the interactions between services. In addition to the normal XSLT syntax, our system also supports programs written in XSLiTe, an alternative syntax we have developed which uses more concise representations of language constructs, increasing the ease of development, and bringing code readability closer to that of traditional programming languages. Our goal is to ease the construction of applications based on web services composition, such as those used in eScience and other fields in which service oriented architectures are prominent.

1 Introduction

The concept of *grid computing* has received considerable attention in recent years as an approach for harnessing large amounts of computing resources distributed over a wide geographical area, and linking together disparate services and data sources to enable a new class of applications. A wide range of software packages have been written to enable the development of applications that run in such an environment, and have been deployed in research, scientific, and commercial environments.

One of the most prominent approaches to grid computing today is that of *Service Oriented Architectures* (SOA), in which a set of hosts on a network each provide one or more *services*. Each service consists of a set of *operations* that can be invoked by clients by submitting a request to the server containing the name of the service, the name of the operation, and the values for each of the required parameters. Upon completion of the call, the server replies with the result of the operation. No platform-specific knowledge is required by either side, as long as they communicate using an agreed protocol. Many different standards have been proposed over the years that fit this model; today, the most prominent example is the set of specifications for *web services* produced by the standards bodies such as the World Wide Web Consortium, the Global Grid Forum, and OASIS.

Many grid applications, in particular those used for eScience, are built using *web service composition* [13], also known as *orchestration*. This programming model allows the use of functionality implemented by a set of different web services, which each provide some portion of the overall program's functionality. The program which describes the composition specifies the calls that are made to web service operations, and uses the results returned by these operations as inputs to other operations. This model has similarities with the idea of function calls to shared libraries in traditional programming languages, except that instead of restricting the composed functionality to that provided by locally installed libraries, it is a richer system building framework that allows this functionality to be provided by different types of grid resources in a distributed manner.

Another related programming model is *parallel programming*, in which a program consists of a set of concurrently executing tasks which interact with each other by exchanging messages over a network. The goal of parallel programming is to achieve better performance by distributing the work over a number of processors or separate machines. This idea can be applied to the problem of web

Proceedings of the First International Conference on e-Science and Grid Computing (e-Science'05)
0-7695-2448-6/05 $20.00 © 2005 IEEE

IEEE COMPUTER SOCIETY

services orchestration by invoking multiple web service operations simultaneously, so that different machines providing services can be performing their respective operations at the same time, instead of in a sequential manner in which one operation gets called after another.

Our work addresses the problem of providing a web services orchestration framework which provides both ease of use for the programmer, and the ability to coordinate web services in parallel. In contrast to other work done in this area which requires the parallelism to be specified explicitly, we provide a higher level programming model that abstracts the details of concurrency away from the programmer, so that they can concentrate on the functionality of their program without regard to the details of how it is to be parallelised. By doing this, we aim to make it easier to build parallel web service compositions.

2 Related Work

A significant amount of work has been done on the problem of orchestrating web services, both in terms of language design and implementation. The most prominent such language is *Business Process Execution Language* (BPEL) [6], an imperative language consisting of basic control constructs and the ability to invoke calls on remote web services. Each BPEL program is itself exposed as a web service; upon receiving a request from a client, it performs the necessary actions by interacting with other services, and then returning the result to the client. Among the control constructs provided is the ability to explicitly specify service operations which can be executed in parallel. Similar languages include WSCI [16] and WSIPL [1].

The Taverna Workbench [7] provides a graphical environment and execution engine for developing and executing web service coordination workflows in the SCUFL language, which uses a graph based model to connect operations together. While it provides a less powerful set of control structures than BPEL, the workflows lend themselves more directly to parallelism, and the engine exploits this to enable a high degree of parallelism between operations without being explicitly directed to by the programmer. Triana [5] provides a similar environment in which a workflow is constructed from operations implemented as Java classes, and distributed over a set of execution hosts. Web services are also supported as components within the workflow. The workflow model used by these systems is similar to that of the concept of *distributed process networks*. Previous work done in this area [14] has explored the distribution of different nodes of a process network, or *dataflow graph*, to different machines, to enable parallelism.

Existing systems for web service orchestration exhibit a number of shortcomings which we seek to address. Programming directly in BPEL requires the use of a verbose XML-based syntax which requires significantly more code to express ideas than other popular languages; assignment statements that take one line of code in Java or C require six lines of BPEL code, and the language includes no provision for features such as user-defined functions which today are generally considered essential for writing any sort of complex program. The graphical approach used by Taverna and Triana works well for small programs, but it is difficult to write large and complex programs in these systems because the graphical representation of a program can become difficult to work with when large numbers of nodes are present in the graph. The restricted control structures provided by these systems also limit the set of programs that can be expressed.

While mainstream programming languages can easily scale to thousands of lines of code and beyond, the limitations of the systems described above soon become apparent when trying to write large and complex service orchestrations, especially when a non-trivial amount of application logic is to be included in the orchestration code itself. Additionally, the explicit parallelism in some orchestration languages such as BPEL requires a lot of effort on the part of the programmer to identify sections of the code which can be run in parallel, and to manually specify dependencies between instructions in order to avoid race conditions. Instead of this low-level model of parallel programming, we instead take the approach of using a functional language to specify the orchestrations, which allows the parallelism to be automatically inferred by the compiler. Considerable research has been done in this area in the past; languages such as Lisp [11], Haskell [4], and SISAL [9, 15] are examples, all of which have had parallel implementations successfully developed. Although this area of programming language research has seen little application to date in the area of web services orchestration, we feel that it is a useful approach for achieving the goals of easily developing complex service compositions that enable the use of parallelism.

3 System overview

To achieve these goals we have chosen to write a parallel, distributed implementation of the XSLT programming language [18]. XSLT is a pure functional language, with single assignment semantics and side-effect free functions, which means that it can easily be parallelised by a compiler. As a fully-featured programming language, it provides a significantly more powerful programming model than the previous web service composition languages described above, particularly in terms of manipulating XML data, a feature useful for processing the re-

Proceedings of the First International Conference on e-Science and Grid Computing (e-Science'05)
0-7695-2448-6/05 $20.00 © 2005 IEEE
IEEE COMPUTER SOCIETY

sults of web service operations. Web service composition is a new area for the language - to our knowledge, no other XSLT implementation supports it. We believe that the language has a lot of potential in this area that has yet to be realised.

The other key reason for our decision to use XSLT for our work is that it is based on the XML Schema type system, used to specify the parameters and result types of all web service operations defined using WSDL. Many other languages used for web service development, such as Java, have an "impedance mismatch" between their own native type system and that used for web service calls [12]. Some of the constructs available in XML Schema, such as multiple element occurrences, mixed content, and derivation by restriction, do not translate easily into Java language constructs, and some types of objects in Java such as threads and socket connections cannot be transmitted meaningfully as parameters or results of a web service operation. When programming in XSLT, the match between the type systems, combined with the fact that our interpreter supports serialisation of all data types, frees the programmer from having to deal with the sometimes awkward task of translating between two different sets of data semantics.

The powerful programming constructs, ease of parallelism, and type system support of XSLT make it a compelling alternative to other approaches to web service orchestration. In particular, it allows significant application logic to be implemented in the composition code itself, rather than being delegated to web services implementations. This allows a programmer to more effectively make use of a series of web services by augmenting their functionality with additional logic.

3.1 Model of computation

In order to implement parallelism in our execution engine, we compile XSLT programs into *dataflow graphs*. In the dataflow model of computation [10], a program is represented as a set of operations, each of which takes a set of input tokens and produces a set of output tokens. The operations are connected together based on the flow of data from one operation to another. The graph corresponding to a program contains a node for each operation, with edges between the nodes corresponding to the flow of data from an output port of one operation into the input port of another. This model is an alternative to the von Neuman architecture, which represents a running program by global state which is updated by operations executed in sequence. The dataflow model is more suited to parallelism because the dependencies between instructions are explicit; it is thus possible for independent operations to be executed in parallel. Since all mainstream processors available today are based on the von Neuman model, we take the approach of simulating the dataflow model on top of sequential hardware [2]. We achieve parallelism by partitioning the dataflow graph into multiple sections, each of which can be executed on a separate, sequential processor.

The GridXSLT engine consists of two main components: a compiler and an interpreter. The compiler takes an XSLT program as input and generates a dataflow graph from it. The interpreter then executes this dataflow graph either sequentially, using a single instance of the interpreter, or in parallel, using multiple instances of the interpreter running on different machines. This distributed execution model is further described in Section 3.3.

3.2 Web services support

There are two ways in which web services are supported in GridXSLT: exposing a program as a web service, and acting as a client to a remote service.

On the client side, a call to a web service corresponds to a single node in the dataflow graph. The input ports of the node receive data to be submitted as parameters to the web service operation, and once the response is received from the service, the result value is placed on the single output port from the node, from which it is transferred to subsequent nodes in the graph. These semantics are the same as those for other types of function calls. As far as the programmer is concerned, there is no distinction made between calls to internal functions specified by the language (such as string or date manipulation), other functions defined within the program, or web service operations. The specific calling conventions for each of these are handled internally within the interpreter, which involve either calling the internal function, activating the dataflow graph of the user-defined function, or submitting a HTTP request with the name and parameters of the web service operation and asynchronously retrieving the response. A client program written to invoke operations on a set of different services and pass data between these services is known as a *web services composition*, and may either be invoked directly from the command line, or exposed as a service itself.

The GridXSLT engine also allows an XSLT program to be exposed as a web service. Each function corresponds to an individual operation provided by the service, and the parameters and return value from the function correspond to the input and output messages for the operation. All functions in the program can thus be invoked through the web service interfaces by clients. If the client is another XSLT program, this appears as a normal function call; for other languages this is depends on the semantics of the web services support that language provides, such as a proxy object which handles marshalling of the parameters. When a client requests a WSDL definition of the

Proceedings of the First International Conference on e-Science and Grid Computing (e-Science'05)
0-7695-2448-6/05 $20.00 © 2005 IEEE
IEEE COMPUTER SOCIETY

service, this is automatically generated by the interpreter by examining the source code for the program and creating the appropriate definitions for port types, messages, data types and operations, as well as binding information describing how the service can be accessed.

An XSLT program incorporating both of these mechanisms satisfies the requirements for what is often referred to as a *composite web service* [8]. This is a web service which, when invoked by a client, coordinates the actions of other web services, and produces a result based on the output of these services. These other services may either implement specific functionality themselves, or be compositions of further services, in which case the composition effectively consists of multiple levels.

While much research in the area of web service coordination focuses on having the execution engine only perform a small amount of processing, with the bulk of the computation being performed by the actual web services being coordinated, we do not restrict our engine to this model. The amount of computation that is performed by the XSLT program itself versus the amount that is handled by web services implemented in other languages and merely invoked from the dataflow graph, is up to the programmer. The ratio between the two will vary according to the needs of a particular application.

A key feature of the web services support in GridXSLT is that it does not require the programmer to do anything other than write the code for a service or client. Many other web service implementations require significant extra effort to expose a program as a web service, or to invoke operations on a remote service. This often involves tasks such as the creation of WSDL files, generation of proxy classes and so forth. While these tasks can in some cases be automated, they add extra complexity to the development process as they require consideration within the build process and still often take some effort to set up.

Deploying a web service under GridXSLT simply involves placing the program on a web server which is configured to launch the interpreter whenever it receives requests for XSLT files. If the client request is for the WSDL service definition, this will be automatically generated based on the function signatures declared within the program. If the request is for invocation of an operation, the interpreter will handle the de-marshalling of parameters, and then invoke the function in the program that corresponds to the operation name, after which it then encodes the return value and sends it back to the client in the HTTP response.

Because XSLT uses the XML schema type system internally, there is no need for hand-crafting WSDL files and schema definitions separately from the program as is sometimes necessary when working in other languages, due to the type system mismatch mentioned previously. It also does not require wrapper classes to be generated to represent complex data structures described in the XML Schema definitions used by the service, since support for these types is handled directly within the interpreter itself.

3.3 Distributed execution

Execution of programs by the interpreter occurs in a distributed manner. A set of machines, either within a cluster or a grid, is configured to run an instance of the engine. Each of these executes a portion of the dataflow graph, which is split up and assigned to machines by a *scheduler*. This scheduler has knowledge of the capabilities of each machine, what services it provides, and other information such as CPU load and memory capacity. The goal of the scheduler is to divide the graph among machines in such a way that maximises performance, such as taking advantage of lightly-loaded machines and avoiding the transfer of large volumes of data over the network where possible. Each host assumes responsibility for executing its portion of the dataflow graph, and the values that flow from one node to another in the graph are transmitted across the network as necessary.

There are two types of hosts that can participate in the execution of a program:

- *Execution hosts* take a complete or partial dataflow graph, and execute it by invoking the operations and passing values from one node to another. Each of these hosts must run an instance of of the GridXSLT engine.
- *Service hosts* provide web services that are invoked by the program, and do not need any knowledge of the dataflow graph. These do not need to run an instance of the engine; they instead run a hosting environment for web services.

Execution hosts thus require the user to have the appropriate access privileges in order to run an instance of the engine, or to have another party set it up for them, while service hosts can be any arbitrary host on the Internet or other wide area network that are accessible only through web service interfaces. Any execution host may also act as a service host, either by exposing XSLT programs as web services through the engine itself, or by running a separate web service hosting environment.

There are two widely recognised modes of distributed execution for systems such as this: *centralised orchestration* and *decentralised orchestration*. In the centralised approach, one specific host centrally coordinates all of the work done by the others. After parts of the program are executed on another host, the result is sent back to the central one which then passes the data onto the host responsible for executing subsequent nodes in the graph. In

Proceedings of the First International Conference on e-Science and Grid Computing (e-Science'05)
0-7695-2448-6/05 $20.00 © 2005 IEEE
IEEE COMPUTER SOCIETY

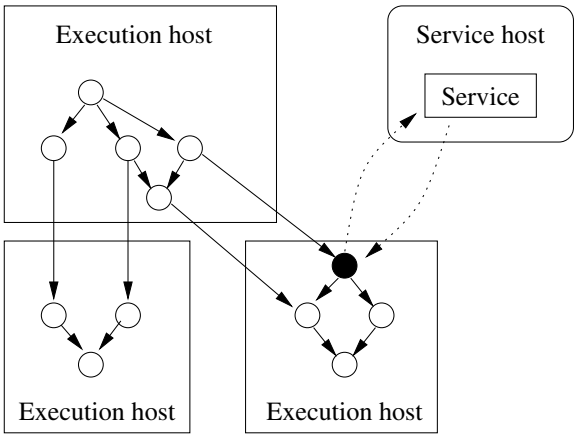

Figure 1: Distributed execution of a dataflow graph

the decentralised model, the work is split up between the hosts in a distributed fashion, and there does not need to be any central knowledge of the distribution. The hosts act independently from any coordinating entity and simply exchange data and interact with each other. We intend to explore both modes of execution in our engine. Other work done in this area suggests that decentralised orchestration tends to yield significantly higher scalability [3].

Figure 1 shows an example of how a dataflow graph would be executed in a distributed setting. Three execution hosts each have portions of the graph assigned to them; the flow of values between nodes within a single host is implemented using memory references, while between hosts the data must be serialised and sent over a socket connection. The shaded node is a call to a remote web service; one of the execution hosts acts as a client to the service and submits the request once the call node becomes ready to execute. The response received from the web service is then sent along the two edges out of this node to the other nodes which are connected to it.

3.4 Load balancing for service access

Instead of specifying specific services to access, the programmer can alternatively specify an identifier which corresponds to a set of possible services, one of which will be chosen at execution time. It is possible to have multiple machines providing an identical service, such that the result of the program is independent of which one gets accessed. In these cases the execution engine can decide which service to submit the request to based on a load balancing algorithm, which takes into account various types of information about the grid resources. If a service call needs to happen multiple times, for example once for each element in an array, then each call can go to a potentially different service instance, thus spreading the load across the set of machines providing that service.

The service resolution is performed by a *resource broker*, which consults a registry of available services. The identifier specified by the programmer is supplied to the registry, which returns a list of matching services that provide the requested functionality. Each time a call needs to be made to one of these services, this list is consulted and a single service instance is chosen to which the request is sent. If the chosen service instance cannot be reached due to a network failure or some other error, the resource broker chooses a different instance instead. In this way, fault tolerance is achieved seamlessly from the point of view of the programmer and user.

Another way in which service instances can be resolved is through the provision of multiple bindings in a WSDL file. For a given abstract service, there may be multiple concrete instances specified, possibly residing on different hosts and accessible through different messaging protocols. This method is used if a service is identified in a program by reference to specific WSDL file, rather than a generic identifier. The use of this method is illustrated in Section 4.

3.5 A concise language syntax

While the semantics of XSLT are well suited to web service composition, one drawback of the language for this purpose is its syntax, which is based on nested XML elements. Each construct in the language corresponds to an element, with attributes indicating certain aspects of the construct. For example, consider the following code for computing the price of a product inclusive of sales tax, depending on the type of customer:

```
<xsl:function name="f:calcprice"
              as="xs:float">
 <xsl:param name="cost"
            as="xs:float"/>
 <xsl:param name="tax"
            as="xs:float"/>
 <xsl:param name="customer"
            as="xs:string"/>
 <xsl:choose>
  <xsl:when
     test="$customer = 'exempt'">
   <xsl:value-of
     select="$cost"/>
  </xsl:when>
  <xsl:otherwise>
   <xsl:value-of
     select="$cost * (1 + $tax)"/>
  </xsl:otherwise>
 </xsl:choose>
```


Proceedings of the First International Conference on e-Science and Grid Computing (e-Science'05)
0-7695-2448-6/05 $20.00 © 2005 IEEE
IEEE COMPUTER SOCIETY

```
</xsl:function>
```

This syntax is considerably more verbose than most other programming languages, and for large programs, it can be tedious to write and difficult to read; BPEL also suffers from this same problem. The use of such a syntax for XSLT likely came about because of the language's focus on processing XML data, and many XSLT programs do in fact contain significant amounts of XML intermixed with language constructs. While it is appropriate for some cases, we believe that for programs such as web service compositions, which are mostly logic oriented, a more condensed syntax is desirable.

XSLT is actually two languages in one; many constructs in the language are specified using XPath expressions [17], which are not based on XML but instead use a more traditional syntax. The `$cost * (1 + $tax)` expression in the example above, as well as the `select` and `test` attributes given for the other `value-of` and `when` elements, are written in XPath syntax. We have developed a language called XSLiTe, which is a superset of XPath that contains equivalent constructs for all elements used by XSLT, while maintaining the same semantics. This syntax eases the task of writing programs, and makes the code considerably clearer and shorter. The above code sample translated into this syntax looks like this:

```
float calcprice(float cost,
                float tax,
                string customer)
{
 if ($customer = 'exempt')
  $cost;
 else
  $cost * (1 + $tax);
}
```

We have developed a parser for this language which can act as input into our dataflow compiler, and can also output the program in the standard XSLT syntax. Using this, we have successfully been able to write programs in XSLiTe and then run the translated programs using existing XSLT engines as well as our own. Initial experiments have shown that in many cases the condensed syntax requires around half the amount of code to express a given piece of functionality compared to the normal XSLT syntax.

4 Example

In this section we describe an example of how the GridXSLT engine can be used to coordinate web services in parallel. We base our example on the use case of a search engine, which, given a set of search terms, performs a query on an index produced from a web crawl and returns a list of matching documents, along with information about them. This example uses two web services - a *reverse index* service, which provides a mapping from a word to a list of matching documents containing that word, and a *forward index*, which, given a document URI, returns information about that document, such as the content or metadata.

The code shown below first uses the reverse index to do a lookup on each of the words provided in the query, and then merges the resulting document lists together. It then consults the forward index to obtain details of each document, which it adds to a result list. The result list is returned as a `resultset` element, containing a series of `result` children. Each of these has a `uri` attribute corresponding to the list entry, an `abstract` child containing the first 1000 characters of the document, and a sequence of other elements corresponding to the document metadata. A `size` attribute is also added to the `resultset` element indicating the number of documents it contains.

This code uses the functional programming style of XSLT to produce the results; rather than assigning to variables or appending to lists in a sequential fashion, the results of each statement are added in-place to the list. For example, the result of executing the `for-each` construct is the list of `result` elements containing the values returned by the code within the block, including the results of the `getmetadata()` service call. This code is evaluated once for each element in the list, with the context variable `(.)` being mapped to the relevant list element for each evaluation.

Namespaces are used to associate a service definition with a particular prefix. In the normal XSLT syntax, these are defined using the standard XML namespace mechanisms, while in the XSLiTe syntax shown here they are declared using a `namespace` statement. When a call to a function such as `ri:lookup()` is made, the namespace associated with the prefix is inspected. If, as in this case, it is mapped to a web service definition, then the function call is then treated as call to the corresponding web service operation.

```
function doSearch(var $searchterms)
{
 namespace ri "reverse_index.wsdl";
 namespace fi "forward_index.wsdl";

 var $matches =
  for $t in $searchterms
   return ri:lookup($t);
 var $distinct =
```


Proceedings of the First International Conference on e-Science and Grid Computing (e-Science'05)
0-7695-2448-6/05 $20.00 © 2005 IEEE
IEEE COMPUTER SOCIETY

```
  distinct-values($matches);

 %resultset {
  #size(count($distinct));
  for-each ($distinct) {
   %result {
    #uri(.);
    fi:getmetadata(.);
    %abstract {
     substring(fi:getcontent(.),
               0,1000);
} } } }
```

The compiled dataflow graph for this function is shown in Figure 2. This graph shows the dependencies between different parts of the code; the reverse index lookups must be done before document information can be obtained, because the latter part of the code requires the merged document list as input. The two rectangular boxes surrounding the lookup operation and result element construction correspond to the code within loops; these parts of the graph are executed multiple times - one for each item. Each subgraph execution is independent of the others, and thus they can all be run in parallel with each other. Several calls to the reverse index service can be in progress simultaneously, and, once the merged document list is constructed, each document can be processed in parallel with the others. Within the code for constructing a single result element, the `getcontent()` and `getmetadata()` operations can also be run in parallel.

With the provision of multiple service instances for both the reverse index and the forward index, the individual calls to `lookup()`, `getcontent()` and `getmetadata()` can be distributed across the set of hosts providing these services, reducing the amount of computation that is performed by each. In this example, which directly maps the namespace prefixes for the operations to WSDL files, the set of available hosts is determined by examining the set of bindings specified in the WSDL definition for each service.

While this is only a fairly simple example, it demonstrates the way in which parallelism can easily be obtained for a set of web services distributed across multiple machines. The programmer does not need to explicitly specify this parallelism in the program code; it is automatically inferred by the compiler. Additionally, the ability to use a web service simply by associating a namespace prefix with its WSDL definition and then making calls to functions in that namespace, makes the task of composing the functionality of several web services easy. The code shown above can be executed directly by the GridXSLT engine without any extra build steps such as the generation of proxy classes, as the serialisation of data sent to and from the web services is all handled internally.

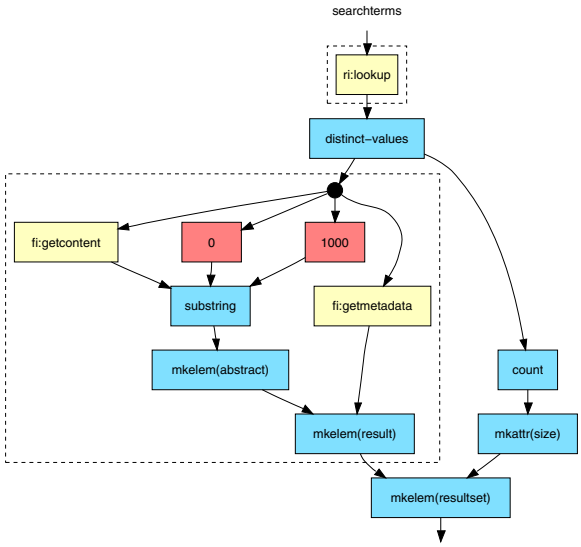

Figure 2: Dataflow graph for the example program

5 Conclusion

In this paper we have presented the design of the GridXSLT engine, which we are currently in the process of implementing. The design addresses our goals of providing an easy to use framework for composing web services in a way that takes advantage of parallelism and allows for the inclusion of application logic in the composition programs themselves. A number of features of our design reduce the effort required to develop both web services and clients, including the use of a concise syntax for specifying programs, implicit serialisation of data without the use of proxy classes, automatic generation of WSDL files, and a high level programming model which lends itself to automatic parallelisation. Our work also enhances the usefulness of XSLT by providing the ability to use it for web service composition and parallel programming, and this is leveraged to provide a more powerful composition framework for web services than many other systems which have been designed for this purpose.

Our current work involves implementation of this design. To date, we have written a prototype compiler and interpreter capable of executing simple XSLT programs, and a parser for XSLiTe which is able to translate programs into the standard XSLT syntax accepted by other engines. Much of the implementation effort so far has been concerned with getting the groundwork in place for a full implementation of the language, including support for parsing and verification of XML Schema and XSLT source files, and associated infrastructure for regression

Proceedings of the First International Conference on e-Science and Grid Computing (e-Science'05)
0-7695-2448-6/05 $20.00 © 2005 IEEE
IEEE COMPUTER SOCIETY

testing and development tracking. In the near future we expect to have a demonstration available which will include web services support and parallel execution. Our intention is to eventually provide support for 100% of the specification, however for the medium term we have identified a subset of the specification necessary for evaluation of our research ideas.

The concepts of grid computing and specifically service oriented architectures are becoming increasingly prominent in both scientific and commercial areas. The ability to easily and efficiently compose services together in a high level programming model is a desirable capability which we feel is likely to lead benefits in a range of areas that utilise distributed computing resources. Our research attempts to address this need by using a number of novel approaches as outlined in this paper.

Further information about this project is available at `http://gridxslt.sourceforge.net/`.

References

[1] D. W. Cheung, E. Lo, C.Y. Ng, and T. Lee. Web services oriented data processing and integration. In *The Twelfth International World Wide Web Conference (WWW2003)*, Budapst, Hungary, May 2003.

[2] S. A. Edwards. Tutorial: Compiling concurrent languages for sequential processors. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 8(2):141–187, April 2003.

[3] G. B. Chafle et. al. Decentralized orchestration of composite web services. In *WWW Alt. '04: Proc. 13th international World Wide Web conference on Alternate track papers & posters*, pages 134–143, New York, NY, USA, 2004. ACM Press.

[4] P. W. Trinder et. al. GUM: a portable parallel implementation of Haskell. In *PLDI '96: Proc. ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 79–88, New York, NY, USA, 1996. ACM Press.

[5] S. Majithia et. al. Triana as a graphical web services composition toolkit. In Simon J. Cox, editor, *Proc. UK e-Science All Hands Meeting*, pages 494–500. EPSRC, CD-Rom only, September 2003.

[6] T. Andrews et. al. Business Process Execution Language for Web Services version 1.1. http://ifr.sap.com/bpel4ws/, May 2003.

[7] T. Oinn et. al. Delivering web service coordination capability to users. In *Proc. 13th international World Wide Web conference on Alternate track papers & posters*, pages 438–439, New York, NY, USA, 2004. ACM Press. http://taverna.sf.net.

[8] V. Agarwal et. al. A service creation environment based on end to end composition of web services. In *WWW '05: Proc. 14th international conference on World Wide Web*, pages 128–137, New York, NY, USA, 2005. ACM Press.

[9] J. T. Feo, D. C. Cann, and R. R. Oldehoeft. A report on the SISAL language project. *Journal of Parallel and Distributed Computing*, 10(4):349–366, 1990.

[10] W. M. Johnston, J. R. Paul Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys (CSUR)*, 36(1):1–34, 2004.

[11] D. A. Kranz, Jr. R. H. Halstead, and E. Mohr. Mul-T: a high-performance parallel Lisp. In *PLDI '89: Proc. ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 81–90, New York, NY, USA, 1989. ACM Press.

[12] E. Meijer, W. Schulte, and G. Bierman. Programming with circles, triangles and rectangles. In *Proc. XML 2003*, 2003.

[13] S. Tai, R. Khalaf, and T. Mikalsen. Composition of coordinated web services. In *Proc. 5th ACM/IFIP/USENIX international conference on Middleware*, pages 294–310, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[14] D. Webb, A. L. Wendelborn, and K. Maciunas. Process networks as a high-level notation for metacomputing. In *Proc. International Parallel Programming Symposium (IPPS'99), workshop on Java for Distributed Computing*, Puerto Rico, April 1999.

[15] A. L. Wendelborn and H. Garsden. Exploring the stream data type in SISAL and other languages. In *Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, pages 283–294, 1993.

[16] World Wide Web Consortium (W3C). Web service choreography interface (WSCI) 1.0. W3C Note, August 2002. http://www.w3.org/TR/wsci/.

[17] World Wide Web Consortium (W3C). XML path language (XPath) 2.0. W3C Working Draft, April 2005. http://www.w3.org/TR/xpath20/.

[18] World Wide Web Consortium (W3C). XSL transformations (XSLT) version 2.0. W3C Working Draft, April 2005. http://www.w3.org/TR/xslt20/.

Proceedings of the First International Conference on e-Science and Grid Computing (e-Science'05)
0-7695-2448-6/05 $20.00 © 2005 IEEE

IEEE COMPUTER SOCIETY