Extensions to Web Service Techniques for Integrating Jini into a Service-Oriented Architecture for the Grid

Yan Huang and David W. Walker

Department of Computer Science
Cardiff University
PO Box 916
Cardiff CF24 3XF
United Kingdom
{Yan.Huang,David}@cs.cardiff.ac.uk

Abstract. This paper discusses how to adapt Jini to create an OGSA-compliant system for Grid computing by introducing Web services techniques into the Jini system. Service Workflow Language (SWFL), an extension to WSFL for describing jobs composed of interacting Web services, is presented. SWFL provides a simple and succinct way of describing the conditional and loop constructs of Java, and supports more general data mappings than WSFL. In addition, a tool for automatically generating Java code to execute a composite job described in SWFL is described.

1 Introduction

This paper describes how Jini services can be integrated with Web services within a common Service-Oriented Architecture (SOA) for Grid computing. This common SOA permits the transparent interaction of Jini services and Web services, thereby extending the usefulness and applicability of both approaches. The main benefits of this work are: (1) Jini services will be accessible from outside of a Jini community; (2) Jini services will be invocable in the same way as other Web services; and, (3) Jini services will be able to be integrated with other Web services.

Web services have arisen as an essential component of the infrastructure of e-Business, and enable business-to-business transactions via the Internet. In general, these B2B transactions take place directly between computer programs, rather than between computer programs and users.

A Web service has five essential attributes [5]: It can be *described* using a standard service description language, usually Web Service Description Language (WSDL) [1]; it can be *published* to a registry of services, usually a UDDI (Universal Description, Discovery, and Integration) registry; it can be *discovered* by searching the service registry; it can be *invoked*, usually remotely, through a declared API; and, it can be *composed* with other Web services.

A WSDL document describes one or more Web services, each of which is made up of multiple messages, port types, and bindings. A message gives an abstract definition of the input and output data of a Web service. A port type is used to describe the functionality of a Web service in terms of a set of operations. A binding associates a concrete protocol and data format with a port type. The Simple Object Access Protocol (SOAP) is a widely used protocol for Web service messaging, and uses an XML data encoding and HTTP transfer protocol. The binding may also specify the security mechanism for the port's communications.

The Jini networking system aims to support the rapid configuration of devices and software within a distributed computing environment [2]. These devices and software are made available to remote clients as Jini services. Jini is one of the network technologies that are suitable for building the middleware for Grid computing. A central theme of Grid computing is the sharing of resources within a virtual organization through direct transactions between computer programs [4]. This has led to the emerging concept of a Grid service and to the Open Grid Services Architecture (OGSA) which is currently being developed on the basis of Web service concepts and technologies [3]. In this approach, Grid services are regarded as specialized extensions of Web services, and support new types of problem-solving applications that are composed of services. In this paper, a Jini-based Grid will be developed into a OGSA-compliant system. This is done by combining Web service and Jini service SOAs.

Jini's SOA is very similar to that used by Web services. By using Jini lookup services, Jini services can be published, discovered, and invoked and so possess three of the five essential attributes of a Web service enumerated at the start of this section. However, the other two of the five essential attributes, service description and composition, are not part of the basic Jini system. In addition, Web services use XML-based messaging which is not prescribed (nor prohibited) by the Jini communication model.

The simplest way to combine the Web service and Jini service SOAs is by registering services using both the service registry and lookup service mechanisms. The service registry works in the world external to the Jini community, and the lookup service works within the Jini community. Thus, correspondingly, there are two sorts of publish operations, two sorts of find or discovery operations, and two sorts of bind or invoke operations.

Henceforth, a request for a Jini service through the non-Jini mechanism will be called an *indirect service request*, and a request for a Jini service through a Jini lookup service will be called a *direct service request*. An indirect service request finds the service requested by sending a query based on the service description to the service registry and a task description based on a service flow language is formed, also based on the service description. An agent service called the *Workflow Engine* accepts the indirect service request, translates the request into a service request understandable by Jini, and makes sure the request is carried out and the results sent back to the requestor.

2 Service Workflow Language

In many cases it is desirable to create a job or application by composing multiple services. Such applications can be represented by a graph of interacting services that must be specified in a job description[1]. Thus, not only a service description language, but also a job description language, are needed to standardize the description of both services and composite jobs.

In May 2001, IBM's WSFL [6] and Microsoft's draft of XLANG [7] were released as languages for describing the composition of Web services. Although the intended uses of WSFL and XLANG are broadly the same, they have completely different structures: WSFL directly represents the control and data flow of an application in terms of its workflow graph; XLANG is closely based on the use of Java-like language constructs to describe a job. In August 2002, a combined version of WSFL and XLANG, called Business Process Execution Language for Web Services (BPEL4WS) [9], was published that largely inherited the programming-oriented approach of XLANG, rather than the graph-based approach of WSFL. BPEL4WS is effective for representing service-based composite applications for which the order of execution of the services is pre-defined. However, it does not have the same flexibility as WSFL, in which the only constraints on the order of execution of services are implicit in the workflow graph. Having a pre-defined service execution order, as in XLANG and BPEL4WS, is not suitable for representing a distributed application, where the ability to dynamically partition and schedule services at runtime is important in order to exploit potential parallelism and to make best use of the available distributed resources. WFSL, however, does allow this capability, and hence provides a flexible and effective basis for representing a Grid application.

Service Workflow Language (SWFL) extends WSFL in two important ways:

1. SWFL improves the representation of conditional and loop control constructs. Currently WSFL can handle *if-then-else*, *switch*, and *do-while* constructs and permits only one service within each conditional clause or loop body. SWFL also handles *while* and *for* loops, and permits sequences of services within conditional clauses and loop bodies.
2. SWFL permits more general data mappings than WSFL. SWFL can describe data mappings for arrays and compound objects.

SWFL can describe the conditional and loop control constructs of the Java programming language. The motivation for SWFL was to develop a tool for automatically converting the workflow description of a composite job into a Java program for running it. This tool, `SWFL2Java`, is also described in this paper.

As an example, consider loop control constructs. WSFL provides loop control flow through the optional `exitCondition` attribute of an activity. Exit conditions are represented with an XPath-based syntax. If on completion of the activity the exit condition evaluates to false, then the activity is run again. This continues until, after the activity has been run some number of times, the exit

[1] Here a job is a composition of interacting services.

condition evaluates to true, after which control flows to the next activity. For example:

```
<activity name="LoopActivity"
          exitCondition="XPath1 &gt; XPath2"/>
```

For clarity, attributes not relevant to the example have been omitted. The activity `LoopActivity` will be run repeatedly until the exit condition evaluates to true. This is equivalent to the following Java code:

```
do {
    LoopActivity ();
} while (X1 > X2)
```

Now consider the *for* loop. Since a *for* loop can be rewritten in terms of a *while* loop, and since the latter can be represented in WSFL, then it would appear that a *for* loop can also be represented in WSFL. However, there is a problem. Suppose we have a *for* loop that iterates on a certain activity. Then the body of the corresponding *while* loop contains that activity, followed by an new activity that simply updates the loop control variable. In WSFL an exit condition can be associated with only one particular activity, so there can be only one activity within a loop. Furthermore, if the activity in a *for* loop takes the loop control variable as one of its inputs, which is often the case, WSFL provides no way to represent the internal dataflow inside the body of the loop.

The difficulties in representing certain types of conditional and loop control constructs in WSFL have led us to develop SWFL. SWFL is a new job description language that makes three main extensions to WSFL, affecting the elements `wsfl:activityType`, `wsfl:controlLinkType`, and `wsfl:dataLinkType`, and the definition of a new element, `SWFL:jmapType`. In SWFL, conditional and loop processes are treated as special activities, which allows their data and control flow to be defined more clearly. They are added into the `jactivityType` element. The resulting `jactivityType` XML schema is as follows:

```
<xsd:complexType name="jactivityType">
    <xsd:choice>
        <xsd:element name="normal" type="wsfl:activityType"/>
        <xsd:element name="while" type="loopType"/>
        <xsd:element name="dowhile" type="loopType"/>
        <xsd:element name="for" type="loopType"/>
        <xsd:element name="if" type="controlType"/>
        <xsd:element name="switch" type="controlType">
            <xsd:key name="CasePortName">
                <xsd:selector xpath="case"/>
                <xsd:field xpath="@port"/>
            </xsd:key>
        </xsd:element>
    </xsd:choice>
    <xsd:attribute name="operation" type="NCName"/>
</xsd:complexType>
```

In the above `jactivityType` schema, the activity type is extended to six kinds of activities. These are: `normal`, `if`, `while`, `dowhile`, `for` and `switch`. The `normal` activity is the same as an activity defined in WSFL, and is of type `wsfl:activityType`. The others are newly-defined activity types corresponding to the conditional and loop constructs in the Java programming language.

The `for`, `while` and `dowhile` activities are all of type `loopType`. `loopType` is an extension of `wsdl:operationType` through the addition of a new element called `expression` and a new attribute called `setParallel`. A loop activity has `input` and `output` elements to specify its input and output data. In *while* and *do-while* loops the element `expression` is a Boolean expression based on the `input` message of the loop activity. However, in the case of a *for* loop, the `expresssion` element is represented by three statements separated by semicolons in the form: *initial statement ; Boolean expression ; increment statement*. This Java-based format currently allows only simple *for* loops of the type given is the following example. The loop control variable used in the statements is from the `input` message of the activity. Specifying the loop control variable as one of the parts of the `output` message of the loop activity allows the activity within the loop, as well as activities after the loop, to accept the loop control variable as input. As an example, consider the following `for` loop activity:

```
<wsdl:message name="forloopMessage">
    <wsdl:part name="out0" type="string"/>
    <wsdl:part name="out1" type="int"/>
    <wsdl:part name="index" type="xsd:integer"/>
</wsdl:message>

<activity name="forActivity">
    <for setParallel="no">
        <input message="forloopMessage"/>
        <output message="forloopMessage"/>
        <expression>
            <![CDATA[index=0;index<100;index++]]>
        </expression>
    </for>
</activity>
```

In this example there is a `part` element named `index` in the message input to the `for` activity which is used in the `expression` of the activity. The `setParallel` attribute of the activity is used to indicate whether the iterations of the `for` activity can be performed in parallel. If `setParallel` is set to "`yes`" a scheduler could arrange for the loop iterations to be done in parallel on different machines that provide the service in the loop body.

Normally an activity has one control input port and one control output port, but conditional and loop activities have multiple output control ports. A loop activity has two output control ports. As long as the loop condition is satisfied, control flows to the sequence of activities inside the loop body; otherwise, control flows to an activity after the loop body.

For `if` and `switch` activities there is a control output port for each conditional clause. The following is an example of a *switch* statement in Java:

```
switch(int0){
    case 10: ...; break;
    case 20: ...; break;
    case 30: ...; break;
    default: ...;
}
```

In this example, the switch has four cases so there are four output control ports to which control can flow on exiting the activity. The corresponding description in SWFL is as follows:

```
<SWFL:switch expression="int0">
    <SWFL:input name="switchInputMessage" message="switchMessage"/>
    <SWFL:output name="switchOutputMessage" message="switchMessage/">
    <SWFL:case port="0">10</SWFL:case>
    <SWFL:case port="1">20</SWFL:case>
    <SWFL:case port="2">30</SWFL:case>
    <SWFL:defaultCase port="default"/>
</SWFL:switch>
```

The attribute `expression` specifies the expression controlling the switch statement which must be of type `byte`, `short`, `int`, or `char`, and is from the input message of the switch activity. The value of each `case` must be a literal with the same type as the `expression` attribute. The `port` attribute of the `case` element is used to specify for that case the port of the switch activity that a control link flows out of.

The `if` conditional activity is very similar to the `switch` activity, as may be seen in the following example an `if` activity defined in SWFL:

```
<SWFL:if>
    <SWFL:input name="ifInputMessage" message="ifMessage"/>
    <SWFL:output name="ifOutputMessage" message="ifMessage/">
    <SWFL:case port="0"><![CDATA[int0 < 0]]></SWFL:case>
    <SWFL:case port="1">int0==0</SWFL:case>
    <SWFL:defaultCase port="default"/>
</SWFL:if>
```

An `if` activity does not have an `expression` attribute. Instead, the value of the `case` element is a Boolean expression composed from data in its `input` message and from literals.

The introduction of `SWFL:controlType` and `SWFL:loopType` activities having multiple control output ports, makes it necessary to specify which port of the source activity a control link flows out if the source activity is a conditional or loop activity. `SWFL:jcontrolLinkType` extends `wsfl:controlLinkType` by adding an optional attribute called `controlPort` to the `controlLink` element which is used only when the source activity of the control link is a conditional or loop activity. It specifies which control output port of the source activity is the source of the control link.

The WSFL syntax for data mapping is sufficient to define the mapping in a data link when the mapped data is a one-layer `complexType` (i.e., the elements in the `complexType` are `simpleType`), and is not an element of an array. This means that WSFL can handle only cases in which the input to an activity is a primitive datatype or an object containing primitive datatypes. Also WSFL allows only one data mapping in a data link. This means that an activity can accept data from only one source activity. SWFL provides a new definition of data mapping, `SWFL:jmapType`, that overcomes these limitations.

In the specification of `SWFL:jmapType` the attributes `sourceMessage` and `targetMessage` specify the source message and the target message of a data link to which the data map applies. A `SWFL:jmapType` can have multiple `part` elements, each corresponding to a different data mapping. A `part` has two sub-elements of type `SWFL:dataPartType`, named `sourcePart` and `targetPart`, which are used to specify a particular data element in a message. `SWFL:dataPartType` specifies a particular path leading to the final data element involved in the data mapping. It can contain any number of `item` elements. An `item` element has either a `field` element to specify a field of a complex type, or an `index` element to specify an element of an array.

In the example of `SWFL:mapType` below there is one data mapping: the data `u1.a.b[5]` is mapped to `u2.c` in Java notation. `u1` is a `part` of the `sourceMessage`. The mapped data is the 5th element of the array `b` which is a `field` of `a`. `a` itself is a `field` of part `u1`. `u2` is a `part` of `targetMessage` and `c` is a `field` of `u2`.

```
 <SWFL:jmap sourceMessage="sourceMessage" targetMessage="targetMessage">
    <SWFL:part name="part0">
        <SWFL:sourcePart name="u1">
            <SWFL:item><SWFL:field name="a"/></SWFL:item>
            <SWFL:item><SWFL:field name="b"/></SWFL:item>
            <SWFL:item><SWFL:index dimension="0" index="5"/></SWFL:item>
        </SWFL:sourcePart>
        <SWFL:targetPart name="u2">
            <SWFL:item><SWFL:field name="c"/></SWFL:item>
        </SWFL:targetPart>
    </SWFL:part>
</SWFL:jmap>
```

3 Implementation Aspects

We have discussed how a job composed of interacting services can be represented by the SWFL job description language. The main motivations for developing SWFL were to describe Java-oriented conditional and loop constructs, to permit sequences of more than one service within conditional clauses and loop bodies, and to overcome limitations inherent in WSFL's data mapping approach. In addition, we have developed: (1) a tool called `SWFL2java` that converts the description of a job in SWFL into executable Java code; and, (2) a Workflow Engine that provides an execution environment to run the composite job.

The details of `SWFL2Java` are discussed elsewhere [8], however, we will give here an overview of the implementation. In `SWFL2Java` a SWFL document is not translated directly into a Java program but is stored in an intermediate form as a Java `FlowModel` object. This is made up of two Java `Graph` objects: `DataGraph` and `ControlGraph`. The former stores the data flow structure of the flow model, and the latter stores its control flow structure. One reason for storing the job description in this intermediate form is to be able to interact readily with Java-based tools for the visual composition of Web services. In such tools a composite job is represented as a graph in which activities/services correspond to nodes, and data links and control links correspond to different kinds of directed edges. The graph can be stored as a `FlowModel` object and converted to and from SWFL, as well as into a Java program. Another reason for using the `FlowModel` form is to reduce the overhead when the same job is used many times and scheduled on different resources. In such cases it is easier to generate the Java code from the intermediate form rather than starting from the original SWFL.

Given a graphical `FlowModel` of arbitrary structure to be transformed into a Java Jini-based distributed program, the following three issues have to be addressed: how to find all the services; how to decide the order of execution of services based on the flow model; and, how to automatically name all the variables, class names, and methods.

Normally the first thing a Jini-based distributed program does is to discover the services that are going to be used in the program. Thus, to automatically create a Java Jini-based program from a graphical flow model, service discovery is the first issue to be addressed. Service discovery is performed by an assistant class called *ServiceFinder*. The *ServiceFinder* thread takes a list of services and discovers each using a *ServiceDiscoveryManager*. Once a service is discovered, its *ServiceItem* is downloaded and stored.

After all the services needed in the program have been found and stored, the main program is generated. The problem here is determining the order in which tasks are processed by services. This order is deduced from the control graph using the Task Processing Sequential Order Generation algorithm described in [8].

In automatically generating a Java program several naming issues need to be decided. In the automatically produced program, the class name takes the form *FlowModel_XXX* where *XXX* is the name of the flow model. The class has a constructor and an *execute()* method that runs the job specified in the flow model. The parameters of the *execute()* method are determined by `flowSource` in the flow model. The parameter names are the same as the corresponding `part` names of the `flowSource` message. The return variable of an activity takes the form `XXX_return` in which `XXX` is the name of the activity.

Here we provide an example of a Java program automatically created by `SWFL2Java`.

```
import net.jini.core.lookup.ServiceItem;
```

```
  import net.jini.core.lookup.ServiceTemplate;
  import java.util.HashMap;
  import SWFL2Java.XML2Graph.ServiceFinder;
  public class FlowModel_Example1{
    private HashMap taskTemplateMap = null;
    private ServiceFinder serviceFinder = null;
    public FlowModel_Example1(){
        this.taskTemplateMap = new HashMap();
        this.setTaskTemplateMap();
        this.serviceFinder = new ServiceFinder(this.taskTemplateMap);
        Thread thread = new Thread(this.serviceFinder);
        thread.start();
        try{
            thread.join(2*60*1000);
        } catch (java.lang.InterruptedException e){
            System.out.println("Failure in finding services.");
            System.exit(-1);
        }
    }
    private void setTaskTemplateMap(){
        this.taskTemplateMap.put("task_1", new ServiceTemplate(null,
            new Class[]{services.Matrix.MatrixInterface.class}, null));
        this.taskTemplateMap.put("task_0", new ServiceTemplate(null,
            new Class[]{services.Math.MathInterface.class}, null));
    }
    public double[][] execute(int size, double[][] B)
                    throws java.rmi.RemoteException{
        for(int in0=0; in0<size; in0++){
            for(int in1=0; in1<size; in1++){
                services.Math.MathInterface task_0 =
                    (services.Math.MathInterface)
                    (this.serviceFinder.getServiceItem("task_0")[0].service);
                double task_0_return = task_0.function(in1 ,in0);
                B[in0][in1] = task_0_return;
            }
        }
        services.Matrix.MatrixInterface task_1 =
            (services.Matrix.MatrixInterface)
            (this.serviceFinder.getServiceItem("task_1")[0].service);
        double[][] task_1_B = task_1.inverse(B);
        return task_1_B;
    }
}
```

4 Conclusions

This paper has discussed several technical issues involved in updating a Jini system into a OGSA-compliant system and has focused on one of the most important issues – defining an XML-based description language for describing composite service-based Grid applications. SWFL, an extension to WSFL, is such a language that describes Java-oriented conditional and loop constructs and enhances WSFL's data mapping facility. Whereas WSFL can represent only *if-then-else*, *switch*, and *do-while* constructs, SWFL can also represent *while* and simple *for loop* constructs. SWFL also permits sequences of more than one activity within conditional clauses and loop bodies. SWFL enhances WSFL's data mapping capabilities by handling compound objects and arrays, and by permitting activities to accept input data from more than one source activity.

Given an SWFL job description, the `SWFL2Java` tool can generate a representation of the corresponding data and control link structure in the form of a Java `FlowModel` object. From this the corresponding Java program can be automatically generated.

SWFL can also be used to describe composite jobs made up of both Jini and Web services. The work described in this paper is part of a larger programme of research to introduce Web service technology into the Jini service architecture, thereby enabling Jini services and Web services to interoperate. Implementation details of this interoperation of services will be given in a subsequent paper.

References

1. Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana, "Web Services Description Language (WSDL) 1.1," available as a W3C note at http://www.w3c.org/TR/wsdl/, March 2001.
2. W. K. Edwards and T. Rodden, "Jini Example By Example," published by Prentice Hall, 2001.
3. Ian Foster, Carl Kesselman, Jeffrey Nick, and Steven Tuecke, "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration," January, 2002. This is available online at http://www.globus.org/research/papers/ogsa.pdf.
4. Ian Foster, Carl Kesselman, and Steven Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," The International Journal of High Performance Computing Applications, Vol. 15, No. 3, pages 200-222, Fall 2001.
5. K. Gottschalk, S. Graham, H. Kreger, and J. Snell, "Introduction to Web Services Architecture," IBM Systems Journal, Vol. 41, No. 2, 2002.
6. F. Leymann, "Web Services Flow Lauguage (WSFL 1.0)", available at http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf, May 2001.
7. S. Thatte, "XLANG: Web Services for Business Process Design." Available at http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm, 2001.
8. Yan Huang, "The Role of Jini in a Service-Oriented Architecture for Grid Computing," PhD thesis, Department of Computer Science, Cardiff University, December 2002.
9. F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, S. Weerawarana "Business Process Execution Language for Web Services", available at http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/, August 2002.