

On-the-fly Wrapping of Web Services to Support Dynamic Integration

Gerald C. Gannod*[†], Huimin Zhu and Sudhakiran V. Mudiam

Dept. of Computer Science & Engineering

Arizona State University

Box 875406

Tempe, AZ 85287-5406

E-mail: {gannod, huimin.zhu, kiranmvs}@asu.edu

Abstract

Service-oriented development has gained much attention in recent years with the advent of technologies such as Web Services, .NET, and Jini. Standards for specifying, publishing, and delivering services has enabled approaches for specifying applications that can integrate and bind at run-time with services that meet application requirements. Jini is a service-based framework that was originally developed to support integration of devices as services. Web services are services that are delivered over the World Wide Web using WSDL as a descriptor and SOAP as the delivery mechanism. In this paper, we describe a wrapping technique that facilitates the integration of web services into Jini service applications on-the-fly. As a result, an application can take advantage of run-time service lookup facilities provided by Jini without needing a priori knowledge of the web services it eventually uses.

1. Introduction

Service-oriented development has gained much attention in recent years with the advent of technologies such as web services, .NET, and Jini. Web services are services that are delivered over the World Wide Web using the Web Service Description Language (WSDL) as a descriptor. WSDL is an XML format for describing network services and operating on messages containing document or procedural oriented information. The *Universal Description, Discovery and Integration* (UDDI) registry was developed in order to support storage and delivery of web service descriptors, thus facilitating an

environment whereby components can be accessed by service clients on an as-needed basis. One of the challenges of using UDDI arises from the fact that service providers must have knowledge about the static location of a UDDI server. As such, UDDI uses a central, broker-based, method for enabling delivery of services from providers to clients.

Jini is an interconnection technology that supports dynamic registration of services whereby services locate and register with discovered lookup services using multicasting. The Jini framework for specifying, publishing, and delivering services has enabled approaches for specifying applications that can dynamically discover services that meet application requirements and integrate and bind with those services at run-time [4]. Within this framework we have demonstrated the use of legacy applications and command-line systems as services [5].

In this paper we describe an approach for specifying and wrapping web services in order to provide access to those services within the Jini interconnection framework. The wrapping approach is fully automatic, thus enabling run-time or on-the-fly generation of “glue code” for the interface between a web service and an invoking Java Jini application. The advantage of such an approach is that applications that utilize the Jini framework can apply the use of web services without modifying their invocation model. As a result, an application can incorporate run-time service lookup facilities provided by Jini when using web services. That is, the wrapped web services can be discovered and integrated at run-time by relying on architectural descriptions of high-level behavior.

The remainder of this paper is organized as follows. Section 2 discusses background material in the areas of design patterns, software architecture, Jini, and web services. Section 3 describes our approach for wrapping web services while Section 4 illustrates the overall approach with an example application that incorporates the use of several web services. Related work is discussed in

* This author supported in part by NSF CAREER Grant CCR-0133956.

[†] Contact Author.

Section 5 and, finally, conclusions and future investigations presented in Section 6.

2. Background

This section provides background material on design patterns used in our approach, Jini, and web services.

2.1. The Adapter Pattern

Re-engineering is the process of examination, understanding, and alteration of a system with the intent of implementing the system in a new form [1]. Since the functionality of the existing software has been achieved over a period of time, it must be preserved for many reasons, including providing continuity to current users of the software. One approach to re-engineering is to use the *adapter pattern* [2] whereby a legacy interface is converted into a form that a client application can utilize. The adapter pattern allows components that otherwise could not work together because of incompatible interfaces to be combined to form a new software system. In our approach the adapter pattern is used to “re-engineer” web services in order to provide access to those web service via a Jini network. Specifically, in terms of the Gamma et al. adapter pattern, we use the concept of the object adapter in the manner shown in Figure 1. Web services are adapted to new interfaces, as required by client applications, by creating an adapter around the component.

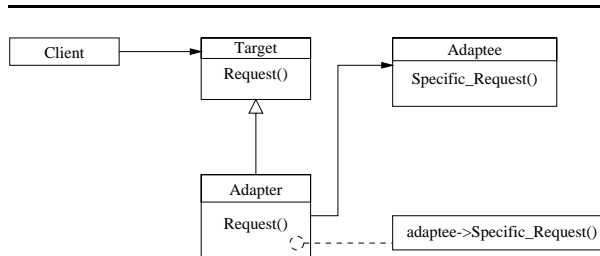


Figure 1. Object Adapter [2]

2.2. The Proxy Pattern

Proxy Pattern [2] provides a surrogate or a placeholder for another object to control access to it. In this pattern a client accesses a realSubject only via a Proxy. The proxy provides an intermediate layer between the client and the realSubject. The proxy acts as a local representative for the realSubject and typically lives in the client’s address space. It is necessary for the proxy to provide exactly the same interface as the realSubject. All

the access to the realSubject from the client have to go through the Proxy. In most cases the client is not even aware of the proxy object and assumes that it is directly talking to the realSubject. Figure 2 shows the interaction between a client and a realSubject via a proxy. In the approach described in this paper, the proxy pattern plays a central role in the definition of services as well as in the realization of service integration. The proxy talks to the service on behalf of the client that is accessing the service. The proxy becomes part of the client, and it shields the client from the details of where and how the service is implemented. This pattern allows for the client to remain oblivious to the details of the service implementation. In the approach described in this paper the use of a proxy is central to the development and construction of wrappers and interfaces.

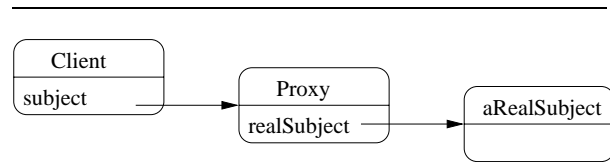


Figure 2. Proxy Pattern [2]

2.3. Jini

The primary enabling resource for the work described in this paper is the use of Jini [3] for the delivery and management of services. In a typical Jini network, services are provided by devices that are connected to the network. Figure 3 shows the layered architecture of the Jini Interconnection technology where the Jini technology layer provides distributed services for such activities as *discovery*, *lookup*, *remote event management*, *transaction management*, *service registration*, and *service leasing*. When a service is plugged into a Jini network, it is registered as a member (e.g., service) of the network by the Jini lookup service. When a service is registered, a proxy [2] is stored by the lookup service. The proxy can later be transported to the clients of the service. Network members can discover the availability of the service via the Jini lookup service. When a client application finds an appropriate service, the lookup service sets up a connection. In our approach to component integration, we use Jini to provide a standard method for registering services, and connecting a client to the software components that are acting as services.

One of the advantages of using a Jini-based integration technique is that it facilitates construction of applications “on-the-fly” whereby components can be used on an as-needed basis. One of the disadvantages is that

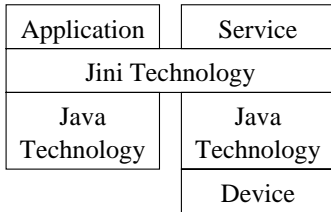


Figure 3. Jini Architecture

clients must have some prior knowledge about how to use member services.

2.4. Web Services

Web services are applications that can be accessed via HTTP. A service is a coarse-grained component that is meant to provide data and procedure-oriented behavior. A *Service Component* is a component-based view of service interactions. A service component is the generic unit of reusability for services, and may be used in different contexts. Furthermore, it captures operational features of a service in such a way that the component can be reused in different systems. Finally, service components are used to generate services through combination and adaptation of individual component behaviors.

A web service is a simple standard for software reuse over the world wide web. Web services represent a type of software component technology that can be used to implement service-based components. Web services currently present the most promising way to facilitate application-to-application integration via the Internet. The World Wide Web Consortium (W3C) has developed interoperable technologies that facilitate web interoperability and exchangeability including support for WSDL, SOAP, and XML (along with its associated family standards).

3. Approach

In this section we describe an integration approach that we have been using to support dynamic integration of software services and then present a wrapping technique that facilitates the use of existing web services in the integration framework. As part of this section, we motivate the use of a Jini lookup service as a means for locating web services rather than using a standard UDDI server.

3.1. Overview

We are developing an approach based on a new paradigm that looks at software reuse from a differ-

ent perspective in which components are viewed as services available on a network. In this paradigm, components are dynamically composed into federations to make up an application [4]. The key features of our approach within this paradigm are:

- Components of varying granularity are bundled as services and made available on a network.
- The paradigm provides an integration framework, or middleware, that allows for the dynamic integration of these components (bundled as services) at run-time.
- The paradigm facilitates the use of various patterns of interaction (architectural styles) between clients and services.
- The services provide a clear set of interfaces that are discovered dynamically at run-time and achieved using filters and adapters.

The process of component integration consists of the following steps.

1. Specification of the components as services.
2. Generation of the services along with the appropriate adapters and storing them to a repository.
3. Specification of a client to make use of services from the repository.
4. Generation of the client.
5. Execution of the client, performs the integration of the specified services at runtime.

Steps 1 and 2 are typically performed only once for each service while Steps 3 through 5 are performed as needed for each application. One of the primary goals of this work is the use of automatic synthesis to generate the source code necessary to achieve service integration. Our preliminary investigations have yielded an approach for generating wrappers of legacy command-line applications for use as services [5].

In this paper we discuss an approach for facilitating the use of web services as Jini services. Specifically, we motivate the use of a dynamic Jini lookup service over a static UDDI server and then present a method for automatically generating wrappers for web services. As a result, the integration approach addresses to problems often associated with reuse-based systems: population (where do the reusable components come from) and recall (how are reusable components located). Furthermore, we address each of the above in an automated fashion in order to reduce the overhead required to develop new applications.

Integration of all services in our paradigm is performed with the execution of clients as each service becomes available. At first a client registers with a lookup

service. Once services become available and join the network, the client is notified. The client integrates with the services by performing the GUI component integration as well as the service adapter integration and utilizes the service.

The research described above makes use of Jini [3] technology to realize service integration. The components are wrapped as Jini services and we make use of the discovery and join mechanism to enable services join a Jini network.

3.2. Jini Lookup and UDDI Servers

In this section we compare and contrast both UDDI servers and the Jini lookup service in order to place the integration approach in its proper context and to motivate the use of web services within our Jini-based framework.

3.2.1. UDDI Servers. Web services publish and discover information about services in a Universal Description, Discovery and Integration (UDDI) registry. UDDI represents the discovery layer within the web services protocol stack. The UDDI specification was created by the OASIS, a global consortium that oversees the development of standards.

UDDI contains two parts: a technical specification and a business registry. UDDI is a technical specification for creating a distributed directory of web services. It stores data in XML and includes an API for searching and publishing web services. A UDDI registry contains *White Pages*, *Yellow pages* and *Green Pages*. White pages contain generic information about companies, like business name, description and its address. Yellow pages contain industrial categorization based on standard taxonomies; this data includes industry, products, and geographic location. Green pages contain technical information about the web services that businesses provide. This information includes the address of the webservice and the specification of the webservice.

Figure 4 shows a model of how UDDI is used. A service provider publishes its WSDL specification on a UDDI registry. Subsequently, a service requestor search the UDDI registry for a desired service. This search may be manual or automated. Finally, a service requestor accesses the service directly. One property of this access is that the service may be accessed without the need to perform a UDDI search.

3.2.2. Jini Lookup. Jini's Lookup service provides services for registering and locating Jini services. Both clients and services first discover the lookup service using a discovery process. The discovery process can be either lookup service initiated or a direct discovery. The Lookup service periodically multicasts

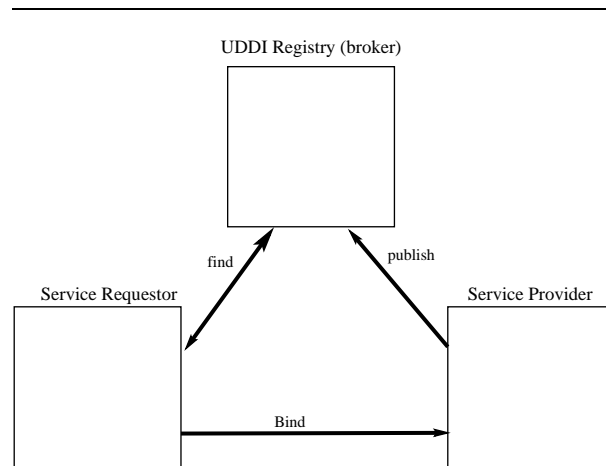


Figure 4. UDDI registry

its existence. Alternatively, services can directly connect with a lookup service if the identity and location is known.

Figure 5 shows an the model of how Jini's Lookup service is used. Once a lookup service is found, a service provider can register a service proxy along with a list of attributes describing the service. A service requestor uses a set of attributes to perform a lookup for a service provider that meets certain criteria. When found, the service proxy can be downloaded and used by the requestor to directly talk to the service provider.

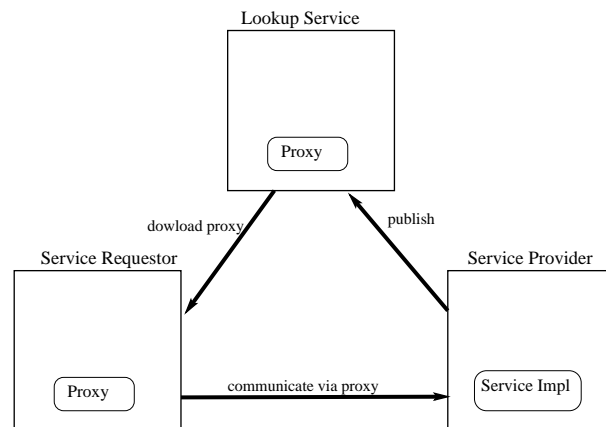


Figure 5. Jini Lookup Service

3.2.3. Comparison. Table 1 summarizes the comparison between the UDDI registry scheme and Jini's lookup service. The discovery process provided by UDDI is static, in the sense that the service provider need to know the location of the UDDI registry. In the case of Jini, the lookup service is multicasting its availability on the network, thus potential providers of services can learn of the existence of a lookup service at run-time.

The search supported by UDDI is through its white, yellow pages and green pages. The attributes in the registry include industry codes, protocols and geographic locations. The search for service providers in Jini uses Java types and attribute-based searches which can use a white, yellow, and green page scheme.

Jini's lookup service provides the ability for service providers to store a smart proxy along with service attributes. The service requestor downloads the smart proxy from the lookup service and uses that proxy to talk directly to the service provider. UDDI only provides a description of the service provider. This description contains the endpoint of the service along with the ports and the methods provided by the service provider. It is up to the service requestor to develop the access interface.

UDDI registry allows service providers to be written in any language as long as they provide a service and describe it and register with the registry in XML using the standard WSDL (Web Service Description Language). On the other hand Jini Lookup allows only service providers written in Java.

Jini's lookup service provides both leasing and events, whereby a service requester leases a service for a fixed amount of time, after which it must renew its lease. Lookup supports the ability for service requesters to be notified about a service provider when it becomes available. UDDI registry does not provide any such functionality at this time.

Both UDDI registry and the Jini Lookup services address the needs of service discovery publishing, lookup and binding. The web services model and the Jini services model can be seen as complementary technologies despite their differences. Jini's lookup service provides the possibility of a smart proxy and allows it to be downloaded, which then is used by the service requester to talk to the service provider. This shields the service requester for the location and the implementation of the service provider. On the other hand the main advantage of using the web services model and using the UDDI registry allows for platform independence, whereby the service requester and the service provider can be created in different languages.

3.2.4. Discussion. The existence of UDDI and web services provides a rich source of reusable components. The intent of the research described in this paper is to provide a bridge between Jini and web services in such a way applications within Jini can take advantage of the web service resource while also utilizing the capabilities of the Jini lookup service. From the standpoint of reengineering, the approach described in this paper is a form of adaptation that allows a "legacy" framework of sorts to utilize new technology.

3.3. Specification Requirements

The WSDL2Java translator is created for wrapping web services from a WSDL document and generating service applications either with or without a Jini network environment. This system can automatically generate Jini services and dynamic network components from WSDL documents. This section discusses issues for correctly specifying a WSDL2Java specification as well as the java code generation. In this section we describe basic issues and requirements that arise and are placed upon WSDL specifications in order to more readily support interaction and delivery of data. These issues and requirements define more concretely the nature of the data being passed to and returned from web services. These additions are all performed as extensions in WSDL and adhere directly to the WSDL standard.

The architecture of WSDL documents consists of six major elements including *types*, *messages*, *operations*, *ports*, *port types*, and *bindings*. These elements play an important role in enabling the generation of the wrappers needed to enable the use of a web service as a Jini service. Unfortunately, WSDL specifications are largely provided at an abstract level. In order to perform wrapping web services knowledge about web services need to be specified in a WSDL document. In the remainder of this section, we describe issues related to each of the elements of a WSDL specification and the resolution of those issues.

The types section provides data type definitions using type systems such as an XML Schema Definition (XSD). In our current tool, we support handling of XSD although support for other type definition languages is planned. From the XSD specification, a relational data set is generated and includes information regarding type sub-elements such as WSDL *sequence*, *choice* and *all*.

The operation and message elements of a WSDL document define the procedures and data elements that pass between clients and services, respectively. In this context, operations can be one-way or two-way (e.g., output only vs. input-output). In our approach we handle two-way operations although one-way operations can be supported with little modification.

Consider the following WSDL piece of code (POC) describing messages, a portType, and the embedded operation within the portType.

```
<message name="GetStockQuotesHttpGetIn">
  <part name="QuoteTicker" type="s:string" />
</message>
<message name="GetStockQuotesHttpGetOut">
  <part name="Body"
    element="s0:ArrayOfQuote" />
</message>

<portType name="StockQuotesHttpGet">
  <operation name="GetStockQuotes">
    <documentation>...</documentation>
    <input message="s0:GetStockQuotesHttpGetIn" />
```

Attribute	UDDI	Jini Lookup
Discovery	Static	Static and dynamic using multicast
Search	keyword based, white, yellow and green pages	Java Types, inheritance and interfaces
Attributes	Yes, industry codes and geographic locations	Yes
Proxy	No	Yes, Smart Proxy
Downloadable code	No	Yes
Language	Any	Java
Leasing	No	Yes
Events	No	Yes

Table 1. UDDI Vs Jini Lookup

```
<output message="s0:GetStockQuotesHttpGetOut" />
</operation>
</portType>
```

The format of an operation element includes the use of input and output elements. These elements refer specifically to the messages passed between respective client and server actors. Messages are typically viewed as abstract data type definitions and can be either request or response messages. In the sequence of code above, the input message `s0:GetStockQuotesHttpGetIn` within the operation tag is defined by the `GetStockQuotesHttpGetIn` message located above it. In this context, `s0`, refers to the namespace for the given message. The message itself contains a *part* or attribute called *QuoteTicker* which is expected to be a string.

In order to generate the corresponding method in a service application, the response message `s0:GetStockQuotesHttpGetOut` not only needs to be described abstractly but also needs to provide information regarding what will be returned from the server. In this case, (as with most web services), the returned data is received via a hypertext link. To more concretely describe the nature of this returned data, an extra tag can be described as a legal extension to WSDL. For example, the element below named “webLink” contains an element with its tag name “webLink” and describes, for this particular web service, the access mechanism for receiving returned data.

```
<s:element name="webLink">
<webLink>http://www.swanandmokashi.com/
HomePage/WebServices.asmx/
GetStockQuotes?QuoteTicker=</webLink>
</s:element>
```

To provide flexibility to developers, we have also provided handling for two elements: parameters and functions. The type definition `parameter` allows for the identifying a message as a parameter to an operation and allows the conversion tool to then support stronger type checking. The format of the tag is as follows.

```
<s:complexType name="parameter" />
```

where ‘s’ stands for the current XSD namespace of the WSDL document. To further support flexibility, we allow for the use of user-defined functions placed in a *Helper* class by allowing for the creation of an as follows:

```
<wsdl:complexType name=functionName />
```

where `functionName` is the user-defined operation. The generated java service will call this function after a response is returned from web server. This gives developers tremendous flexibility when using the tool to generate java services. In this context, “parameter” and “functionName” are used in request and response messages, respectively. Usage of the function defined above would appear as follows for an arbitrary WSDL specification.

```
<message name=messageName>
<part name=someName
type="namespace:functionName"/>
</message>
```

Finally, for web services where the returned data is a simple type, the following tags can be added to support storage in a variable.

```
<WSDL:element name="variableName">
<variableName>dataValue</variableName >
</WSDL:element>
```

As stated earlier, each of the extensions described above falls within the specification standards for WSDL. The addition of these facilities merely enhances the ability to generate source code that is more flexible from the end-user perspective.

3.4. Synthesis

The basic approach that we take for wrapping is based on the adapter pattern as described in Section 2. The information contained in WSDL specifications is

mapped to equivalent constructs in a generic Java interface, and specific instantiations of method names, parameters, etc., deduced from the mappings. The code synthesis supported by the WSDL2JINI tool can either be entirely self-contained or can rely on user-defined helper functions. Regardless of whether the helper functions are used or not, code generation is fully-automatic and depends only upon the input found in a WSDL specification. The generated Jini service includes a service class, a service interface, and a Jini service. The service class implements the service interface while the Jini server defines how to connect to a lookup server and register itself in the Jini network.

Table 2 summarizes the general mapping used to generate the Jini service wrappers based on WSDL specifications. As shown in the table, class and data type definitions relate to the types section in a WSDL document. Within the types section of a WSDL specification a namespace defines a contiguous environment that can be used to group a set of definitions. The WSDL2JINI tool recognizes specific data according to its namespace as well as more refined information including complexTypes or element names. In this context, the element either contains detailed information similar to that described in the previous section data types. The tool uses these value either directly or will resolve the element or data type to an expected element.

In some cases, Those elements contained in types may be frequently used by functions generated within Jini services. In these instances those elements are translated into attribute variables for a generated Jini service class. Accordingly, the variables are assigned values at run-time based on mapping information provided in a WSDL document.

The operation and message elements in a WSDL document correspond to methods in Java service classes. In a WSDL specification, each operation typically contains associated input and output elements. The WSDL2JINI tool locates the two messages types (request and response messages) and generates a corresponding method using these messages as inputs and outputs. The parameter types and name in a method call are defined based on the part elements in the request message. The returned value is generated based on the part element in the response message and either contains a complexType element or just refers to a data type. The return value can be defined through resolution to an atomic element or complexType element. Some response messages may contain a part element having a “type” attribute whose value refers to a function. For example, if the part element is defined as follows,

```
<part name= someName
      type="typens: functionName ">
```

where “typens” is the namespace of the function identified as “functionName”, the value for response message will be determined by calling the method and assigning the result based on a returned value.

3.5. Tool Architecture

Figure 6 shows the architecture of the WSDL2JINI tool. As shown in the diagram, the tool consists of a Service Generator, a Namespace Definition module, and a JINI server generator. Three files are automatically generated from the tool: Service source code, interface source, and JINI connection source. The service generator reads a WSDL document using the Apache Xerces XML parser. The Namespace Definition component defines the namespace of data types based on the parsed WSDL document, which assists in generating the data types based on namespace information. JINI server generator generates Jini service code which allows the component to discover and join a Jini lookup server, and registers the service in a lookup server. The tool can also generate a Java interface that does not utilize Jini. In that case, the JINI server generator is skipped and slightly different Interface and Service sources generated. The generated services in Java are standard classes that can be used in standard non-Jini systems.

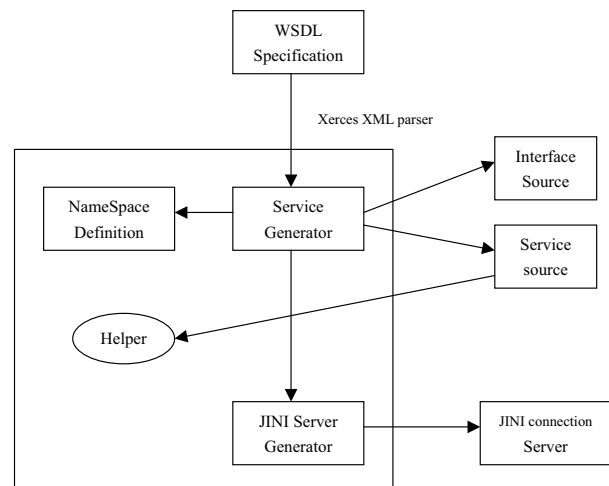


Figure 6. WSDL2JINI Tool Architecture

The Helper class is a set of routines that are incorporated into the generated Service source. One function defined in the Helper class normalizes an input to a string, thus facilitating legal parsing by web services. The Helper class also provides the interface for user defined code.

WSDL Entity	Generated Equivalent (Java)
Types	Java class and data type definitions
Elements (within complexType)	Class attributes and variables
Operation	Method
Message	Method input and output parameters

Table 2. WSDL to Jini/Java Mapping

4. Example

The example application we have developed is based on the following scenario. A user wishes to perform some research into a publicly traded company. The company itself is headed by a public figure and as such the user includes biographies or other printed material in the research. Accordingly, the application uses the Amazon search service and a stock quote service. As part of the task, the user wishes to write a document that will be included into a report and therefore requires the use of an editor and print service. Each of these services can be determined at run-time using a Jini lookup service and an appropriate architecture describing required services [4].

Figure 7 depicts the graphical user interface for the sample application that was constructed using web services that were wrapped using the WSDL2JINI tool and integrated using our architectural specification and integration toolsets. The application also includes the use of editing and printing services (shown in the top and bottom panes, respectively). The figure itself shows stock information on Walmart as well as a list of books published on its founder Sam Walton.

Figures 8 and 9 depict POCs for a WSDL specification for the stock portion of the example application and the corresponding autogenerated code, respectively. The service class name is defined based on the attribute value in “service” element from WSDL document (“StockQuotes”). A “webLink” was added to an original WSDL specification to provide a mechanism for retrieving result data. The element becomes the private variable `s0_webLink` in the generated service source code, with `s0` being the namespace of “webLink”. The method “GetStockQuotes” is created from the “GetStockQuotes” operation element in the WSDL document. The returned value comes from the “GetStockQuotesHttpGetOut” message element. This message tells the returned type is a string and returned value is obtained from the “s0:ArrayOfQuote” element contained in types element. A sequence of elements is contained in “Quote”. Thus, the returned value will be the concatenation of “webLink” and the input parameter “QuoteTicker”.

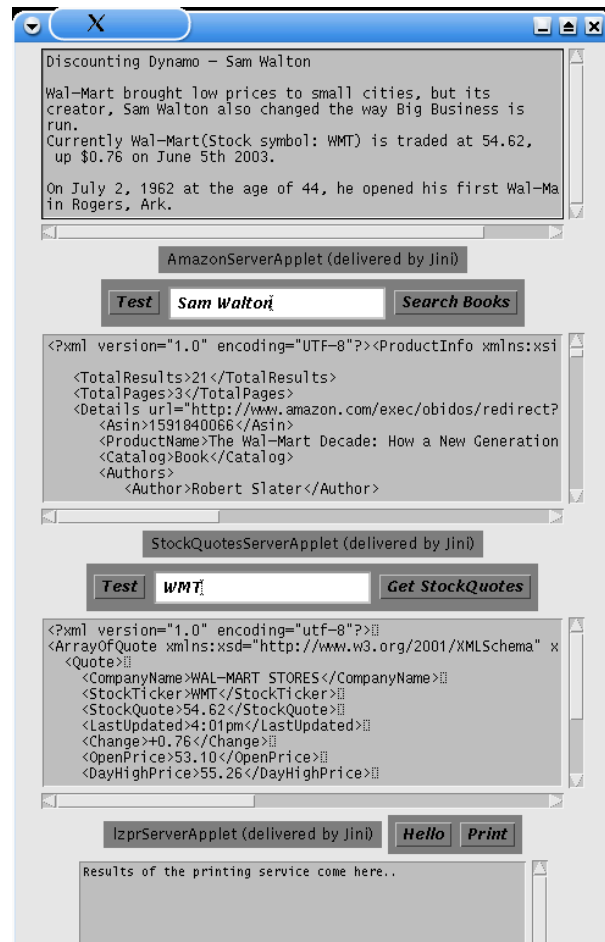


Figure 7. Stock Research Example

The sample application demonstrates the fact that once the web services are wrapped and registered with a Jini lookup server, the web services can be automatically integrated into any Jini-based application. As such, the web services act as Jini services and can be dynamically discovered at run-time by Jini applications.

5. Related Work

Cheng and Berzins [6] propose an interface wrapper model that allows distributed objects to work together

```

...
<definitions>
<types>
<s:schema elementFormDefault="qualified"
  targetNamespace="http://swanandmokashi.com/">

<s:element name="webLink">
<webLink>http://www.swanandmokashi.com/HomePage/
WebServices/StockQuotes.asmx/
GetStockQuotes?QuoteTicker=</webLink>
</s:element>

<s:complexType name="ArrayOfQuote">
<s:sequence>
<s:element minOccurs="0" maxOccurs="unbounded"
  name="Quote" type="s0:Quote" />
</s:sequence>
</s:complexType>

<s:complexType name="Quote">
<s:sequence>
<s:element minOccurs="0" maxOccurs="1"
  name="s0:webLink"/>
<s:element minOccurs="0" maxOccurs="1"
  name="QuoteTicker"
  type="s:parameter" />
</s:sequence>
</s:complexType>
...
</s:schema>
</types>

<message name="GetStockQuotesHttpGetIn">
  <part name="QuoteTicker" type="s:string"/>
</message>
<message name="GetStockQuotesHttpGetOut">
  <part name="Body"
    element="s0:ArrayOfQuote" />
</message>

<portType name="StockQuotesHttpGet">
<operation name="GetStockQuotes">
<documentation>...</documentation>
<input message="s0:GetStockQuotesHttpGetIn"/>
<output message="s0:GetStockQuotesHttpGetOut"/>
</operation>
</portType>

<service name="StockQuotes">
<port name="StockQuotesHttpGet"
  binding="s0:StockQuotesHttpGet">
...
</port>
</service>

```

Figure 8. WSDL Specification

like local objects. They also develop an automatic wrapper generator, called AIAG (Automated Interface Codes Generator). AIAG is built on top of JavaSpaces. Pawlak, Duchien, Florin and Seinturier [7] describe JAC, Java Aspect Components, a dynamic wrapping framework in Java providing a highly flexible framework with an optimization technique for executing wrapped objects. Dynamic wrappers can be used as a basis to implement a dynamic aspect-oriented systems and are able to add or remove an aspect of the application at run-time. Wohlstadter, Jackson, and Devanbu [8] developed the Cal-Aggie Wrap-O-Matic system (CAWOM) for creating CORBA wrapper for wrapping legacy systems. The wrapper generator works in a CORBA distributed envi-

```

package common;

import java.rmi.*;
public interface StockQuotesInterface
  extends Remote
{
  public String GetStockQuotes (String QuoteTicker)
    throws java.rmi.RemoteException;
  ...
}

package server;

import common.StockQuotesInterface;

import java.util.*;
import java.io.*;
import java.rmi.*;

public class StockQuotes implements
  StockQuotesInterface, java.io.Serializable {

  private String s0_webLink =
    "http://www.swanandmokashi.com/HomePage/\
WebServices/StockQuotes.asmx/\
GetStockQuotes?QuoteTicker=";

  public String GetStockQuotes (String QuoteTicker)
    throws java.rmi.RemoteException {

    String result=
      s0_webLink +
      Helper.getInputParam(QuoteTicker) ;
    return result;
  }
}

```

Figure 9. Generated Code

ronment. Finally, Sahuguet and Azavant [9] present the World Wide Web Wrapper Factory (W4F), a Java toolkit for generation of wrapper for web sources. This wrapper system is composed of three layers: the retrieval layer, the extraction layer and the mapping layer. The HTML content is obtained from a web data sources at the retrieval layer. The HTML Extraction Language (HEL) to extract the information at the extraction layer. HEL is a DOM-centric language where HTML documents are presented as a label graph. At the mapping layer, they describe the Java mapping and XML mapping.

The primary difference between the wrapping approaches described above and the one described in this paper lies in the source of the wrapped components. Specifically, in our approach we focus on the wrapping of web services and their subsequent integration in a dynamic integration framework.

Hodes, Czerwinski, Zhao, Joseph and Katz [10] present the architecture and implementation of a secure wide-area Service Discovery Service (SDS). SDS provides the method for advertising the description of available or already running services. This approach allows clients to use SDS to compose complex queries for locating the service.

Predonzani, Sillitti and Vernazza [11] present Automated Information Router (AIR), an integration archi-

ture and a tool based on the components and data-flow paradigm. The architecture is composed of an integration network, the builder for the construction of the integration networks, and a controller that manages a collection of servers for various protocols.

Lau and Ryman [12] describe the experience of the development associated with XML web services and in WebSphere. This tool supports a new and growing set of specifications, such as SOAP, WSDL, UDDI, XML and its associated family standards. The adoption of XML allows maximum flexibility and interoperability across the Internet. They develop the schema of mapping between XML and programming languages. WSDL is the standard web language used in this tool. The WebSphere Studio Application Developer completes the following tasks. 1) Discover existing web services, which is solved by using UDDI business registry that indexes web services so the service can be discovered quickly. 2) Access existing web services and compose them into new application and web services.

Each of the above integration frameworks is similar to that suggested in this paper although the Jini-based integration framework has the advantage of supporting dynamic lookup and registration and can more readily support creation of federated services.

6. Conclusions and Future Investigations

Service-oriented development is quickly gaining in popularity. As with any reuse-based approach, two problems must be addressed in order to ensure widespread usage: population of repositories with components (services) *for reuse*, and search and recall of components (services). In this paper, we have motivated and provided solutions to both issues by facilitating incorporation of web services to address the population issue and utilizing Jini lookup to address the search and recall issue. In addition, by automating the generation of wrappers for web services, we have enabled the use of technologies from competing approaches, thus allowing for interoperability without sacrificing or compromising architectural integrity of either approach.

As part of our future investigations, we intend to provide a bridge in the opposite direction (Jini services as web services) from the one described in this paper (web services as Jini services). That is, we intend to develop an approach that will allow for the delivery of Jini services as web services via publication of a service proxy over a SOAP-based interface. In addition, we are interested in developing an approach that will allow for providing a Jini-like lookup service as a web service that is based on dynamic registration. Finally, we are interested in developing a Jini service that can be used to search a UDDI server directly in order to achieve true on-the-fly

integration by wrapping web services at the point of request.

References

- [1] Elliot J. Chikofsky and James H. Cross. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, January 1990.
- [2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Longman, 1995.
- [3] W. Keith Richards. *Core Jini*. Prentice-Hall, 1999.
- [4] Gerald C. Gannod, Sudhakaran V. Mudiam, and Timothy E. Lindquist. Automated support for service-based software development and integration. *Journal of Systems and Software (Special Issue on Automated Component-Based Software Engineering) (to appear)*, 2003.
- [5] Gerald C. Gannod, Sudhakaran V. Mudiam, and Timothy E. Lindquist. An Architecture-Based Approach for Synthesizing and Integrating Adapters for Legacy Software. In *Proc. of the 7th Working Conference on Reverse Engineering*, pages 128–137. IEEE, Nov 2000.
- [6] Ngom Cheng, Valdis Berzins, Luqi, and Swapan Bhat-tacharya. Interoperability with distributed objects through java wrapper. In *Computer Software and Applications Conference (COMPSAC)*, pages 479–485, October 2000.
- [7] Renaud Pawlak, Laurence Duchien, and G'erard Florin. Dynamic wrapper: Handling the composition issue with jac. In *Technology of Object-Oriented Languages and Systems. TOOLS 39. 39th International Conference and Exhibition on*, pages 56–65, 2001.
- [8] Eric Wohlstatter, Stoney Jackson, and Premkumar Devanbu. Generating wrappers for command line programs: The cal-aggie wrap-o-matic project. In *Software Engineering, 2001. ICSE 2001. Proceedings of the 23rd International Conference on*, pages 243–252, 2001.
- [9] Arnaud Sahuguet and Fabien Azavant. Looking at the web through xml glasses. In *Proceedings of the Fourth IFCIS International Conference on Cooperative Information Systems*, pages 148–159, September 1999.
- [10] D. Hodes, Steven E. Czerwinski, Ben Y. Zhao, Anthony D. Joseph, and Randy H. Katz. An architecture for secure wide-area service discovery. *Wireless Network* 8, pages 213–230, 2002.
- [11] Paolo Predozani, Alberto Sillitti, and Tullio Vernazza. Components and data-flow applied to the integration of web services. In *Industrial Electronics Society, 2001. IECON '01. The 27th Annual Conference of the IEEE*, volume 3, pages 2204–2207, 2001.
- [12] C. Lau and A. Ryman. Developing xml web services with websphere studio application developer. *IBM System Journal*, 41(2), 2002.