Web-Based Specification and Integration of Legacy Services

Ying Zou **Kostas Kontogiannis**

Dept. of Electrical & Computer Engineering
University of Waterloo
Waterloo, ON, N2L 3G1, Canada

Abstract

With the explosive growth of the Internet, businesses of all sizes aim on applying network-wide solutions to their IT infrastructures, migrating their legacy business processes into web-based environments, and establishing their own on-line services. To facilitate process and service integration, a complete and information rich service description language, is essential for server processes to be specified and for client processes to be able to locate services that are available in Web-enabled remote servers.

Within the context of emerging technologies, such as XML, Internet, and Web-enabled application servers, we propose an architecture that allows for the migration of legacy services to distributed environments. The architecture is based on legacy component wrapping, a service description language that allows for the specification of services at higher levels of abstraction than the standard Interface Description Languages (IDLs), and on techniques that support service registration and dynamic service localization.

1 Introduction

Tremendous changes are taking place in the business world today due to the frequent occurrence of new computer technologies. Modern software systems must conform to requirements, such as flexibility, adaptability, time to market, and continued business process reengineering. Driven by these requirements, the migration and integration of legacy systems towards new platforms and operating environments provides an effective strategy for organizations to maintain their competitive edge [1]. Organizations, which integrate new development with the existing legacy systems, will have a higher success rate, and optimal cost, for the implementation of client/server applications. [1].

As Internet technologies become the mainstream, the focus of Web-based activities is shifting from information retrieval and customer-to-business transactions, to a more dynamic model [8], which integrates business services and business process models, across corporate Intranets, and the Internet. Towards this objective, multi-tier architectures, networking, and distributed object technologies have made possible for organizations to deploy complex software applications in such distributed Web-based environments.

In this paper, we present an architecture and a methodology that allow for the description, registration, and integration of existing stand-alone services into Web-based environments.

In the core of the system lies a service description language that provides standard, and well-understood information about the interfaces, and the functionality of the offered services at a higher level of abstraction than CORBA IDLs. The purpose of this work is to provide interface descriptions for software components, which are much richer in content than IDL descriptions with respect to the information they provide to a binder. In this context, XML provides a flexible and extensible formalism to represent component interfaces in a more powerful way than IDL. Component interface information can be

abstracted and presented in the XML format, without the need of specific knowledge of the inner-workings if the OMG IDL or Java JNI.

A service registration tool allows for services to be easily registered with the environment, using a service repository. Finally, a search engine can effectively locate services according to specific search criteria, allowing thus for service location transparency.

This paper is organized as follows. Section 2 provides an overview of enabling technologies. Section 3 discusses a layered architecture for Web-based service integration. Section 4 presents a service description language. Sections 5 and 6 present issues related to service registration and localization. Section 7 and 8 present applications of the proposed model. Section 9 provides a discussion and pointers to future work.

2 Enabling Technologies

2.1 Distributed Software Component

With the advances in networking, the software components [18] are moving from the desktop paradigm towards enterprise-wide, distributed, network-centric environments. In this context, a distributed component possesses the following characteristics [19]:

- An explicit and well-defined interface defining the services it provides;
- An explicit specification for describing the behavior and usage of the component, the required execution environments, and the location of the component;
- Software independence from its clients by encapsulating implementation details;
- Communication with binders for registering with the clients via the Internet, Intranets or Extranets;
- Means for remote access and remote invocation of services, and finally;
- Provision for run-time dynamic component configuration.

The major middle-ware technologies, including CORBA, RMI, Enterprise JavaBeans, and DCOM, meet the above requirements and provide the integration environment for distributed software components.

2.2 Distributing Software Component on the Web

OMG IDL is a key component of the CORBA standard and is recommended as the software interface specification due to its language, platform, and vendor independence. It supports the basic specification for distributed components, such as the operations and attributes provided by the component. However, IDL can not describe all of the information, such as, the pre/post-conditions, and semantic descriptions of functionality, of the distributed component. For example, the server component may match perfectly to the client request by the signatures in the IDL interface, but probably, the functionality of the offered service may not be the best sought by the client process. Moreover, IDLs do not provide any additional information about the server's external dependencies such as, the call-back invocation of a client's method. Although IDL is human-readable in terms of its syntax, it is a type of program level specification and can be compiled into executable code. The description for the functionality of the interface can be indicated in the comments, but it is not easy for an algorithmic process to derive such information from IDL syntax or free-style IDL comments alone.

XML is another key technology that is widely used as distributed standard format for data representation, data integration, data storage, and message exchanging. XML files can be transferred via HTTP on the Web, and provide a machine understandable and human readable syntax, which allows the developers to define their own tags in different domains [20], [21]. In this context, XML tags can be used to define the configuration information and provide the semantic specification for distributed software components. As a result, the combination of XML and CORBA IDL can facilitate the specification and localization of distributed services over Web-based environments.

3 Architecture for Web-Based Service Integration

The Web-based service integration architecture focuses on the use of the Web as an open infrastructure where services and tasks can be defined, composed, and enacted in a fully custo-

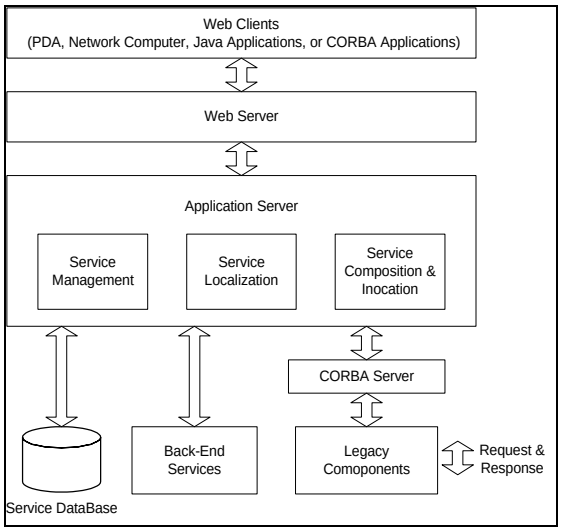

Figure 1: Overall Architecture

mizable way.

As services can be scattered virtually everywhere on the Web, we need a global infrastructure, which enables software components that have been developed independently, be integrated with each other in order to facilitate complex business tasks.

This paper proposes an open, multi-tier infrastructure, where a service can publish itself, and easily to be integrated with other legacy components and services. The proposed architecture is illustrated in Figure 1. The first layer (top) consists of a wide range of Web clients, including Web browsers for handheld and embedded systems, or Java/Pure CORBA based applications running on fully loaded desktops. The second layer relates to Web and application servers that intercept and dispatch client service requests. The application server has been widely adopted as the runtime environment of choice for integrating heterogeneous applications. The Web server captures the requests from Web clients and directs the requests to the application server. The third layer of the architecture relates to the underlying services that are added to the application server, including Service Management, Service Localization, and Service Composition & Invocation.

Service Management maintains a database of the descriptions of the available services. It enables the independently developed and deployed services to dynamically register their description information in a repository. The Service Management module provides a repository for the client processes to use in order to locate available services and compose them for the completion of elaborate business tasks. A service description language provides a customizable way to represent distributed services with enriched information.

The Service Localization module is responsible for selecting the required services among many available ones, according to the criteria set by the client process. The service localization enables the clients to search the service by functionality, signatures, performance, or customizability.

Finally, the Service Composition & Invocation module provides a framework and a scripting language for processes to be dynamically configured, in order to invoke and compose remote services. This module serves as an integrator that allows back-end services and legacy systems to be composed seamlessly.

In order to enable the integration of legacy applications in a Web-based environment, the CORBA standard is adopted. The standard allows for legacy applications to be encapsulated in remote objects using wrapper classes and behave as distributed components. This wrapping technology allows clients in virtually any software or hardware platform to invoke remote legacy components in their native operating environments. The legacy application is resident on the CORBA server, which acts as the gateway for the integration. Such a framework provides system support for the invocation and integration of legacy back-end services from any remote client.

3.1 A Layered Service Integration Model

The three-tier architecture (shown in Figure 2) is instrumental for the implementation of the proposed architecture. From an implementation point of view, the standard three tier-architecture can be seen as a composition of four layers [11] namely: *a)* presentation layer (Web client); *b)* content layer (Web server); *c)* application layer (application server); and *d)* back-end data and services layer. The Web server is responsible to accept HTTP requests from Web clients, and deliver them to the application server, while the application server is in charge of locating the services and returning responses back to the Web server. On the other hand, a thin client in the pre-

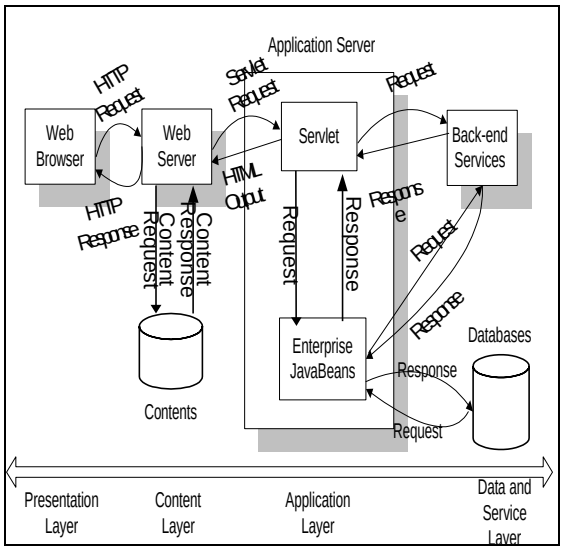

Figure 2: Control Flows in Three-Tier Architecture

sentation layer has little or no application logic. It sends the request via HTTP to the web server for an interactive view of the static and dynamic information (Figure 2).

To provide the static information required, the Web server can maintain a content repository, or a file system, where the information-based resources are stored and serve as static HTML pages. Upon receiving a request from the client, the Web server retrieves the requested document from the content repository and sends it to the client. In this case, the client entirely relies on the Web server. Programming languages, such as Java, and scripting languages, like CGI, can be used to access the database.

To provide the dynamic information generated by software services, the Web server needs to constantly interact with the application server. A servlet provides the dynamic HTML content to clients. When the Web server receives the request for a servlet, it re-directs the client's request along with the parameters to the application server, which loads the servlet and runs it. Servlets not only have all the features of Java like automatic memory management, advanced networking, multithreading, and so forth, but also enterprise connectivity in the form of JNI (Java Native Interface), JDBC, EJBs, RMI, and CORBA. Servlets can initiate invocations to back-end services, other servlets, or to the Enterprise JavaBeans [12].

Once the servlets are deployed on an application server, they can be accessed from any other Web server. This can be achieved provided that the HTML client page contains the URL of the servlet with the correct name, type, parameters, and initial values embedded in the HTML <form> tags. The combination of EJBs and Servlets, CORBA objects and Servlets, and RMI objects and Servlets, can be used to invoke back-end services accessed by the Web thin clients via HTTP connections.

However, CORBA and Enterprise Java Beans are not a panacea for all problems that may arise when integrating services in a distributed environment. Nevertheless, they serve as valuable building blocks for implementing, deploying, and integrating applications over a diverse range of platforms and operating environments.

3.2 Legacy Services

Within the context of legacy software re-engineering, it is important as a first step, to analyze the legacy software in order to recover its major functional components that may implement specific business logic. Reverse engineering techniques that allow for the identification of such components have been investigated in depth by the research community and have been also presented in other related projects within IBM CAS [13, 14]. Once specific legacy components have been identified through the use of program analysis, their behavior can be specified in terms of well-defined object oriented interfaces and wrappers. The primary reason for using wrappers is to shorten the "time-to-market" in migrating components from centralized systems to the Web, and ease the distributed component integration process. In the context of our research, we adopt the wrapping technology in order to assist the migration of the identified components into a heterogeneous Web-enabled distributed environment. We select CORBA as the infrastructure environment of choice. A CORBA object wrapper acts as an interface adapter, which adapts the identified services into the proposed architecture and provides a consistent interface to the registered services and other identified legacy components. In addition, the object wrapper can be applied to make the necessary functionality of a legacy service available to remote clients (i.e. by automatically registering its description into a service repository).

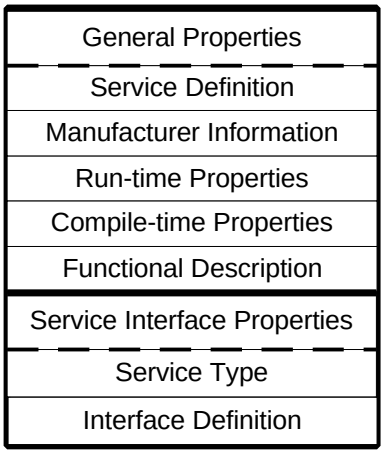

Figure 3: Key Elements of Service Specification

4 Service Description Language

In this section, we present a prototype service description language that provides a standard format to represent, register, and store information related to remote services. To facilitate the integration of back-end services, a meta level description language is essential to effectively locate registered services. The meta-level description of the software services can be published at the same time as the distributed objects are deployed onto the application servers, or some time later when the enterprise would like to make its software services available.

4.1 Structure of Service Description Language

Generally, a service can be represented in a multi-faceted way, by specifying, for example, vendor, run time requirements, compile time requirements, method signatures, as well as, pre- and post-conditions. Each of these aspects is denoted as a Service Description Fact. Different Description Facts specify different properties of the services.

In our work, the specification of the software services is divided into two layers: General Properties and Service Interface Properties as illustrated in Figure 3. Each layer contains specific information at different levels of abstraction.

To enable a service binder to locate the requested service with high precision and recall levels, the General Properties should contain facts that relate to such aspects as general service definition, manufacture information, run-time and compile-time properties, signatures, version numbers, implementation language, and functional descriptions. For example, for a CORBA wrapped service object, it is important to specify the ORB agent address, which is responsible for invoking the requested CORBA object by the name and URL address of the object.

The Functional Description Fact category provides the syntactic properties, describing the service's constraints, features, and functionality. It includes both abstract and detailed service descriptions. The abstract description provides high-level service descriptions in terms of keywords, types, and signatures.

For the purpose of Web-based service integration, it is important to disclose the interface of the distributed components (Web service) to client processes. Similarly, the Service Interface layer specifies the interface of the registered components. As stated earlier, there are different technologies to implement the back-end services, such as servlets, EJBs, and CORBA. Each type of back-end services can be registered by its own specific interface description. For making servlets available, to Web clients, the inputs can be embedded in HTML forms, which contain the specifications of input data types as well as, the names of parameters and their allowable values. An example service description template is illustrated in Figure 4 under the <ServletML> XML tag. For the EJBs, the back-end services can be composed of several beans (session beans, or entity beans) in one jar file. Each bean has its own home interface and remote interface. When a service is implemented by the CORBA standard, it may include several CORBA IDL interfaces as encapsulated in the CORBA IDL "module" name scope. For the interface within CORBA and EJB components, it is necessary to declare the available methods, parameters, types of method parameters, and return values. To reduce the complexity in definition of service description language, we can inherit the interface from EJBs and CORBA IDL by inserting them under the <EJBML> tag and <CORBAML> tag respectively.

Such specifications supply rich information that describe the characteristics of the available software services, and make it easier for client

```
<?xml verstion="1.0"?>
<SDL>
<GeneralInfo>
  <ServiceDef>
    <ServiceName> <!-- Specify service ID and name -- >
    </ServiceName>
    <ServiceCatalog><!--Specify service category-->
    </ServiceCatalog>
    <URL > <!-- Specify service URL linke -->
    </URL>
    <VersionNumber /> <!--Specify version number-->
  </ServiceDef>
  <Manufacturer> <!-- Specify vendor information-->
  </Manufacturer>
  <RunTimeEnv> <!-- Specify run-time environments-->
     <OSs>
          <OS name="" version=""/>
     </OSs>
  </RunTimeEnv>
  <CompileTimeEnv> <!-- Specify compile time environments-->
  </CompileTimeEnv>
  <Funcationality>
          <!-- Specify abstract and detailed information -->
          <!-- about service funcationality-->
  </Functionality>
</GeneralInfo>
<ServiceInterface>
  <Types>
  <!--Lists the Types of components inside the service interface. -->
  </Types>
  <ServletML>
  <!--Lists the servlet interface -->
     <Parameters>
          <Parameter>
              <Name /><Type /><Value />
          </Parameter>
     </Parameters>
  </ServletML>
  <EJBML> <!--Specify the EJBs interface -->
  </EJBML>
  <CORBAML> <!--Specify the CORBA interface -->
  </CORBAML>
</ServiceInterface>
</SDL>
```

Figure 4: Overall Structure of Service Description Language

processes to locate, understand and invoke the various services. XML provides a natural way to represent the specification of software services using the hierarchy relationship inherent in its tags. It's also important to follow the same template or format to express this information, so that registered services and tools can inter-operate using the same application language. Each service description fact has each own DTD. An example Service Description template is illustrated in Figure 4, where information such as service name, service ids, possible URL locations, functionality descriptions, and component signatures is denoted.

4.2 Extensible Service Description Language

The extensibility of the service description language is crucial in order to represent hetero-

```
<?xml version="1.0"?>
<!ELEMENT newTags (newTag)+>
<!ELEMENT newTag (startingPoint, tagDef)>
<!ELEMENT startingPoint (#PCDATA)>
<!ELEMENT tagDef
    (tagName, attList*, containedTags*)>
<!ELEMENT tagName (#PCDATA, tagContent*)>
<!ELEMENT attrList (attr)+>
<!ELEMENT tagContent (#PCDATA)>
<!ELEMENT containedTags (tagDef+, group)>
<!ELEMENT group (group* | tagName*)>
<!ATTLIST group groupName CDATA #REQUIRED>
<!ATTLIST group groupType (SEQ|OR) #IMPLIED >
<!ATTLIST group groupOccurs
          (once|optonal|requried) #IMPLIED>
<!ATTLIST tagDef occurs
          (once|optional|required) #REQUIRED>
<!ATTLIST attr attrName CDATA #REQUIRED>
<!ATTLIST attr attrType CDATA #REQUIRED>
<!ATTLIST attr attrValue CDATA #IMPLIED>
```

Figure 5: A DTD Specification for Adding New Service Descriptions and Content.

geneous services in a customizable way. For example, new service categories can be added or existing service descriptions can be modified. Figure 4 illustrates a collection of default facts that specify a service. The prototype defines a DTD, which is used to extend the service description. The DTD specifies the syntax for adding new facts or modifying existing facts (i.e. service descriptions). Such a DTD specification is illustrated in Figure 5. New Service Description facts can be added by using the *newTag* element contained in the *newTags* element.

With the fact specification DTD, the addition of new facts is uniquely identified and inserted in a way that maintains the syntactic validity of the description. In Figure 6, the addition of a Run-time environment fact is illustrated. In this example, the new fact is inserted under the *GeneralInfo* element with the tag name of *RunTimeEnv*. *RunTimeEnv* element can occur once under *GeneralInfo* element. It contains an *OSs* element, which specifies the operating systems that can deploy the service. The *OSs* element can occur once or more times, and it can have sub-tag *OS* element. *OS* element has the attributes such as its name and the version number of each required operating system. As specified, *OS* elements can occur one or more times in the description document, as specified by the corresponding DTD.

```
<?xml version="1.0"?>
<!DOCTYPE newTags SYSTEM "patSpec.dtd">
<newTags>
  <newTag>
    <startingPoint>SDL.GeneralInfo</startingPoint>
      <tagDef occurs="once">
        <tagName>RunTimeEnv</tagName>
        <containedTags>
            <tagDef occurs="required">
              <tagName>OSs</tagName>
              <containedTags>
                <tagDef occurs="required">
                  <tagName> OS </tagName>
                  <attList>
                    <att attrName="name" attrType="CDATA" />
                   <att attrName="version" attrType="CDATA" />
                  </attList>
                </tagDef>
              </containedTags>
            </tagDef>
            </containedTags>
      </tagDef>
  </newTag>
</newTags>
```

Figure 6: Run Time Environment Fact Definition.

Similarly, new content can be easily introduced into existing service descriptions, under the tag <ServiceCatalog> (Figure 4).

4.3 Service Repository

Within the context of this paper, service descriptions and fact specifications (DTDs) require a database for the persistent storage of the XML encoded service interface description. To keep the database management simple and to achieve flexibility in the service description, we use one table to index the service ID and the corresponding XML service description. Each fact of service description is stored in a table. The primary key of these tables is a service ID generated during service registration. The DTD of each fact is stored in the DTD repository within the same database.

The Service Management module (shown in the Figure 1) is responsible for maintaining the service database. It can insert a new service description, delete, and modify an existing one. For this work, we use the IBM DB2 XML extender to map XML Service Descriptions to DB2 tables. In general, the service manager retrieves from the database table the whole XML document by using traditional SQL queries. When a service is registered, the service manager can check for duplicate definitions, generate the service ID, and insert the description into the database. The Service Management Module is implemented by Enterprise Java Beans, which provide the necessary support for distributed transactions.

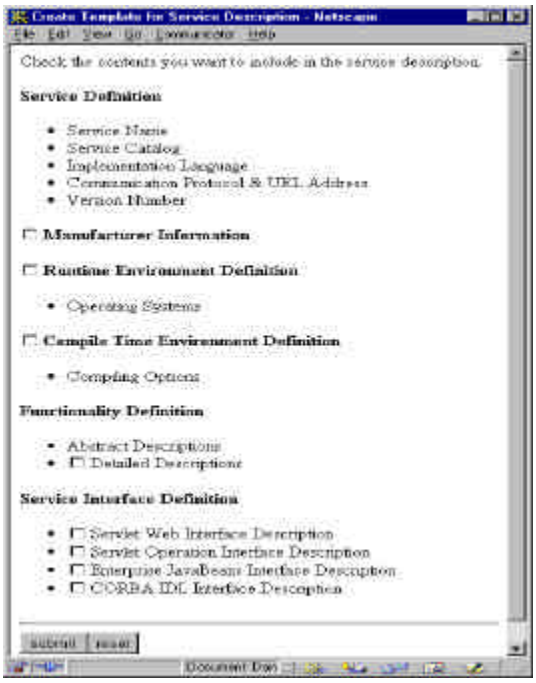

Figure 7: Service Specification Fact List

5 Service Registration

For the service registration, we have designed a Web based interface to serve as a service registration and authoring tool. The interface allows for the user to specify the service description by filling in forms in an HTML Web interface, as shown in Figure 7. The service description, conforming to the DTD in Figure 5, is generated automatically from the information provided by the user.

As mentioned earlier, in order to provide maximum customizability, the service description language is separated into independent facts. Moreover, the environment allows for new facts and new content to be added at anytime. The user interface is generated dynamically according to available facts and the DTD of each fact. This interface allows the user to select the required facts by filling the generated forms. Some facts are indispensable for newly created service descri-

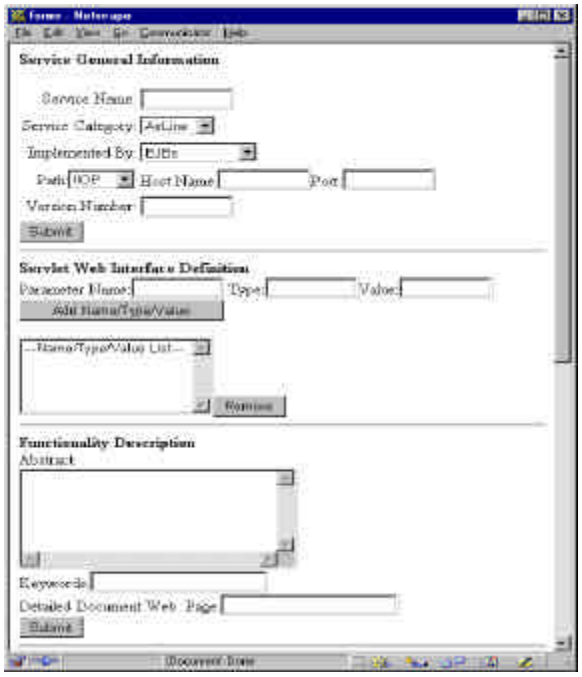

Figure 8: Service Description Interface

ptions, such as Service Definition. After submission, the Web Interface will create an HTML form as shown in Figure 8, where the user can add more information about the newly registered service.

6 Service Localization

A prototype service localization mechanism has also been designed to locate distributed software services much like a search engine locates content (data) in Web pages.

The system can provide two ways for clients to localize services, either via a Web HTML Interface, or dynamically, at run-time via an XML formatted document.

The design of the query language aims to provide the user as many features as possible for specifying the services being sought. Currently, the Description Facts offered (shown in Figure 4) allow the user to search for services according to specific criteria such as service categories, functionality, implementation techniques, and operating platforms. The grammar for the query language is defined in terms of a DTD as shown in Figure 9. The root element for this DTD is the *searchSpec* element, which can include zero or more children, such as service ID, service location

```
<?xml version="1.0" ?>
<!ELEMENT searchSpec (ID*, location*, category*, implBy*, platforms*,
funcDsc*, vendor*, version*, timeLimit*)>
<!ELEMENT ID (#PCDATA)>
<!ELEMENT location (#PCDATA)>
<!ELEMENT category ((keyword, (AND | OR)*)+ | (NOT, keyword)*)>
<!ELEMENT implBy ((keyword, (AND | OR)*)+ | (NOT, keyword)*)>
<!ELEMENT platforms ((keyword, (AND | OR)*)+ | (NOT, keyword)*)>
<!ELEMENT funcDsc ((keyword, (AND | OR)*)+ | (NOT, keyword)*)>
<!ELEMENT vendor (#PCDATA)>
<!ELEMENT version (#PCDATA)>
<!ELEMENT timeLimit (#PCDATA)>
<!ELEMENT keyword (#PCDATA)>
<!ELEMENT AND (keyword | AND | OR | NOT)*>
<!ELEMENT OR (keyword | AND | OR | NOT)*>
<!ELEMENT NOT (keyword | AND | OR | NOT)*>
```

Figure 9: DTD for Service Localization Query

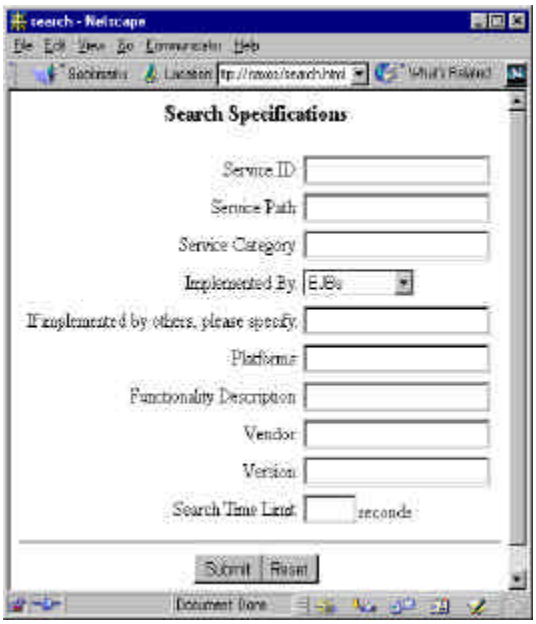

Figure 10: Web Interface for Search Query

and service category. The Web interface of the service search engine is designed for the HTTP users, as shown in Figure 10. After submission, the search criteria composed in the XML format is sent to the Service Localization module.

By extracting the requirements from the query specification, the service localization module looks up the service database. If multiple results meet the search criteria, the available services can be listed and ranked according to performance, response time from the server, or cost. Since service description is encoded in XML, the search locator has been implemented using the XML DOM (Document Object Model)

API and incorporating search logistics, such as exact search, sub-string search, precedence search, and stemming.

7 Application Example

Once a software component has been extracted from a legacy system, or has been built as a new component, its interface can be extracted and represented in XML as illustrated in Figure 11. The CORBA IDL and wrapper generators can be applied to automatically generate the service wrapper from the XML-based specification. A CORBA server source file is responsible to register the distributed objects with the CORBA ORB agent. At the same time, it can automatically send its description information in XML form to the service management module [22] via CORBA IIOP.

The generated CORBA IDL, wrapper files, and server files can be compiled into binary executable code. The CORBA objects are running on the CORBA server and waiting for the clients' requests. The CORBA objects can accept the requests via CORBA IIOP. IBM WebSphere Application Server Enterprise Edition incorporates Component Broker, a C++ ORB implementation, and provides the integration environment for CORBA objects, Enterprise JavaBeans, and Java Servlet [23].

When the CORBA client object sends its service request to the service repository which acts as a binder, the service can be located according to its formal (i.e. signature, version number, service id), or informal (i.e. performance, service cost, network latency to access the service) characteristics. The clients can search via the binder for available services via the service localization interface and obtain detailed information about the requested service. In the current state of implementation, the Web integration architecture allows the CORBA objects to be invoked directly via TCP/IP. To invoke the distributed object directly through a CORBA agent, the client needs to install the required run-time environments. By installing an ORB implementation in the local machine, the client process can send requests to the remote CORBA object by the DII [24] with the IDL repository ID and the remote ORB agent‘s URL address.

Future steps of our work will focus on accepting servlet requests on behalf of HTTP users, and provide a full Web integration environment for building virtual agencies that can integrate Web-based services dynamically.

7.1 Representation of Legacy Services

To wrap a legacy service, the first step is to determine its interface. The interface description allows the implementation details inside the component to be hidden from its external clients. Moreover, the interface defines a set of properties and behaviors that represent a component's API. Properties of interface are represented in terms of attributes, which can be accessed by accessors and mutators.

In general, the interface of a software component may contain the information related to data types, references to external specifications that point to related components, descriptions of public attributes and methods, and return types and parameters that specify input and output.
The representation of a component interface can be independent of a specific programming language, making thus possible for wrapping technology to integrate software components into heterogeneous operating platforms and environments. OMG IDL provides such a language-independent interface description for distributed CORBA components. IDL compilers can be used to generate proxies and skeleton stubs. However, in order to generate automatically wrappers that encapsulate back end-services, a specialized IDL compiler is required.

The XML interface representation of a software component consists of several aggregated elements, such as data type definitions, interface operation definitions, and interface data member definitions. The data type definition publishes the types other than the defined interface for the purpose of invoking interface methods. The interface data member definition declares the accessor and mutator methods associated with a data member. Such an example specification for a component extracted from the AVL GNU tree libraries, is illustrated in Figure 11. Such specification aims on automatic generation of OMG IDL and CORBA wrapper. This specification became part of the service interface properties under <CORBAML> in Figure 7. Furthermore, with the XML specification, additional information can be easily

Figure 11: XML Representation of Component Interface

encoded, such as the self-description information, URL address, pre- and post-conditions, and performance characteristics for the component. Such information introduces new functionality to wrappers, and can be used at the run-time by a binder to efficiently locate a service in a transparent way to the client.

7.2 CORBA Object Wrapping

As a software migration case study we considered the migration of the AVL GNU tree libraries. The new migrant AVL tree libraries [13], [14] can be considered as a collection of distributed components, that consist of several classes. Therefore, the interface for the AVL tree component consists of several sub interfaces that correspond to wrapper classes. The wrappers implement message passing between the calling and the called objects, and redirect method invocations to the actual component services. The concrete process to accomplish wrapping is accomplished in terms of three major steps.

```
module AVL{

interface corba_ubi_btRoot;
interface corba_ubi_btNode;
interface corba_SampleRec;

typedef char corba_ubi_trBool;

interface corba_SampleRec{
        void putName(in string val);
        string getName();
        void putNode(in corba_ubi_btNode val);
        corba_ubi_btNode getNode();
        long getDataCount();
        void putDataCount(in long aval);
};

interface corba_ubi_btNode {
        void putBalance(in char val);
        char getBalance();
        long Validate();
        //.....
};

interface corba_ubi_btRoot{
        corba_ubi_trBool ubi_avlInsert(
                in corba_ubi_btNode  NewNode,
                in corba_SampleRec  ItemPtr,
                in corba_ubi_btNode OldNode );
        // .....
};
};
```

Figure 12: AVL Component Interface Definition

```
class wrapper_ubi_btRoot :
        public _sk_AVL :: _sk_corba_ubi_btRoot
{
private:
        ubi_btRoot& _ref;
        CORBA::Boolean _rel_flag;
        char *_obj_name;
public:
wrapper_ubi_btRoot(
        ubi_btRoot& _t,
        const char *obj_name=(char*)NULL,
        CORBA::Boolean _r_f=0):_ref(_t);
~wrapper_ubi_btRoot();
ubi_btRoot* transIDLToObj(AVL::corba_ubi_btRoot_ptr obj);
void putRoot(AVL::corba_ubi_btNode_ptr val);
AVL::corba_ubi_btNode_ptr getRoot();
AVL::corba_ubi_trBool ubi_avlInsert(
        AVL::corba_ubi_btNode_ptr NewNode,
        AVL::corba_SampleRec_ptr ItemPtr,
        AVL::corba_ubi_btNode_ptr OldNode);
void Prune();
//Other methods are eliminated.
};
```

Figure 13: An Example Wrapper Class

The first step focuses on the specification of components in CORBA IDL as shown in Figure 12.

The second step deals with the CORBA IDL compiler to translate the given IDL specification

into a language specific (e.g. C++), client-side stub classes and server-side skeleton classes. Client stub classes and server skeleton classes are generated automatically from the corresponding IDL specification. The client stub classes are proxies that allow a request invocation to be made via a normal local method call. Server-side skeleton classes allow a request invocation received by the server to be dispatched to the appropriate server-side object. The operations registered in the interface become pure virtual functions in the skeleton class.

The third step focuses on wrapper classes that are generated and implemented as CORBA objects, directly inheriting from the skeleton classes. The wrapper classes encapsulate the standalone C++ object by reference, and incarnate the virtual functions by redirecting them to the encapsulated C++ class methods. The new functionality of the legacy object can be added in the wrapper class as long as the method name is registered in the interface. The wrapper class can be generated automatically, by the proposed high-level XML-based interface description.

For example, the ubi_btRoot class is one of the classes identified within the AVL tree component. The wrapper_ubi_btRoot inherits from the skeleton class sk_AVL::_sk_corba_ubi_btRoot, which is generated from the CORBA IDL to C++ compiler (Figure 13). The wrapper class, wrapper_ubi_btRoot, encapsulates a reference of ubi_btRoot class as shown in Figure 14.

When a client invokes a method through CORBA, it passes the parameters with data types registered in the IDL interface to the server side object. The wrapper classes need to translate the CORBA IDL specific data types from the client calls to the data types used by the encapsulated C++ classes. Figure 14 illustrates the transformation from the CORBA specific type such as corba_ubi_btRoot_ptr to the ubi_btRoot used in the corresponding C++ method. In the same way, the wrapper classes convert the returned values from the C++ class to the CORBA IDL specific data type, which is the wrapper class. In such a way, the wrapper object not only allows the client send requests to the legacy object, but also allows for the legacy object to return its result back to the clients. Since IDL does not support overloading and polymorphism, each method and data field within the interface should have a unique identifier. If the polymorphic and overloaded methods occur in one class, it is necessary to re-

```
ubi_btRoot* wrapper_ubi_btRoot :: transIDLToObj(
            AVL::corba_ubi_btRoot_ptr obj)
{
 if (CORBA::is_nil(obj)) return NULL;

// set up the data members of _ref object.
// these data members are primary data types.
_ref.putCount(obj->getCount());
_ref.putFlags(obj->getFlags());

//translate the ubi_btNode to corba_ubi_btNode_ptr
//by wrapper class rootWrap
ubi_btNode *rootImpl= new ubi_btNode();
if (rootImpl==NULL)return NULL;
wrapper_ubi_btNode rootWrap(*rootImpl,
                        _obj_name);

//translate corba_ubi_btNode_ptr type returned from
//obj->getNode() to ubi_btNode * by transIDLToObj()
// in wrapper object rootWrap.
_ref.putRoot(rootWrap.transIDLToObj(
                        obj->getRoot()));

//.....
  return &_ref;
}
```

Figure 14: Example for Object Type Translation

name these methods by adding the prefix or suffix to the original name when they are registered in the interface, avoiding changing the identified objects. This "naming" technique allows unique naming conventions throughout the system, without violating code style standards. The wrapper classes are responsible to direct the renamed overloaded and polymorphic methods to the corresponding client code.

If the polymorphic and overloaded methods occur in the inheritance relationship, we can take advantage of C++ upcast feature, that is, to only register the sub-class in the component interface, and upcast the sub-class to its super class when the polymorphic or overloading methods in a super class are invoked.

7.3 Automatic Generation of OMG IDL and CORBA Wrapper

Once the interface is defined, the process of generating CORBA wrappers is similar to each of identified components. Therefore, automatic generation of wrappers and IDL specifications is feasible by providing the information about each interface, such as, signatures, renamed operation name for the overloaded method.

C++ to IDL Data Type Mapping

In the XML interface specification, the signatures are indicated in terms of C++ types. Specifically, the IDL generator module, reads the XML script and converts the interface descriptions to IDL types, conforming to the mapping rules of the OMG IDL specification, and writes IDL style interface in a new file.

The complete mapping of basic data types is given in [17]. For example, C++ "long" is mapped into IDL "long", C++ "char *" to IDL "string". Due to the platform independence of OMG IDL, some of basic C++ data types are not specified, such as "int", which is 16 bits in MS-DOS and Win3.1, but is 32 bits in Win32. For the identified component, we assume it works under a 32 bit operating system.

CORBA C++ to Native C++ Data Type Mapping

CORBA C++ stubs are generated from IDL compilers. Usually the CORBA C++ stubs are prefixed with the CORBA namespace. To invoke the methods in the C++ code, parameters are passed from CORBA C++ wrapper objects to native C++ classes. Since the primary data types are passed by value, the parameters can be directly passed into native C++ operations, according to the same mapping rules from C++ to IDL. For the complex data types, such as class, string, array, the parameters are passed by reference. The signatures of such types in wrapper classes are defined by pointers. In this case, the IDLToObj() method is added into to CORBA wrapper object and accomplish the translation from a CORBA C++ wrapper pointer to a native C++ class pointer.

IDL and Wrapper Code Generation

According to the XML component interface representation, the wrapper IDL interface is generated, by denoting the <Interface> tag in the XML specification (Figure 11). Then, externally visible operation identifiers are extracted from the children elements under the <Operation> elements including operation name, return type and parameters' names and their corresponding types. Meanwhile, C++ to IDL type conversion is automatically performed. In addition, the type definition information indicated under <typedef> element is added on the top of the IDL file. Other global variables are defined as well. The result IDL interface is shown in Figure 12.

The CORBA wrapper header file, which declares a wrapper class in CORBA C++, is created in the same way. The signatures of parameters are mapped from CORBA C++ types to native C++ types. The operations in the IDL interface correspond to the public methods in the CORBA wrapper. Moreover, methods as well as, class constructor and destructor specifications along with their body code are added into the class interface declaration, as shown in Figure 13.

The wrapper function bodies are generated in a separated file, which provides the implementation to each of methods. According to the "srcClass" attribute (Figure 11) value in the <Operation> tag and "srcMethod" attribute value in <OpName> tag, the wrapper re-directs the invocation of its public operations to the corresponding source methods in the encapsulated source C++ object. At the same time, the wrapper generator provides the "connecting" code, which is responsible to translate the CORBA IDL specific data type to the native C++ data type, and vice versa.

Most of CORBA IDL compilers provide the generation of a "tie" class from the IDL specification. Tie classes behave as wrappers, which simply call the corresponding methods in the encapsulated class. Moreover, they do not provide any functionality to translate the CORBA IDL specific data type to the native C++ data type. The code only works, when the parameters are basic data types. For a complex data type, they can not convert it to a native C++ pointer that can be recognized by the legacy code. In this context, the tie class is too simple to address the requirements for legacy code wrapping. The proposed XML-based interface description of the software components, and its corresponding wrapper generation module overcome this problem.

Essentially, the wrapper acts as a façade: it offers clients a single, simple interface to the underlying objects. It glues together the CORBA distributed capabilities and the standalone components. Moreover, new functionality can be added to the wrapper based on the XML representation (i.e. emitting run-time information and getting the location of a component in a transparent to the client way). It is notable that the object wrappers are housed within the CORBA

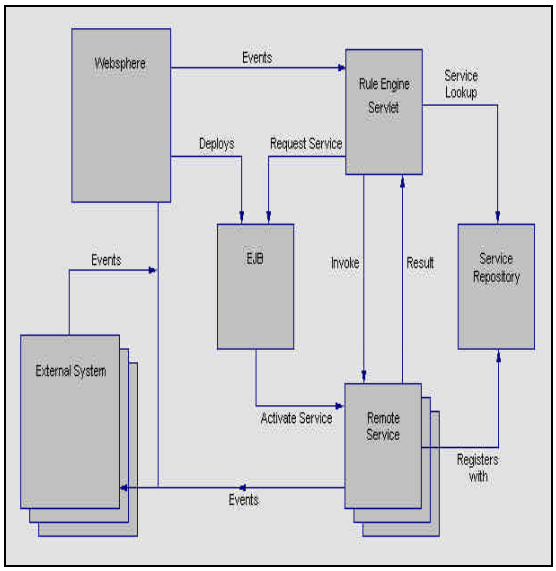

Figure 15: Overall Service Composition Architecture

server infrastructure, providing location transparency for the back-end services.

8 Customizing the Service Integration Process

In this section, we discuss how the proposed system fits within the context of a larger system that allows for service integration and process enactment. In particular, the system presented here relates to the Service Repository and Remote Service modules as illustrated in the service composition architecture diagram depicted in Figure 15. This service composition architecture is based on the component wrapping, a script-based transaction language, and a script enactment engine [25].

In this way, distributed components located virtually everywhere in the world, can be combined on as required basis, forming thus collaborative systems. This integration can occur dynamically by allowing general service properties, functionality and, signatures of components be specified in the service description language, and be registered with the service repository and the binder. Client processes can search via the binder, for available distributed components in a same manner as a search engine would be used to locate information resources on the Internet. After meeting the search requirements, the client process can invoke the identified services without necessarily downloading all the components to the local client machine. On-going work [25] focuses on the invocation of services that is based on scripts encoded in XML, and is enacted using the Event-Condition-Action (ECA) paradigm [15], by which the multiple requests can be dynamically customized in the client side using a rule-based approach. The overall proposed architecture is under development in collaboration with IBM Canada, Center for Advanced Studies, and is illustrated in Figure 15. The core of the system is the Rule Engine Servlet, which accepts triggering events from the Web server. Once the premises (events and conditions) of specific ECA rules are satisfied, the requested service (action) by the rule is localized and invoked. Upon completion, services (actions) produce new events that may trigger new ECA rules. Potential deadlocks and loops are detected by building a rule dependency graph for a given script [16].

The customization of the transaction and integration logic required by various processes to complete complex tasks, open new opportunities in Web-enabled e-Commerce and e-Business environments. In this sense, business partners can customize their business transaction process models to fit specific needs or, specific contract requirements. This customization is transparent to third parties and, provides means to complete business transactions accurately and on-time. Organizations can enter the e-Business area by building and deploying extensible and customizable services over the Internet using existing software components that are readily available as services over the Internet. Moreover, virtual agencies that provide a wide range of services can be formed by integrating existing functionality and content over the Web. For example, a virtual travel agency can be formed, by composing in a customized manner, services that are readily available in various travel related Internet Web sites. Moreover, client processes may post requests to the virtual agency. The agency can enact its transaction logic (scripts) in order to integrate and compose data and services from a wide spectrum of sites. In this scenario, data about pricing, availability and, travel related special offers, can be fetched by various sites, processed by the agency and presented to the client in a customized and competitive for the agency way.

9 Conclusion

In this paper, we presented a prototype system that allows for the specification and the migration of software services into distributed Web-based environments. With the aid of a service description language, automatic CORBA wrapper generation, service registration, and service localization can be achieved in a transparent to the client way.

In this context, we are especially interested in Web-based platforms because the Web is becoming the common denominator for accessing and presenting information over the Internet, Intranets and, Extranets. Moreover, the Web provides the deployment platform for many new enabling technologies such as CORBA, RMI and, EJBs.

As a result, this web-based service integration infrastructure allows for the reuse of the existing software components, shortens the time to architect new applications, and eases the enterprise integration of business operations. This prototype system is currently under development at IBM Canada, Center for Advanced Studies.

Acknowledgments

We would like to thank Bill O'Farrel, Steven Perelgut and Joe Wigglesworth of IBM CAS, Evan Mamas and Richard Gregory of the University of Waterloo as well as, all the anonymous reviewers for their valuable suggestions, comments, and insights.

About the Authors

Ying Zou is a Ph.D candidate at the Electrical and Computer Engineering Department, University of Waterloo. Her research interests include distributed object technology, software re-engineering. Kostas Kontogiannis is an Assistant Professor at Electrical and Computer Engineering Department, University of Waterloo. His research interests include software re-engineering, software migration, software reuse and knowledge based software engineering.

References

[1] Umar, Amjad, "Application (Re)Engineering: Building Web-Based Applications and Dealing with Legacies", Prentice Hall PTR, 1997.

[2] RamPrabhu, Robert Abarbanel, "Enterprise Computing: The Java Factor", Computer, P115, June 1997 IEEE.

[3] Walter Brenner, Rüdiger Zarnekow, and Harmut Wittig, "Intelligent Software Agents: Foundations and Applications", Springer-Verlag Berlin Heidelberg 1998.

[4] Alan R. Williamson, "Java Servlets By Example", Manning Publications Co., 1999.

[5] Victor Lesser, et al. "Resource-Bounded Searches in an Information Marketplace", IEEE Internet Computing, March/April 2000.

[6] Tuomas Sandholm and Qianbo Huai, "Nomad: Mobile Agent System for an Internet-Based Auction House", IEEE Internet Computing, March/April 2000.

[7] Ying Zou, Kostas Kontogiannis, "Localizing and Using Services in Web-Enabled Environments", 2nd International Workshop for Web Site Evolution, Switzerland, 2000.

[8] "Business-to-Business e-Commerce with Open Buying on the Internet", http://www.ibm.com/iac/papers/obi-paper/intro.html.

[9] "Gaining Competitvie Advantage in the Supply Chain: IBM Solution for Business Integration", http://www-4.ibm.com/software/info/ti/issues/scm.html

[10] Ronald Bourret, "XML and Databases", http://www.informatik.tu-darmstadt.de/DVS1/staff/bourret/xml/XMLAndDatabases.html

[11] Paul Dreyfus, "The Second Wave: Netscape on Usability in the Services-Based Internet", IEEE Internet Computing, March/April 1998.

[12] Joqquin Picon, et al, "Enterprise JavaBeans Development Using VisualAge for Java", http://www.redbooks.ibm.com.

[13] Prashant Patil, Ying Zou, Kostas Kontogiannis and John Mylopoulos, "Migration of Procedural Systems to Network-Centric Platforms", CASCON'99, Toronto, 1999.

[14] Kostas Kontogiannis, Prashant Patil, "Evidence Driven Object Identification in Procedural Code", STEP'99, Pittsburgh, Pennsylvania, 1999.

[15] Richard Gregory, Kostas Kontogiannis, "Requirements for a Distributed Tool Integration System", http://www.swen.uwaterloo.ca/~rwgregor.

[16] George Koulouris et.al “Distributed Systems: Concepts and Design”, Addison-Wesley, Second Edition, 1996.
[17] Michi Henning, Steve Vinoski, “Advanced CORBA Programming with C++”, Addison-Wesley, 1999.
[18] Clemens Szyperski, “Component Software: Beyond Object-Oriented Programming”, Addison-Wesley, 1998.
[19] Cynthia Della Torre Cicalease, Shmuel Rotenstreich, “Behavioral Specification of Distributed Software Component Interfaces”, Computer, July 1999 IEEE.
[20] “CORBA and XML; Conflict or Cooperation?”, http://www.omg.org, 1999.
[21] Mark Elenko, Mike Reinertsen, “XML & CORBA”, http://www.omg.org, 1999.
[22] Addreas Vogel, Madhavan Rangarao, “Programming with Enterprise JavaBeans, JTS and OTS: Building Distributed Transactions with Java and C++”, Jon Wiley & Sons, Inc, 1999.
[23] “Using VisualAge for Java Enterprise Version 2 to Develop CORBA and EJB Applications”, http://www.redbooks.ibm.com.
[24] “Programmer’s Guide: Visibroker for C++”, Inprise Corporation.
[25] W. Ku et.al. “End-to-end E-commerce Application Development Based on XML Tools”, IEEE Data Engineering, Vol. Vol. 23, No.1, pp. 29-36.