Unraveling the Web Services Web

An Introduction to SOAP, WSDL, and UDDI

Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana • *IBM T.J. Watson Research Center*

Over the past few years, businesses have interacted using ad hoc approaches that take advantage of the basic Internet infrastructure. Now, however, Web services are emerging to provide a systematic and extensible framework for application-to-application interaction, built on top of existing Web protocols and based on open XML standards.

Say, for example, that you want to purchase a vacation package using an online travel agent. To locate the best prices on airline tickets, hotels, and rental cars, the agency will have to poll multiple companies, each of which likely uses different, incompatible applications for pricing and reservations. Web services aim to simplify this process by defining a standardized mechanism to describe, locate, and communicate with online applications. Essentially, each application becomes an accessible Web service component that is described using open standards. An online travel service could thus use the same Web services framework to locate and reserve your package elements, as well as to lease Internet-based credit check and bank payment services on a pay-per-use basis to expedite fund transfers between you, the travel agency, and the vendors.

The Web services framework is divided into three areas — communication protocols, service descriptions, and service discovery — and specifications are being developed for each. In this article, we look at the specifications that are currently the most salient and stable in each area:

- the simple object access protocol (SOAP, www.w3.org/2000/xp) which enables communication among Web services;
- the Web Services Description Language (WSDL, www.w3.org/TR/wsdl.html), which provides a formal, computer-readable description of Web services; and
- the Universal Description, Discovery, and Integration (UDDI, www.uddi.org) directory, which is a registry of Web services descriptions.

At this point, Web services technology is still emerging, and researchers are still developing important pieces, including quality of service descriptions and interaction models. Because the Web services framework is modular, however, you can use just the parts of the stack you need. Therefore, developers can take advantage of the available specifications and tooling now and incorporate more modules as the technology matures.

Communication: SOAP

Given the Web's intrinsically distributed and heterogeneous nature, communication mechanisms must be platform-independent, international, secure, and as lightweight as possible. XML is now firmly established as the lingua franca for information and data encoding for platform independence and internationalization. Building a communication protocol using XML is thus a natural answer for Web services.

Enter SOAP, which was initially created by Microsoft and later developed in collaboration with Developmentor, IBM, Lotus, and UserLand. SOAP is an XML-based protocol for messaging and remote procedure calls (RPCs). Rather than define a new transport protocol, SOAP works on existing transports, such as HTTP, SMTP, and MQSeries.

At its core, a SOAP message has a very simple

 1089-7801/02/$17.00

structure: an XML element with two child elements, one of which contains the header and the other the body. The header contents and body elements are themselves arbitrary XML. Figure 1 shows a SOAP envelope's structure.

In addition to the basic message structure, the SOAP specification defines a model that dictates how recipients should process SOAP messages. The message model also includes *actors*, which indicate who should process the message. A message can identify actors that indicate a series of intermediaries that process the message parts meant for them and pass on the rest.

```
<SOAP:Envelope xmlns:SOAP=
      "http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP:Header>
      <!– content of header goes here –>
   </SOAP:Header>
   <SOAP:Body>
      <!– content of body goes here –>
   </SOAP:Body>
</SOAP:Envelope>
```

Figure 1. Structure of a SOAP message. The envelope features child elements that contain the message header and body elements.

Messaging Using SOAP

At the basic functionality level, you can use SOAP as a simple messaging protocol. Throughout this article, we'll illustrate the Web services specifications using a simple example drawn again from the travel services industry. Our traveler, Joe, is scheduled for an afternoon flight and wants to checkin electronically. We'll assume that Joe knows of a service with an electronic `CheckIn` method and that he knows the format for encoding the ticket. Given this, he could simply create and send a SOAP message to that service for processing.

Figure 2 shows such a SOAP message, carried by HTTP. The HTTP headers are above the `SOAP:Envelope` element. The `POST` header shows that the message uses `HTTP POST`, which browsers also use to submit forms. Following the `POST` header is an optional `SOAPAction` header that indicates the message's intended purpose. If there were a response, the HTTP response would be of type `text/xml`, as declared in the `Content-Type` header, and could contain a SOAP message with the response data. Alternatively, the recipient could deliver the response message later (asynchronously).

Note that the message in Figure 2 has no SOAP headers; the body simply contains an XML representation of an e-ticket with the person's name and flight details. Any realistic B2B scenario would, of course, have many headers indicating further information, including the sender's credentials and correlation information.

```
POST /travelservice
SOAPAction: "http://www.acme-travel.com/checkin"
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns:SOAP=
      "http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP:Body>
      <et:eTicket xmlns:et=
         "http://www.acme-travel.com/eticket/schema">
         <et:passengerName first="Joe" last="Smith"/>
         <et:flightInfo airlineName="AA"
                        flightNumber="1111"
                        departureDate="2002-01-01"
                        departureTime="1905"/>
      </et:eTicket>
   </SOAP:Body>
</SOAP:Envelope>
```

Figure 2. SOAP message containing an e-ticket. The `SOAPAction` *header indicates the message's purpose. In a real-world scenario, the message would contain additional information, including the sender's credentials.*

Remote Procedure Calls Using SOAP

To use SOAP for RPCs, you must define an RPC protocol, including:

- how typed values can be transported back and forth between the SOAP representation (XML) and the application's representation (such as a Java class for a ticket), and
- where the various RPC parts are carried (object identity, operation name, and parameters).

The W3C's XML schema specification (www.w3.org/XML/Schema) provides a standard language for defining the document structure and the XML structures' data types. That is, given a type like `integer` or a complex type, such as a record with two fields (say, an integer and a string), XML schema offers a standard way to write the type in XML. To enable transmission of the typed values, SOAP assumes a type system based on the one in XML schema and defines its canonical encoding in XML. Using this encoding style, you can produce an XML encoding for any type of structured data. RPC arguments and responses are also rep-

```
POST /travelservice
SOAPAction: "http://www.acme-travel.com/flightinfo"
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns:SOAP=
    "http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP:Body>
    <m:GetFlightInfo
      xmlns:m="http://www.acme-travel.com/flightinfo"
      SOAP:encodingStyle=
        "http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:xsi=
        "http://www.w3.org/2001/XMLSchema-instance">
      <airlineName xsi:type="xsd:string">UL
      </airlineName>
        <flightNumber xsi:type="xsd:int">506
        </flightNumber>
    </m:GetFlightInfo>
  </SOAP:Body>
</SOAP:Envelope>
```

Figure 3. SOAP RPC call. To find out if his flight is on time, Joe sends a string containing the airline's name and an integer with the flight number.

resented using this encoding.

Now, let's say Joe wants to know whether his flight has been delayed. He knows that the service has a function, `GetFlightInfo`, which takes two arguments — a string containing the airline name and an integer with the flight number — and returns a structured value (a record) with two fields — the gate number and flight status. In this case, Joe can get flight status by sending the service an `HTTP POST` carrying a SOAP envelope like the one in Figure 3.

In this SOAP envelope, the call to `GetFlightInfo` is an XML element with attributes that include information about the encoding (note the references to XML schema). The element's children are the method call's arguments: `airlineName` and `flightNumber`. Their types are defined in the type attributes, where `xsd` refers to the XML schema definitions. When the SOAP implementation receives the message, it converts the XML text for `UL` and `506` into the appropriate string and integer based on the service's implementation. It then calls the `GetFlightInfo` method with those arguments.

Figure 4 shows the response to this request. In this case, the response contains a structured value with the subvalues for gate number and flight status. Luckily, Joe's flight is on time.

SOAP implementations exist for several programming languages, including C, Java, and Perl, which automatically generate and process the SOAP messages. Assuming the messages conform to SOAP specifications, they can thus be exchanged by services implemented in different languages.

Description: WSDL

Speaking a universal language is not very useful unless you can maintain the basic conversations that let you achieve your goals. For Web services, SOAP offers basic communication, but it does not tell us what messages must be exchanged to successfully interact with a service. That role is filled by WSDL, an XML format developed by IBM and Microsoft to describe Web services as collections of communication end points that can exchange certain messages. In other words, a WSDL document describes a Web service's interface and provides users with a point of contact.

In this section, our examples are fragments of a WSDL document that describes a Web service that can process the two types of interactions in our SOAP examples. The first interaction, `GetFlightInfo`, is accessed using the SOAP RPC model; it takes an airline name and a flight number and returns a complex (or structured) type with flight information. The second, `CheckIn`, uses pure SOAP messaging; it expects to receive an XML representation of an electronic ticket, and returns no information.

A complete WSDL service description provides two pieces of information: an application-level service description, or abstract interface, and the specific protocol-dependent details that users must follow to access the service at concrete service end points. This separation accounts for the fact that similar application-level service functionality is often deployed at different end points with slightly different access protocol details. Separating the description of these two aspects helps WDSL represent common functionality between seemingly different end points.

Abstract Description

WSDL defines a service's abstract description in terms of messages exchanged in a service interaction. There are three main components of this abstract interface: the vocabulary, the message, and the interaction. Agreement on a vocabulary is the foundation of any type of communication. WSDL uses external type systems to provide data-type definitions for the information exchange. Although WSDL can support any type system, most services use XSD. Figure 5 shows two data types defined in XSD (`string` and `int`), and two data types defined in external schema (`Flight`

InfoType and Ticket). WSDL can import such external XSD definitions using an "import" element specifying their location.

WSDL defines message elements as aggregations of parts, each of which is described by XSD types or elements from a predefined vocabulary. Messages provide an abstract, typed data definition sent to and from the services. The example in Figure 5 shows the three messages that might appear during a Web services interaction. The message, GetFlightInfoInput, has two parts: airlineName, which is an XSD string, and flightNumber, which is an XSD integer. The other two messages, GetFlightInfoOutput and CheckInInput have only one part each. The operation and portType elements combine messages to define interactions. Each operation represents a message exchange pattern that the Web service supports, giving users access to a certain basic piece of service functionality. An operation is simply a combination of messages labeled as input, output, or fault to indicate what part a particular message plays in the interaction.

A portType is a collection of operations that are collectively supported by an end point. In our example, AirportServicePortType describes two operations: a single request-response operation, GetFlightInfo, which expects the GetFlightInfoInput message as input and returns a GetFlightInfoOutput message as the response; and a one-way operation, CheckIn, which just takes the CheckInInput message as input.

Concrete Binding Information

So far, all of the elements that we have discussed describe the service's application-level functionality. To complete the description of client–service interaction, we need three more pieces of information:

- *what* communication protocol to use (such as SOAP over HTTP),
- *how* to accomplish individual service interactions over this protocol, and
- *where* to terminate communication (the network address).

WSDL's binding element provides the "what" and "how" parts of this information, including the communication protocol and data format specification for a complete portType. In short, the binding element tells how a given interaction occurs over the specified protocol. Figure 6 (next page) shows a fragment from our example. The binding describes how to use SOAP to access the travelservice ser-

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns:SOAP=
    "http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP:Body>
    <m:GetFlightInfoResponse
        xmlns:m="http://www.acme-travel.com/flightinfo"
        SOAP:encodingStyle=
          "http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi=
          "http://www.w3.org/2001/XMLSchema-instance">
      <flightInfo>
        <gate xsi:type="xsd:int">10</gate>
        <status xsi:type="xsd:string">ON TIME</status>
      </flightInfo>
    </m:GetFlightInfoResponse>
  </SOAP:Body>
</SOAP:Envelope>
```

Figure 4. SOAP RPC response. The travel service responds to Joe's request with a structured value containing subvalues for gate number and flight status.

```
<message name="GetFlightInfoInput">
  <part name="airlineName" type="xsd:string"/>
  <part name="flightNumber" type="xsd:int"/>
</message>

<message name="GetFlightInfoOutput">
  <part name="flightInfo" type="fixsd:FlightInfoType"/>
</message>

<message name="CheckInInput">
  <part name="body" element="eticketxsd:Ticket"/>
</message>

<portType name="AirportServicePortType">
  <operation name="GetFlightInfo">
    <input message="tns:GetFlightInfoInput"/>
    <output message="tns:GetFlightInfoOutput"/>
  </operation>
  <operation name="CheckIn">
    <input message="tns:CheckInInput"/>
</operation>
</portType>
```

Figure 5. WSDL abstract description. This fragment shows the string *and* int *data types, which are defined in XSD, and two other data types defined in external schema:* FlightInfoType *and* Ticket*, which we assume were imported earlier in the WSDL file.*

```
<binding name="AirportServiceSoapBinding"
         type="tns:AirportServicePortType">
  <soap:binding transport=
         "http://schemas.xmlsoap.org/soap/http"/>
  <operation name="GetFlightInfo">
    <soap:operation style="rpc"
         soapAction="http://acme-travel/flightinfo"/>
    <input>
      <soap:body use="encoded"
         namespace="http://acme-travel.com/flightinfo"
         encodingStyle=
         "http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>
    <output>
      <soap:body use="encoded"
         namespace="http://acme-travel.com/flightinfo"
         encodingStyle=
         "http://schemas.xmlsoap.org/soap/encoding/"/>
    </output>
  </operation>
  <operation name="CheckIn">
    <soap:operation style="document"
         soapAction="http://acme-travel.com/checkin"/>
    <input>
      <soap:body use="literal"/>
    </input>
  </operation>
</binding>

<service name="travelservice">
  <port name="travelservicePort"
         binding="tns:AirportServiceSoapBinding">
    <soap:address location=
         "http://acmetravel.com/travelservice"/>
  </port>
</service>
```

Figure 6. WSDL's concrete binding information. As this fragment shows, `GetFlightInfo` *is a SOAP RPC interaction and* `CheckIn` *is a pure messaging interaction that uses XSD to describe the transmitted XML.*

vice. In particular, the WSDL document shows that

- `GetFlightInfo` will be a SOAP-RPC-style interaction, in which all message exchanges use standard SOAP encoding, and
- `CheckIn` is a pure messaging interaction ("document-oriented," in WSDL terms) in which the SOAP message's body contains the encoded message with no additional type encoding; that is, it uses XSD to literally describe the transmitted XML.

All that remains now is to define "where" to access this combination of abstract interface and protocol and data marshalling details (the binding). A `port` element describes a single end point as a combination of a binding and a network address. Consequently, a service element groups a set of related ports. In our travel service example, a single port describes an end point that processes SOAP requests for the `travelservice` service.

Using WSDL

For users and developers, WSDL provides a formalized description of client–service interaction. During development, developers use WSDL as the input to a proxy generator that produces client code according to the service requirements. WSDL can also be used as input to a dynamic invocation proxy, which can then generate the correct service requests at runtime. The result in both cases is to relieve the user and developer of the need to remember or understand all the details of service access. For example, travel service users need only obtain the WSDL description and use it as input to the tooling and runtime infrastructure to exchange the correct SOAP message types with the service.

Discovery: UDDI

The Universal Description, Discovery, and Integration specifications offer users a unified and systematic way to find service providers through a centralized registry of services that is roughly equivalent to an automated online "phone directory" of Web services. The browser-accessible UDDI Business Registry (UBR) became available shortly after the specification went public. Several individual companies and industry groups are also starting to use "private" UDDI directories to integrate and streamline access to their internal services.

UDDI provides two basic specifications that define a service registry's structure and operation:

- a definition of the information to provide about each service, and how to encode it; and
- a query and update API for the registry that describes how this information can be accessed and updated.

Registry access is accomplished using a standard SOAP API for both querying and updating. Here we focus on the first aspect, which provides a good idea of how the registry operates.

Organizing Structure

UDDI encodes three types of information about Web services:

- "white pages" information includes name and contact details;
- "yellow pages" information provides a categorization based on business and service types; and
- "green pages" information includes technical data about the services.

The UDDI registry is organized around two fundamental entities that describe businesses and the services they provide. The `businessEntity` element illustrated in Figure 7 provides standard white-pages information, including identifiers (tax IDs, social security numbers, and so on), contact information, and a simple description. Each business entity might include one or more `businessService` elements, shown in Figure 8 (next page), that represent the services it provides. Both business and service entities can specify a `categoryBag` to categorize the business or service (we discuss later how to encode this information).

Figures 7 and 8 (next page) show an important aspect of UDDI's design: Unique keys identify each data entity — businesses, services, and so on — that might be cross-referenced. These assigned keys are long hexadecimal strings generated when the entity (in this case, a business) is registered. The keys are guaranteed to be universally unique identifiers (UUIDs). For example, the `businessKey` attribute uniquely identifies a business entity and the `serviceKey` attribute identifies a service. A service also references its host by its business key. In addition to a human-readable description, name, and categorization, the service entity contains a list of `bindingTemplates` that encode the technical service-access information. Each binding template represents an access point to the service. The assumption is that the same service can be provided at different endpoints, each of which might have different technical characteristics.

```
<businessEntity businessKey=
       "A687FG00-56NM-EFT1-3456-098765432124">
  <name>Acme Travel Incorporated</name>
  <description xml:lang="en">
    Acme is a world leader in online travel services
  </description>
  <contacts>
    <contact useType="US general">
      <personName>Acme Inc.</personName>
      <phone>1 800 CALL ACME</phone>
      <email useType="">acme@acme-travel.com</email>
      <address>
        <addressLine>Acme</addressLine>
        <addressLine>12 Maple Avenue</addressLine>
        <addressLine>Springfield, CT 06785</addressLine>
      </address>
    </contact>
  </contacts>
  <businessServices> ...
  </businessServices>
  <identifierBag> ...
  </identifierBag>
  <categoryBag> ...
    <keyedReference tModelKey=
          "UUID:DB77450D-9FA8-45D4-A7BC-04411D14E384"
          keyName="Electronic check-in"
          keyValue="84121801"/>
  </categoryBag>
</businessEntity>
```

Figure 7. Simplified `businessEntity` *structure. This element offers standard white-pages information, including contact information and basic service descriptions.*

Technical Descriptions and tModels

A closer look at the binding template shows a great deal about how UDDI enables business and service descriptions using arbitrary external information (that is, information that is not defined by UDDI itself). Most of the information in a binding template is what we would naturally expect for an endpoint. Foremost is the end point address where the service can be accessed. This address might be a URL, e-mail address, or even a phone number. We also find the expected unique key (`bindingKey`) and a cross-reference to the service key.

The most interesting field, however, is `tModelInstanceDetails`, which provides the service's technical description (the green-pages information). The field contains a list of references to the technical specifications with which the service complies. The service provider first registers the required technical specifications in the directory, which assigns a corresponding unique identifier key. UDDI represents each registered technical specification using a new information entity, the `tModel`. Service endpoints that support the specification can then simply add the corresponding reference to their `tModelInstanceDetails` list.

As an example, let's say we want to register a WSDL document as a `tModel`. Assuming that the travel industry has defined the standard WSDL interfaces and bindings for electronic check-in and retrieval of flight information, we first create a `tModel` like the one in Figure 9 (page 93) to represent these WSDL definitions. Service end points that implement those interfaces can then include

```
<businessService serviceKey=
      "894B5100-3AAF-11D5-80DC-002035229C64"
    businessKey=
      "D2033110-3AAF-11D5-80DC-002035229C64">
  <name>ElectronicTravelService</name>
  <description xml:lang="en">Electronic Travel
Service</description>
  <bindingTemplates>
    <bindingTemplate bindingKey=
          "6D665B10-3AAF-11D5-80DC-002035229C64"
        serviceKey=
          "89470B40-3AAF-11D5-80DC-002035229C64">
      <description>
        SOAP-based e-checkin and flight info
      </description>
      <accesssPoint URLType="http">
        http://www.acme-travel.com/travelservice
      </accessPoint>
      <tModelInstanceDetails>
        <tModelInstanceInfo tModelKey=
          "D2033110-3BGF-1KJH-234C-09873909802">
        ...
        </tModelInstanceInfo>
      </tModelInstanceDetails>
    </bindingTemplate>
  </bindingTemplates>
  <categoryBag> ...
  </categoryBag>
</businessService>
```

Figure 8. Simplified `businessService` *structure. A business might include such a structure for each service it provides. The* `tModelInstanceDetails` *provides the service's technical description (the green-pages information).*

the corresponding `tModels` in their instance-details lists. To be useful, of course, users and implementers of compliant services must be aware of the registered `tModels` and their keys.

The idea behind the `tModel` mechanism is simple and powerful: To adequately describe a service, we often have to reference information whose type or format cannot (and should not) be anticipated. Replacing the information itself with a unique key provides a reference to arbitrary information types.

Categorization

Effectively locating particular types of businesses and services depends on our ability to qualify the directory's business and service entries according to a categorization scheme or taxonomy. In fact, in any realistic situation, we typically need multiple information bits, such as geographical location or type of industry or product, to characterize the service we are seeking.

To identify taxonomical systems, each classification system itself is registered as a `tModel` in the UDDI registry. Taxonomy information is then encoded in name-value pairs, qualified by a `tModel` key reference that identifies which taxonomy each pair belongs to. Three standard taxonomies are cited by UDDI and preregistered in the UBR:

- an industry classification complying with the North American Industry Classification System (NAICS) taxonomy,
- a classification of products and services complying with the Universal Standard Products and Services Code System (UNSPSC) taxonomy, and
- a geographical categorization system complying with the International Organization for Standardization Geographic taxonomy (ISO 3166).

Using categorization, we can query the UDDI directory to locate very specific types of services. In our case, we might search for travel services that provide electronic checkin and operate in Joe's metropolitan area. Once we find the service in UDDI, we can retrieve the WSDL description it complies with to learn how to interact with it using SOAP messaging and SOAP RPC calls.

What Next ?

In a real business situation, even our simple travel scenario would require more than three pieces of the Web services framework to operate properly. At the very least, we would have to ensure that transactions like the electronic check-in were conducted in a secure environment and that messages were reliably delivered to their destinations.

Why must we build additional security when we have technologies such as Secure Multipurpose Internet Mail Extensions (S-MIME), HTTP Secure (HTTPS), and Kerberos? The answer lies in the difference between end-to-end and single-hop usage. Business messages typically originate deep inside one enterprise and go deep inside another. Mechanisms such as Secure Sockets Layer are great for securing (for confidentiality) a direct connection from one machine to another, but they are of no help if the message has to travel over more than one connection. That's why we need security at the SOAP level.

Researchers are now defining a security model as a set of add-on specifications. For example, the SOAP Security Extensions: Digital Signatures proposal (www.w3.org/TR/SOAP-dsig/) describes how SOAP messages can be digitally signed. Groups are also developing specifications for authentication,

confidentiality, and authorization using SOAP.

A second important aspect for real business integrations is the communication protocol's reliability. Again, we could use a reliable transport, such as Reliable HTTP (HTTPR), but the multihop scenario demands that reliability be defined at the SOAP level. We expect researchers to add a "reliable SOAP" standard to the basic protocol soon.

Finally, complex business interactions require support for higher levels of business functionality. In particular, we need descriptions of quality-of-service properties that service end points offer. On the other hand, business interactions are typically long running and involve multiple interactions between partners. To deploy and effectively use these types of services, we must be able to represent business processes and stateful services, and be able to create service compositions (complex aggregations) in a standardized and systematic fashion. Several proposals for accomplishing this now exist; see, for example, Web Services Flow Language (www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf) and X-Lang (www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm).

Francisco Curbera is a research staff member in the Component Systems group at IBM's T.J. Watson Research Center. He received a PhD in computer science from Columbia University. He has worked for several years on the use of markup languages for application development and software-component composition. He is also coauthor of the WSDL and WSFL specifications.

Matthew Duftler is a software engineer in the Component Systems group at IBM T.J. Watson Research Center. He was one of the original authors of Apache SOAP, and is the co-lead of JSR110, Java APIs for WSDL.

Rania Khalaf is a software engineer in the Component Systems group at the IBM T.J. Watson Research Center. She received her BS and MEng in computer science and electrical engineering from MIT.

William Nagy is a software engineer at IBM's T.J. Watson Research Center. His research interests include the development of wide area distributed services and the infrastructure necessary to use and support them. His current focus is on Web services and includes the development of IBM's Web Services Gateway and the WS-Inspection specification.

Nirmal Mukhi is a research associate in the Component Systems group at the IBM T. J. Watson Research Center, where he does Web services research. He is codeveloper of the Web Services Invocation Framework.

Sanjiva Weerawarana is a research staff member in the Component Systems group at IBM T.J. Watson Research Center. He is coauthor of the WSDL and WSFL specifications, and codeveloper of Apache SOAP, WSTK, WSDL Toolkit, WSIF, and WSGW. He received a PhD in computer science from Purdue University.

Readers can contact the authors at {rkhalaf, sanjiva, curbera, duftler, nmukhi}@us.ibm.com.

Web Services Resources

- Web services news, articles, and software information • www.webservices.org
- IBM's developerWorks site (Web services tutorials, articles, forums, and tools) • www–106.ibm.com/developerworks/webservices
- Microsoft's Web services pages • www.gotdotnet.com/team/XMLwebservices
- W3C Web services Workshop • www.w3.org/2001/01/WSWS
- W3C SOAP specification • www.w3.org/2000/xp
- WSDL 1.1 specification • www.w3.org/TR/wsdl.html
- UDDI pages • www.uddi.org
- SOAP Digital Signatures Proposal • www.w3.org/TR/SOAP-dsig/
- Web Services Flow Language • www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf
- X-Lang • www.gotdotnet.com/team/xml_wsspecs/xlang-c/

```
<tModel tModelKey="">
  <name>http://www.travel.org/e-checkin-interface</name>
  <description xml:lang="en">
    Standard service interface definition for travel
  services
  </description>
  <overviewDoc>
    <description xml:lang="en">
      WSDL Service Interface Document
    </description>
  <overviewURL>
    http://www.travel.org/services/e-checkin.wsdl
  </overviewURL>
  </overviewDoc>
  <categoryBag> ...
  </categoryBag>
</tModel>
```

Figure 9. Sample `tModel` *definition. To register a WSDL document as a* `tModel`*, we assume standard industry definitions for electronic check-in and retrieval of flight information and create a* `tModel` *to represent these WSDL definitions.*