Automating Metadata Web Service Deployment for Problem Solving Environments

Ozgur Balsoy[1], Ying Jin[2], Galip Aydin[2], Marlon Pierce[2], and Geoffrey Fox[2]

[1]Computer Science Department, Florida State University, Tallahassee, FL 32306
Ozgur@csit.fsu.edu
[2]Community Grids Lab, Indiana University, 501 N. Morton Street, Suite 224, Bloomington, IN 47404-3730
{Yinjin, Gaydin, Marpierc, Gcf}@indiana.edu

Abstract. XML-based metadata information services are a crucial core service needed by Problem Solving Environments built over emerging service-based, globally-scaled distributed systems, as envisioned by the Open Grid Services Architecture and the Semantic Web. Developing user interfaces and services bindings for manipulating instances of particular schemas is thus extremely important and needs to be made as simple as possible. In this paper we describe procedures for automating the creation of Web Service environments that can be used to simplify the creation and deployment of schema-based metadata services.

1 Introduction

One of the most important underpinnings of Problem Solving Environments (PSEs) is the management of diverse types of information. Grid computing and distributed Web services are metadata rich: all entities should be described by XML metadata. Metadata must be used to describe the various parts that must be managed by the PSE, such as available applications, service interfaces and their binding points, batch queuing systems, hardware profiles, and so on. Many projects have attempted to define metadata pieces needed to build service grids, and such pieces will be needed as grid service data in the Open Grid Service Architecture [1,2]. In addition to the service metadata, there is also need to provide organizational metadata descriptors for the services themselves, and we have suggested data models for describing applications [3,4] and their associated service bindings. We thus anticipate that metadata richness will only increase.

We may place these issues into larger contexts. *Autonomic computing* (see [5] for a description and [6,7] for sample academic projects) seeks to define the interaction of components in next generation computing at all scales. In the autonomic computing vision, components mean everything from the physical component parts of individual computers to the service components of globally-scaled service environments. The ideal autonomic system is self-aware, self-monitoring, adaptive, and self-healing, among other characteristics.

Developing standards for Web services for computational grids is a major (and somewhat independent) component of the autonomic computing vision. Web services

P.M.A. Sloot et al. (Eds.): ICCS 2003, LNCS 2660, pp. 393–402, 2003.

are somewhat loosely defined, but may be characterized in general as using existing Web technologies and standards (such as HTTP and XML) to build the distributed computing environments. Important Web service standards include the Web Service Description Language (WSDL) [8] and the Simple Object Access Protocol (SOAP) [9]. The Open Grid Service Architecture, developed under the auspices of the Global Grid Forum and primarily led by IBM and the Globus team, seeks to extend the capabilities of Web services in order to support more sophisticated services. The key to this development is the extension of WSDL portTypes to include metadata and service state information.

In a major alternative effort, other groups are using various XML standards of the World Wide Web Consortium define what it terms the Semantic Web [10]. The Semantic Web vision borrows ideas from federated information systems and artificial intelligent, modified and simplified to apply to the much more ambitious problem of intelligent information retrieval and service execution on a global scale. The cornerstones of the Semantic Web are metadata descriptions (RDF) [11] and ontologies that provide semantic meaning.

XML-based information services are the key to both the OGSA and the Semantic Grid. We may go a step further and claim that the there is a general need for many informal, lightweight information management systems for particular schemas. We have developed such a system, described in Refs [12] and [13], which provides several per-schema implementations: newsgroups, citation and reference managers, training and conference registration systems. All such systems possess the same general capabilities: browsing, search, "push" and "pull" style event delivery options, and access controls on information and publishing. More sophisticated systems that support schema-spanning searches (with the associated meta-ontology research problems [14,15]) may be built on top of this practical base.

Given the demonstrable importance and proliferation of XML metadata, it becomes necessary to automate the creation of publication and editing tools for particular schemas. We describe the details of two such systems, which we refer to as "Schema Wizards". The first is designed to simplify the creation of XML wizards for specific schemas. These wizards are in turn used for creating and publishing instances in the XML messaging system described in [13]. The Schema Wizard is thus a wizard for wizards. The second wizard system, a follow-on to the first, is designed to automate metadata-based Web services (such as service data associated with Grid Web services). In both cases, a particular schema is used as the underlying data model, and the wizard generates the user interface (view) and action bindings (control) necessary to create and deploy the new metadata-based service.

2 Schema Wizard: Automating Metadata Creation for Publication

Applications that guide users through a complicated task are usually called wizards. XML wizards are applications that receive user inputs to generate XML documents. In our research on XML information frameworks (see particularly Ref [13]), XML is used as the medium for storing information, but the systems are user driven. Wizards help users generate schema-based XML content and publish into an XML messaging broker. Such documents are then typically published to various sources, including both persistent repositories and listeners demanding immediate notification.

In our current implementation, our information management system provides general purpose tools that can be used to create specific applications that are tied to single, specific schema. Thus to add a new application, one of the necessary tasks is to create a new publishing wizard to the particular schema. Creating such publishing wizards requires the following steps:

1. Writing an XML Schema which has all the necessary elements regarding with the current project.
2. Producing the Object Models of the XML Schema elements and attributes. This is done by using Castor's [16] Source Generator, which creates Java beans to correspond to the elements of the schema.
3. Obtaining user data. This step requires us to prepare HTML forms which have form elements to represent the schema elements. We use JavaServer Pages (JSP) technology, which allows us to interact with HTML forms and data objects at the same time. In that way we immediately assign the user entries to the data objects.
4. Generating the XML document. Using Castor's marshalling framework we create an XML document which contains the user inputs.
5. Publishing this XML document. This may use publish/subscribe systems such as the JavaMessaging System (JMS).

Given the repetitive, tedious nature of this process, we have developed a general purpose tool that automates the creation of this process. We refer to this tool as the SchemaWizard.

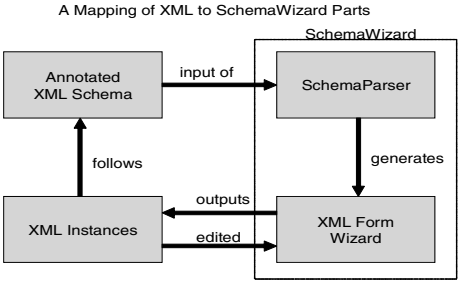

Fig. 1 The Schema Wizard is used to generate interfaces for particular XML schemas, which in turn generate XML instances for messaging.

2.1 Schema Wizard Architecture

The Schema Wizard maps XML Schema elements to HTML form elements through its schema parser, and creates the framework and logic for an XML form wizard, as depicted in Fig. 1. Users use newly generated wizards to create and publish XML instances, which follow a schema, to any destinations such as publish/subscribe messaging systems or through SMTP. XML form wizards are Web applications that also serve as validating XML editors and are customized through schema annotations.

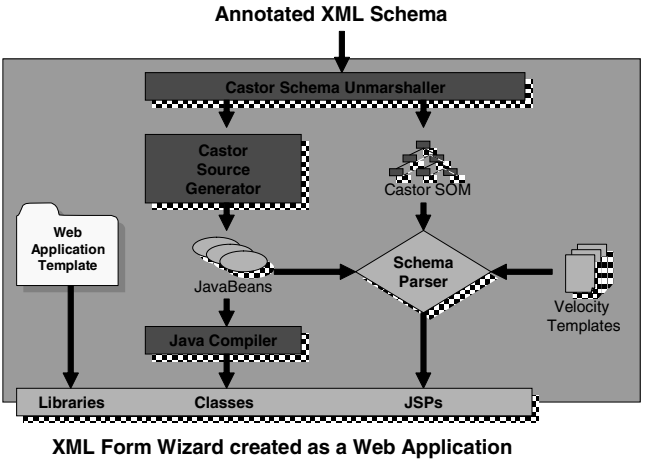

Fig. 2 The XML Schema Wizard simultaneously generates user interfaces, in-memory data bindings, and actions links between interface elements and data objects for an XML schema.

The Schema Wizard has been developed as a Web application compliant to JSP/Servlet specifications that generates and deploys new Web applications to help instantiate XML documents. A SW user defines her data model using a XML schema and annotates the schema to customize the HTML form view. [Fig. 2] After provided with a schema, the Wizard first creates the necessary directory structure and required libraries for a XML form wizard by unpacking a Web application template package. Second, the Wizard invokes Castor SourceGenerator [16] to create Java object definitions (a.k.a. javabeans) corresponding to individual XML schema elements, and compiles them into the previously created Web application directories. As the third step, the Wizard uses Castor's Schema Object Model (SOM) API to parse and process the schema in the memory. The SOM API is used for its convenient and specialized schema type access methods, which we prefer for this purpose over other XML processing APIs, i.e. DOM or JDOM.

The SchemaParser module, the heart of the Wizard application, traverses the schema using the SOM API and collects XML type and structure information. This information later is used to decide what type of template is necessary to generate final JSP pieces or *nuggets*. Velocity [21] templates are used for their simplicity to define JSP nuggets, each of which maps either a simple schema type or a complex type to one or more HTML elements rendered by JSP engine. The JSP page created for the root element is included by a higher level index page where the entire form view is finally constructed. The rules for mapping from XML schema types to HTML elements are listed in the following section.

2.2 Schema Element to HTML Mapping Rules

To generate XML instances based on XML schema elements, each element is mapped to one or more corresponding HTML form elements. The guidelines are as follows:

1. One HTML form page is generated for each possible root (or global) element, which is based on an anonymous or named global complex type.
2. XML schemas define elements for any level within XML document hierarchy. Some elements, for example, may not need any parent element to exist and can root a XML document. Such elements are called global elements and defined at the top level in a XML schema. Elements can also include sub elements and attributes. These elements are based on complex type definitions. SW generates forms for only global elements that are based on either named or anonymous complex types.
3. Each form has one form level submission, which submits all the content inside form elements and alters current values on the server side at once. Each form may also have more than one element level submission, which affects only related elements or attributes.
4. Simple, non-enumerating, non-repeating elements or attributes are mapped to a single input text field.
5. Simple typed elements or attributes must be based on predefined XML schema types such as numbers, character strings, and date and duration types. Any simple type can be restricted into a narrower type range. For an example, a new type can be generated from the integer type by restricting the values to the integer numbers between 0 and 100. Another example could be a character string type that must follow a pattern such as social security numbers with 9 digits and two dashes. Such schema types are mapped by SW to single input text fields and their values are only updated if a form level submission occurs. Data validation is performed on the server side while forms are used for some input size restrictions.
6. An enumerated simple type is mapped to a selection box with the values filled in. XML schemas allow some of the simple element or attribute values to be enumerated. Enumerated values are provided with type definitions inside schemas. Drop-down selection boxes with enumerated values filled in are used to map such types. If an enumerated element or attribute is optional, users are provided with an option to remove it. These types are updated by only form level submissions.
7. An unbounded simple type is mapped to an input field, a selection box, and two buttons to add and remove new values in addition to form level submissions. Unbounded simple types are elements that can repeat with the same or different values. The content of the input field is submitted and a new element is created when the submission is performed through the addition button. The new element is also added to the selection box. Users can remove any element by choosing its value from the selection box and submitting the form through the remove button.
8. A defaulted simple type is mapped to an input field with a default value.
9. A fixed value simple type is mapped to no form element, but the fixed value is written in plain text.
10. A simple Boolean-typed element or attribute is mapped to a pair of radio buttons with labels, “Yes” and “No.”

An unbounded complex value is mapped to, along with its sub elements, a selection box where the element's index can be chosen, and a pair of add and remove buttons. If any complex element index is chosen from the list, it can be removed by submitting through the remove button.

2.3 View Construction

Velocity templates are used to generate JSP pages. Examples of Velocity templates and JSP pages generated automatically may be found in [17]. Here, we will briefly go over an example and the generation of a JSP for an enumerated element.

All the Velocity variables starting with $ and surrounded by braces are replaced with the schema types information gathered by the SchemaParser. Elements and attributes are treated the same way in terms of the interface however, Castor SourceGenerator generates different source code for them, and the template reflects this through a Velocity variable, $attribute. Also, enumerated values are embedded into the JSP nugget through an array of values, $values, obtained through the SOM API and forwarded to the Velocity template engine by SchemaParser.

In the example, the parent element of the type is a complex type called rootElement. The JSP starts with retrieving the user input data, and checks whether this element has a parent or not. Based on the result of this check, the JSP nugget decides whether the memory object needs to be updated or the form is to be filled with the old data in memory. Finally, a selection box is printed out in HTML with enumerated values stored in an array of strings, values[].

All variables defined in generated JSP nuggets are local. Since parent elements include sub elements' JSPs, initializations are performed within parents' scope. Finally, the last part of the JSP decides whether this element is the root element and the submission buttons need to appear.

In XML schema, all the elements defined as global elements can be referenced at any level within the schema hierarchy. To support referenced global elements, we have designed the JSPs for parent elements so that they can be included by other complex typed elements.

The entry pages for HTML forms are index pages, which must not be included by other JSPs. SW does not provide such inclusions. Developers who use generated JSPs should also notice this requirement.

3 WSDL Wizards for Metadata Web Services

In the proceeding section we described an XML Schema Wizard that can be used to automate the process for creating specific wizards for generating XML messages. This system is appropriate for publish/subscribe style interactions. Here we describe our design for a variation of this system that is used to automatically generate remote procedure call (RPC)-style services. Instead of supporting all schema types, we need only to support one: WSDL. User interfaces generated from WSDL instances will be used to create server-side SOAP calls to back-end XML repositories.

As we argued in the introduction, an important subclass of Web services must be geared toward the creation and management of XML metadata. We refer to this particular subclass of Web services as Metadata Web services (MWS). Creating and deploying a new MWS has the following steps: *first*, create Java class bindings for the schema; s*econd*, create a Façade class to serve as Web service implementation and simplify manipulation of the data classes; *third,* deploy the service; *fourth*, generate client stubs from the service's WSDL, and *finally*, create the user interface to the service using the client stubs.

The Façade class for the server (step 2) may be used to simply manipulate the schema data objects, but for complicated schemas it also serves to simplify (at the expense of lost functionality) the task of step 5. These steps are somewhat tedious to perform manually. The tedious nature of creating these services has some unpleasant side effects: it inhibits the modification phase of trial schemas and (more globally) discourages the proliferation of lightweight, schema-based services.

The Schema Wizard itself, as described above, is designed for creating XML messages. In particular, it is not directly suitable for creating services and user interfaces from the WSDL schema. The WSDL document consists of two parts: service interface definitions and implementation bindings. The interface part contains the abstract definition of a service, which allows one to retrieve information of operations, and the types of the messages the operations receive and return. The implementation section contains concrete information on port and binding protocol. We thus do not want to create the user interface for a WSDL service description from the entire document but instead only the abstract portion (the portTypes) of the document.

The WSDL Wizard is designed to automate all parts needed to create a Metadata Web Service. This essentially has the following parts: a) selection of desired WSDL instance, b) generation of user interface and server-side bindings, and c) dynamic invocation of WSDL interface methods to manipulate the remote XML instance.

We are developing a Java package that allows SOAP procedures to be dynamically constructed for a given WSDL instance without needing to create client stub programs. This is described elsewhere [20], and in this paper we focus its combination with the Schema Wizard.

The major components need to accomplish this are illustrated in Figure 3. The user is assumed to have selected a particular service by inspecting a WSIL [19] document through a WSIL proxy client. The WSDL description of this interface is downloaded and the service generation procedure begins.

The important additional components are the WSDL Parser, GhostWSDL Parser, and User Data Parser. These parsers each are designed to provide access for manipulating in-memory representations of XML. A Controller component is responsible for managing all the components. The WSDL Parser is given the WSDL document that describes the desired service and creates an in-memory representation of the service interface. This document is then inspected by the Controller in order to dynamically generate a SOAP call.

As explained above, we do not need the entire WSDL description of a service to generate its interface. We thus construct a new “ghost” schema for the particular WSDL instance, based on the port type. This GhostWSDL instance contains only the information necessary to generate the user interface and the SOAP call. The generated GhostWSDL is passed to the Schema Wizard, which generates a user interface.

After the Schema Wizard user interface has been generated and the user has entered information, this must be passed to the Controller, which will need to use this data in the dynamically generated SOAP call. This data is accessed through the User Data Parser, which accepts the XML output of Schema Wizard. This output is simply an instance of the generated GhostWSDL schema, with the users’ data. For the simple types, the User Data parser parses the data in the XML to its corresponding Java value. For the complex type, the parser uses data to instantiate the Java object class which is generated by Schema Wizard. The results are placed to a Java object array.

Figure 3 shows how these components are related to each other. The arrow lines indicate the communication between the two components. The diagram uses the thicker lines for Internet request/response messages with the indicated protocols, and the thinner lines represent internal communications. The following is detailed explanation for each numbered step.

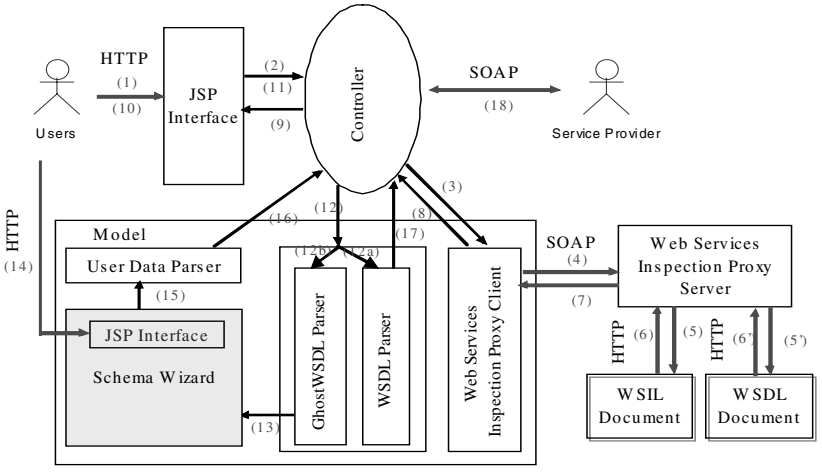

Fig. 3 A usage scenario illustrates the interaction of WSDI and Schema Wizard components.

1. The user requests the service list through a browser interface.
2. The JSP interface sends the user's request to Controller.
3. Controller communicates with the inspection proxy client (a WSIL client).
4. Proxy client makes the SOAP call to the web services inspection proxy server and requests the WSIL service list.
5. Proxy server makes the HTTP connection to a remote/local server for getting WSIL document
6. The remote/local server sends the WSIL document back to the proxy server
7. Proxy server sends back the service list to the client.
8. The proxy client responds to Controller.
9. Controller returns the service list to the user.
10. The user picks a desired Web service from the list WSIL list.
11. Steps (2) - (8) are repeated in order to retrieve the WSDL document, this time going through (5′) and (6′).
12. The WSDL document is sent to the parsers. In 12a, the WSDLParser creates in memory representation of WSDL. In 12b, the GhostWSDL is extracted from WSDL instance.
13. The GhostWSDL is sent to the Schema Wizard, which generates the user interface.
14. The user enters the required information of service into the system.
15. The wizard generates a GhostWSDL instance with the user's data, is created. If the GhostWSDL interface contains complex types, JavaBean classes are generated.

16. A Java Object array with the user data is passed to Controller.
17. The controller obtains an in-memory representation of the WSDL instance.
18. The Controller inspects the WSDL document, constructs, and launches the SOAP call to invoke the service.

4 Summary and Future Work

As part of our attempts to address a larger metadata management problem, this paper has presented two mechanisms for automatically deploying web-based applications for creating and manipulating metadata based on XML schemas. The Schema Wizard is a general purpose system for creating user interfaces and action bindings for a particular XML schema. As we have illustrated, the SW is used to generate browser interfaces automatically from a particular schema data model. It also binds the schema to in-memory data objects that can be used to manipulate schema instances. In addition to generating the view of the data model, we also generate the action bindings (in this case, HTML Form actions) that connect the user-supplied values to the data model. This is a useful component in message-based systems such as we describe in [13]. Future work on the Schema Wizard will be to introduce more customization of the presentation.

In contrast to one-to-many, message-style metadata services, the one-to-one, RPC-style metadata services are more meaningfully described with WSDL interfaces than message formats. To automatically generate the user interface to these services, we need to modify the Schema Wizard operation to display not the entire WSDL schema but the only the portion appropriate for display. In addition, the bindings must now change so that we can tie user interface actions to backend (rather than server-side) data instances, which we do with SOAP invocations dynamically generated from the WSDL instances. We have completed development of the preliminary libraries for dynamic WSDL invocations (supporting XML primitive types), and the WSDL wizard implementation based on these libraries is currently in progress. The next step will be to add support for WSDL complex types, as we describe in the paper.

References

1. Foster, I. Kesselman, C., Nick, J., Tuecke, S.: The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Available from www.globus.org/research/papers/ogsa.pdf. See also Foster, I., Kesselman, C., Nick, J., Tuecke, S.: Grid Services for Distributed Systems Integration. Computer, 35(6), (2002).
2. Tuecke, S., Czajkowski, K., Foster, I., Frey, J., Graham, S., Kesselman, C.: Grid Service Specification, Draft 4. (2002).
3. Pierce, M., et al: Interoperable Web Services for Computational Web Portals. To be published in Proceedings of Supercomputing 2002 (2002).
4. Youn, C., Pierce, M., Fox, G.: Building Problem Solving Environments with Web Services. Submitted for publication in proceedings of Workshop on Complex Problem Solving Environments for Grid Computing, International Conference on Computational Science (2003).

5. Autonomic Computing: IBM's Perspecitive on the State of Information Technology. Available from www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf.
6. Rhea, S., Wells, C., Eaton, P., Geels, D., Zhao, B., Weatherspoon, H., Kubiatowicz, J. : Maintenance-Free Global Data Storage. IEEE Internet Computing , Vol 5, No 5, pp 40–49 (2001).
7. Abramson, D., Buyya R., Giddy, J.: A Computational Economy for Grid Computing and Its Implementation in the Nimrod-G Resource Broker. To appear in Concurrency and Computation: Practice and Experience, Special Issue on Grid Computing.
8. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web Service Description Language (WSDL) version 1.1 Available from www.w3c.org/TR/wsdl.
9. Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H. F., Thatte, S., Winer, D.: Simple Object Access Protocol (SOAP) 1.1. W3C Note 08 May 2000. Available from www.w3.org/TR/SOAP/.
10. T. Berners-Lee, J. Hendler, and O. Lassila.: The Semantic Web. Scientific American (2001).
11. Lassila, O., Swick, R. R.: Resource Description Framework (RDF) Model and Syntax Specification. W3C Recommendation. 22 February 1999.
12. Balsoy, O., et al: The Online Knowledge Center: Building a Component Based Portal. Proceedings of the International Conference on Information and Knowledge Engineering (2002).
13. Aydin, G., Kaplan, A.,Topcu, A. E.,Yildiz, B., Balsoy, O., Pierce, M., Fox, G.:. An XML Based System for Dynamic Message Content Creation, Delivery, and Control. Proceedings of IASTED International Conference on Information and Knowledge Sharing (IKS 2002).
14. Kahng, J., McLeod, D. :Dynamic Classification Ontologies. Computing the Brain: A Guide to Neuroinformatics, Academic Press, 241–254 (2001).
15. Thacker, S., Sheth, A.,Patel, S.: Complex Relationships for the Semantic Web S.. In *Creating the Semantic Web* (D. Fensel, J. Hendler, H. Liebermann, and W. Wahlster, eds). MIT Press, 2001.
16. Birbeck, M. et al.: Professional XML, 2nd Ed.. Wrox Press, Inc. (671-721) 2001.
17. SchemaWizard Project Documentation. Available from ptlportal.communitygrids.iu.edu/schemawizard/index.html
18. Apache Axis User's Guide. Available from cvs.apache.org/viewcvs.cgi/~checkout~/xml-axis/java/docs/user-guide.html
19. Web Services Inspection Language, www-106.ibm.com/developerworks/webservices/library/ws-wsilspec.html
20. Jin, Y.: Web Service Dynamic Invocation. Community Grids Lab Internal Report. Available from grids.ucs.indiana.edu:8035/doc/Dynamics_Web_Services_Accessing.doc.
21. Apache Jakarta-Velocity Project. Available from jakarta.apache.org/velocity/index.html