Attribute-Based Programming for Grid Services

Glenn Wasson and Marty Humphrey
Computer Science Department
University of Virginia
Charlottesville, VA 22904

One of the key challenges for a "Grid programmer" today is how to expose functionality as an OGSI-compliant Grid Service. One of the principle differences between programming a Grid Service and simply coding the service logic is that a certain amount of meta-data is needed to actually expose that service logic as a grid service. This meta-data consists of instructions to the service's hosting environment about what elements (methods and data) should be exposed to clients and the conditions under which clients can access them (policy). This goes beyond simple automatic generation of service WSDL and client-side proxies to describing events that must be analyzed at runtime. In this paper, we describe the attribute-based programming model that we are developing as part of OGSI.NET, an OGSI-compliant hosting environment for the Microsoft .NET platform. Programmers annotate their code with attributes that are subsequently used by both pre-processing tools (e.g., to generate static information such as WSDL documents) and by the service's hosting environment (e.g,. to detect policy events). We focus on Service Data Elements (SDEs) and Service Security Policies. We argue that an attribute-based model can ultimately make creating a Grid Service as easy as creating a Web Service.

1. Introduction

Many of the standard behaviors and operations of Grid Services as defined in the OGSI specification [7] can be thought of as meta-information about the Grid Service's service logic. For example, a Grid Service must implement the Grid Service port type and therefore the `findServiceData` function. A Grid Service author must not only provide the logic for `findServiceData` (though it may come from a standard library), but also make that function available to clients using one of a set of Grid Service References (GSRs), communicating over any of a set of transport and messaging protocols. Both the decision to expose `findServiceData` and the fact that the Grid Service supports clients calling it using SOAP over HTTP are independent of the logic used to implement `findServiceData`. These decisions are meta-data necessary for the service author's logic to be utilized by other grid clients and services. However, this information is not typically as important to the service author as the logic itself. That is, Grid Service authors want to be able to say "here's my code… now make it available to the grid". Similar desires exist for exposing elements of a service's internal state ("make the following table available to clients who wish to inspect it").

In this paper, we describe how we exploit the support for attributes inherent in .NET to expose this meta-information in our OGSI-compliant hosting environment, OGSI.NET. We focus on Service Data Elements (SDEs) and Service Security Policies. In .NET, attributes are a simple declarative mechanism for adding meta-data to source code. These annotations can provide useful information both to static analysis tools and (more interestingly) to the runtime system through reflection on the entities defined in the code. Combined with the powerful development environment of Visual Studio .NET, we believe that this allows the Grid Service author to concentrate on the Grid Service logic and "forget about the grid plumbing," thereby making *Grid Service authoring on par with Web Service authoring!* Overall, the purpose of this paper is both [a] to show and convince Grid Service developers the value of an attribute-based Grid Service programming model, and [b] to argue for the use of an attribute model in other emerging OGSI-compliant hosting environments. The remainder of the paper is organized as follows. Section 2 introduces attributes used to expose service logic and discusses how they are used by the OGSI.NET hosting container and associated tools. Section 3 presents attributes used to expose service data (SDEs) and section 4 discusses on-going work in defining and using attributes to specify service policy. Section 5 contains the conclusions.

2. Attributes in OGSI.NET

The use of attributes in OGSI.NET attempts to strike a balance between powerful/expressive descriptors and simple annotations that make common cases easy to program. In keeping with this philosophy, a service's methods can be exposed to the grid in two ways. The first is with the OGSIPortType attribute, which can be placed on a class, as shown in Figure 1.

```
[OGSIPortType(typeof(FactoryPortType))]
[OGSIPortType(typeof(NotificationSourcePortType))]
[OGSIPortType(typeof(GridServicePortType))]
public class FactoryService : GridServiceSkeleton
{
```

Figure 1. Exposing Service Logic via Class Attributes

The OGSIPortType attribute allows Grid Service authors to specify that their service implements standard port types defined in libraries outside of the current service. Figure 1 shows a class, FactoryService, which implements several port types defined in the OGSI specification and which come with OGSI.NET. A Grid Service can be written simply by using the attribute [OGSIPortType(typeof(GridServicePortType))]. Service authors can also define their own port types that can similarly be included in other services via this same attribute. However, sometimes it is too cumbersome to define a port type and then a separate service which implements that port type (often leaving most of the logic in the port type implementation and very little in the service). For this case, OGSI.NET provides attributes which can be set on methods themselves to expose them as members of a given port type.

```
[WebMethod]
[SoapDocumentMethod("http://gcg.virginia.edu/samples/counter#subtract",
Binding="CounterSOAPBinding")]
[return: XmlElement("returnValue")]
public int subtract(int value)
{
```

Figure 2. Attributes to Define a Method of a Port Type in the Service Code

When the OGSI.NET container loads a particular service instance, an application domain [3] is created for the service instance and a Grid Service Wrapper (GSW) is instantiated within that domain. The GSW creates a table that maps binding-specific names to sets of service logic that clients can invoke. When client requests arrive at the container, they are dispatched to the service (and hence GSW) specified in the request and the GSW looks up the method to invoke from its table. For requests using SOAP/HTTP, the index into this table is the request's SOAP Action header. In Figure 2, the subtract function is mapped to the http://gcg.virginia.edu/samples/counter#subtract SOAP Action through the SoapDocumentMethod attribute. The return attribute is used to specify that the integer value returned is to be wrapped in an XML element named returnValue (e.g. <returnValue> 7 </returnValue>). Note that similar definitions exist within the implementations of the port types imported with the OGSIPortType attribute and the OGSI.NET container adds those mappings to the table as well.

The attributes in Figure 2 are actually defined and used by the .NET framework itself in the context of ASP.NET web services. Here, while they express similar content, they are used internally by the OGSI.NET container and not the ASP.NET pipeline.

A final attribute used to expose service methods is the OGSIOperationOverride attribute. This attribute allows a programmer to import a port type using the OGSIPortType attribute, but override the implementation of a specific function of that port type with their own. If the container detects this attribute on a function, it will call that function for any request who's SOAPAction matches the one given as the OGSIOperationOverride attribute's parameter, ignoring any previous functions mapped to that SOAPAction.

The OGSIPortType and SoapDocumentMethod attributes are used by OGSI.NET's WSDLGenerator to automatically generate WSDL for the service. The service's GSW will parse the WSDL at service load time to determine the SOAPAction mappings. However, the GSW must also check for

OGSIOperationOverride attributes (using reflection on the service class) at runtime because they represent a change in a method's implementation, though not its WSDL.

3. Exposing Service Data

A fundamental concept in OGSI is that of service data - publicly exposed service state that can be queried and set through well-known interfaces. The OGSI specification [7] defines extensions to the WSDL port type element to express data items that any service implementing that port type must expose. The number and type of the values associated with each data item as well as initial values may be specified in the WSDL. OGSI.NET provides a single parameterized attribute, SDE, which can be used on both classes and data members. This attribute is used by both the WsdlGenerator to create port type SDE declarations and by the runtime container to handle certain dynamic behaviors used for getting and setting SDE values.

```
[SDE("interface", typeof(XmlQualifiedName), false,
       MutabilityType.constant, false, "1", "unbounded")]
[SDE("serviceDataName", typeof(XmlQualifiedName), false,
       MutabilityType.mutable, false, "0", "unbounded")]
[SDE("factoryLocator", typeof(LocatorType), true,
       MutabilityType.mutable, false, "1", "1")]
[SDE("gridServiceHandle", typeof(HandleType), false,
       MutabilityType.extendable, false, "0", "unbounded")]
[SDE("gridServiceReference", typeof(ReferenceType), false,
       MutabilityType.mutable, false, "1", "unbounded")]
[SDE("findServiceDataExtensibility",
       typeof(FindServiceDataExtensibilityType), false,
       MutabilityType.mutable, false, "1", "unbounded")]
[SDE("setServiceDataExtensibility",
       typeof(SetServiceDataExtensibilityType), false,
       MutabilityType.@static, false, "2", "unbounded")]
[SDE("terminationTime", typeof(TerminationTimeType), false,
       MutabilityType.mutable, false, "1", "1")]
public class GridServicePortType : GridServiceSkeleton
{
```

Figure 3. SDE Declarations on the Grid Service Port Type

Figure 3 shows an example of using SDE attributes on a class definition, in this case the definition for the Grid Service Port Type. Each SDE attribute contains parameters for the name of the SDE, the type of the SDE's values, whether or not the SDE is nullable, the mutability type [7], whether or not it is modifiable, and the minimum and maximum number of values permitted. .NET types are used for the "type" parameter rather than XSD types because the .NET types used are serializable [4] and so the XSD type can be inferred.

OGSI.NET also supports *Get* and *Set* event handlers for service data. These handlers are code that is called whenever a service data element's values are retrieved or altered (add/delete/insert). Each SDE can have 0 or more Get and 0 or more Set handlers, which will be invoked in the order that they are added to the SDEs handler lists. The Get handler is useful when a service wishes to expose, through the service data interface, some information that must be determined dynamically, e.g. the current time. Whenever a call to a service's findServiceData method is received for an SDE with a Get handler, that handler is run and the value returned by the handler is returned to the caller. A Get handler can also be used to implement the "service data name" SDE which lists the names of all other SDEs exported by a service. Any SDEs added by the service at runtime are automatically reflected in the "service data name" SDE because the values of that SDE are determined at the time for the findServiceData call.

If an SDE has set handlers, they are invoked whenever a setServiceData call is received for that SDE (or whenever OGSI.NET's internal interface to service data is used to alter that SDE's values). Set handlers can be used to implement a number of important behaviors for grid computing, such as sending notification to subscribers when an SDE's values change, or saving an SDE's values to permanent storage.

```
[SDEGet("currentTime")]
public ArrayList getTime(OGSIServiceData sde, GridServiceWrapper gsw) { … }

[SDESet("sde1")]
public void saveSDE(OGSIServiceData sde, Object newVal, bool adding,
                    GridServiceWrapper gsw) { … }
```

Figure 4. SDE Get and Set Event Handlers

Attributes can be used to specify Get and Set event handlers for SDEs defined on the port type class. The Grid Service author can write custom functions or pick from a set of predefined functions included in OGSI.NET[1]. The `SDEGet` and `SDESet` attributes can be placed on methods (with appropriate function signatures), as shown in Figure 4.

The Get handler receives as parameters the actual SDE being retrieved and the wrapper of the service itself. It returns a list of values that are then returned to the client as the SDE's values. The Set handler's parameters are the SDE being altered, the actual value being added or removed from the SDE's values, a Boolean indicating whether an "add" or "remove" is occurring and a reference to the service wrapper.

A final use of the `SDE` attribute is as a short hand to simplify exposing a service's public data members. In Figure 5, a public data member, foo, has an SDE attribute. This attribute allows clients to access `SomeServicePortTypes`'s data member foo through the SDE interface. The attribute is used by the WsdlGenerator to define an SDE called "foo" in the service data section of the `SomeServicePortType` definition.

The SDE "foo" will be defined (in the service's WSDL) as modifiable, mutable, non-nillable and having a min and max occurs of 1. A Get handler will also be setup to return the current value of the variable foo when `findServiceData` is called on it. Based on the type of the data member receiving the `SDE` attribute, the WsdlGenerator will create different SDE declarations. For example, if the data member is an ArrayList, the minOccurs will be set to 0 and maxOccurs will be set to "unbounded". If the data member is of type string, it will be nillable.

The `SDE` attribute represents an attempt to balance powerful expressibility with ease of use. The full range of service data definitions can be expressed using the class-level `SDE` attribute. In its simpler form, on public data members, the `SDE` attribute provides an easy means of exposing dynamically changeable state to the grid. It is believed that the assumptions made by the WsdlGenerator and container for these member-level attributes represent common cases. This allows Grid Service authors to only manipulate programming-language types in their code, without worrying about transferring the values of those types into some SDE mechanism to exposes them.

4. Service Policy

Another fundamental aspect of service-based grid computing is service policy. Service policy is a set of declarative statements by the service that describe the allowable use of the service (who can use the service, how the service is to be invoked, etc.). For example, a service might require that all request messages be encrypted with a Kerberos token, or signed with an X509 certificate issued by a particular CA. Service policy represents the Grid Service author's description of the requests that the container should "allow in" and hence which requests should be screened out (typically throwing an exception), without invoking any of the service's logic. Here, service policy is used as directives to the container. Recall that a service can always implement its own defacto policy internally by examining requests within the service logic and returning an exception for any non-compliant requests.

Currently, we are experimenting with allowing service policy to be specified in OGSI.NET using a set of attributes placed on the service class. These attribute might be used in a variety of ways. A static analysis

[1] Currently, we provide handlers to automatically save and load SDE values to permanent storage and to timestamp SDEs when their values are altered/created.

tool, akin to the WSDLGenerator, could create a static policy file using WS-Policy [1] to be used by the Web Service Extensions (WSE) pipeline [8]. This file would be used to configure the pipeline filters

```
public class SomeServicePortType : GridServiceSkeleton
{
	SomeServicePortType() { … }

	[SDE]
	public int foo;
```

Figure 5. SDE Attributes on Public Data Members

distributed with the WSE. A second option is to generate a new WSE filter that enforces the service's policy and install that filter in the service's pipeline. The OGSI.NET container uses a separably configurable WSE pipeline for each Grid Service instance, allowing custom pipeline filters for each service. A final option is to have the container itself check that client requests conform to the stated policy. The difference between this and the former options is that the "in pipeline" processing typically returns an exception to the requestor for a non-policy compliant request without ever activating any code in the container or service itself. If the container were to process requests, it could take any number of actions when non-compliant requests are received. Additionally, the policy support provided by the current WSE pipeline (v2.0) is mostly for security policy. While generating new pipeline filters may allow other types of policy to be processed, such as those specific to grid computing, deciding whether or not a request complies with a service's policy may require dynamic information that is available only to the container itself and not to the pipeline.

We believe that a simple set of attributes will allow Grid Service authors to express desired service policies without having to create separate policy configuration files. OGSI.NET defines the following policy attributes: `Integrity`, `Encryption`, `Token`, and `Message`, which are shown in Figure 6. These attributes are based on the capabilities of the WSE 2.0 [8].

```
[Integrity("required", "body", "token1, token2")]
[Confidentiality("rejected", "header1", "token3")]
[Message("required", "age=30")]
[Token("token1", "x509v3", "subject=Bob Smith, CA=UVA")]
[Token("token2", "Kerberosv5ST")]
[Token("token3", "SecurityContextToken",
        "issuer=http://token.virginia.edu/mytoken")]
public class Service1 : GridServiceSkeleton
{
```

Figure 6. Policy Attributes Applied to a Service Class

The `Integrity` attribute allows the service to specify digital signing requirements for messages it receives. There are three parameters called usage, messageParts and token. The usage parameter can be one of `required` (meaning the message will be rejected if it *does not* comply with this requirement) or `rejected` (meaning the message will be rejected if it *does* comply). The messageParts parameter defines the portions of the message that must be signed and can specify specific header elements and/or the message body. Finally, the token parameter defines the acceptable tokens that can be used for the signature. These are further defined within `Token` attributes (see below). The `Integrity` attribute in Figure 6 defines a policy where all requests to this service must have their message bodies signed by either token1 or token2.

The `Confidentiality` attribute uses the same parameters as the `Integrity` attribute, but it specifies the encryption requirements for incoming messages. The `Confidentiality` attribute in Figure 6 specifies a policy in which any message with header1 encrypted by token3 will be rejected. The `Token` attribute describes the security tokens that are referenced in the `Integrity` or `Confidentiality` attributes. This was made a separate attribute (rather than a set of additional parameters for the other two attributes) so that Grid Service authors can make multiple references to the same token requirements. The `Token` attribute takes 3 parameters, the token name, the token type and a type-specific parameter string.

The token name is the identifier used in other policy attributes to refer to this token requirement description. Token type can be one of "x509v3" (for X509 certificates), "Kerberosv5ST" (for Kerberos tickets), "Username" (for Windows username/ password tokens) and "SecurityContextToken" (for symmetric key tokens such as those issued by a service implementing WS-Secure Conversation [2]). The interpretation of the final parameter depends on the token type. The currently supported parameter values are "subject", "CA" and "issuer". The "subject" string requires that the token have the specified subjectDN, in the case of an x509 certificate, or the specified username in the case of a Username token. This parameter has no meaning for other token types. The "CA" string requires x509 certificates to be signed by the specified root certificate authority. Finally, the "issuer" string requires that SecurityContextTokens be issued by the service whose GSH appears in the parameter. In Figure 6, token1 is an x509 certificate that must have the subject DN "Bob Smith" and have an issuer DN of "UVA" for some certificate in the certificate chain. Token2 must be a kerberos token and token3 must be a SecurityContextToken from the token issuing service at http://token.virginia.edu/mytoken.

The final policy attribute, `Message`, specifies other requirements for the message. This attribute takes the usage parameter, similar to the other policy attributes, and a configuration string. The only currently supported parameter in this string is "age", which specifies the maximum age of the request message in seconds. In Figure 6, the service sets its policy to not except messages that are older than 30 seconds.

5. Conclusion

In order for Grid Service authors to turn a set of source logic into a Grid Service, a collection of meta-information is needed. This meta-data must specify the methods and data that the service exposes as well as the policy by which they are exposed. We believe that the best way for programmers to express this meta-data is with simple annotations to the service logic. Such annotations free the programmer from having to generate potentially multiple external configuration files for each service. The annotations can be processed by static analysis tools and the container's runtime system to transform the author's service logic into a Grid Service without requiring the author to have a detailed knowledge of the container.

.NET provides a mechanism for annotating code using attributes. Attributes are annotations that can be placed on any object (e.g. class, method, data member, etc.) and can be recovered at runtime through .NET's reflection interface. The attributes provided in OGSI.NET attempt to strike a balance between powerful expressibility and ease of use. OGSI.NET's `SDE` attributes allow full specification of service data elements as defined in the OGSI specification [7]. However, the simple `[SDE]` annotation can also be used to quickly specify common use cases where SDE parameters can be inferred from the data type so annotated. OGSI.NET's attribute-based programming model provides an easy-to-use mechanism for describing many of the common aspects of the service-based model of grid computing that Grid Service authors need to turn their service logic into operational Grid Services.

6. References

[1] Microsoft, IBM, BEA, SAP. 2003. Web Services Policy Framework. http://msdn.microsoft.com/ webservices/default.aspx?pull=/library/en-us/dnglobspec/html/ws-policy.asp

[2] IBM, Microsoft, Verisign and RSA Security. 2002. Web Service Secure Conversation Language. http://msdn.microsoft.com/ webservices/default.aspx?pull=/library/en-us/dnglobspec/html/ ws-secureconversation.asp

[3] .NET Framework Class Library. 2003. MSDN Documentation. http://msdn.microsoft.com/ library/default.asp?url=/library/en-us/cpref/html/ frlrfsystemappdomainclasstopic.asp

[4] .NET Framework Class Library. 2003. MSDN Documentation. http://msdn.microsoft.com/ library/default.asp?url=/library/en-us/cpguide/ html/cpconbasicserialization.asp

[5] Open Grid Services Architecture Working Group. 2003. Global Grid Forum. https://forge.gridforum.org/projects/ogsa-wg.

[6] Open Grid Services Infrastructure Working Group. 2003. Global Grid Forum. https://forge.gridforum.org/projects/ogsi-wg.

[7] Teucke, S., Czajkowski, C., Foster, I., Frey, J., Graham, S., Kesselman, C., Maquire, T., Sandholm, T., Snelling, D., and Vanderbilt, P. 2003. Final OGSI Specification v1.0. Global Grid Forum. https://forge.gridforum.org/docman2/ ViewProperties.php?group_id=43&document_content_id=347

[8] Web Service Enhancements 2.0. 2003. Microsoft. http://msdn.microsoft.com/webservices/building/ wse/