GFD-R-P.xxx
Category: Recommendation
GLUE Working Group
http://redmine.ogf.org/projects/glue-wg

Authors:
Sergio Andreozzi°, INFN
Stephen Burke, RAL
Felix Ehm, CERN
Laurence Field, CERN
Gerson Galang, ARCS
Balazs Konya, Lund University
Maarten Litmaath, CERN
Shiraz Memon*, FZJ
David Meredith, STFC
Paul Millar, DESY
JP Navarro*
Florido Paganelli
Warren Smith
Adrian Taga, Oslo University
*co-chairs
°editor

Jan 06, 2013

GLUE v. 2.0 – Reference Realization to XML Schema

Status of This Document

This document provides information to the Grid community regarding the realization of the GLUE information model (v.2.0) as XML Schema. Distribution is unlimited. This realization is derived from the proposed recommendation of the specification document [glue-2].

Copyright Notice

Abstract

The GLUE 2 specification is an information model for Grid entities described in natural language enriched with a graphical representation using UML Class Diagrams. This document presents a realization of this information model as XML Schema.

Contents

1. About this Document 3
2. Notational Conventions 3
3. XML Schema Realization 3
 3.1 Namespace 4
 3.2 Document Root Element (`Entities`) 4
 3.3 Entity Elements 6
 3.4 Enumerations 6
 3.5 Associations 7
  3.5.1 Associations Elements and ID Reference Elements 7
  3.5.2 Document Style Choice - Flat vs. Nested 9
  3.5.3 Traversing Associations and Document Validation 9
 3.6 Inheritance 10
  3.6.1 Abstract Classes and Element Implementations 10
  3.6.2 Abstract Element Base Type 10
  3.6.3 BaseType Attribute 11
  3.6.4 Substitution Group 11
 3.7 Importing the GLUE 2 Schema to Define Extended Custom Sub Types 13
 3.8 Custom Associations 13
 3.9 Extensibility 16
 3.10 The Normative XML Schema Realization of GLUE 2.0 17
4. Security Considerations 17
5. Author Information 18
6. Contributors & Acknowledgements 19
7. Intellectual Property Statement 19
8. Disclaimer 19
9. Full Copyright Notice 19
10. References 21

1. About this Document

The GLUE 2.0 Information model defined in [glue-2] is a conceptual model of Grid entities. In order to be adopted by technology providers, a realization in terms of a concrete data model is needed. This document provides the normative realization of the GLUE 2.0 conceptual model in terms of an XML Schema (XSD). The document also elaborates on the design choices adopted to map the entities and relationships of the conceptual model into the concrete data model.

This document is maintained by OGF's GLUE Working Group, which signs responsible for documenting errata and releasing revisions as defined by the OGF document process. Errors and feedback in general should be directed to the GLUE WG mailing list, at `<glue-wg@ogf.org>`.

2. Notational Conventions

The key words 'MUST," "MUST NOT," "REQUIRED," "SHALL," "SHALL NOT," "SHOULD," "SHOULD NOT," "RECOMMENDED," "MAY," and "OPTIONAL" are to be interpreted as described in [rfc-2119].

3. XML Schema Realization

There are many possible approaches to map the GLUE 2 conceptual model into an XML schema. Depending on which aspects are important, different design choices are preferable. The core design characteristics of this rendering include:

- A single Document Root element (`Entities`), which serves as a global element bag (see Listing 1).
- `Entities` nests child elements as siblings in an established order. The nested child elements represent the core GLUE 2 entity classes. The multiplicity of all the entity elements is zero-to-many.
- Associations between sibling elements are modeled using element ID references rather nesting elements into parent-child relationships. There are a number of functional justifications for choosing a 'flat' document style over a nested/hierarchical style described in Section 3.5.
- To fully implement the GLUE 2 conceptual model, the XSD defines a number of abstract elements that correspond to the GLUE 2 abstract entity classes. These abstract elements allow different (concrete) element implementations to be derived and substituted into the document. For example, `Service`, `ComputingService` and `StorageService` may substitute for `AbstractService`. The element implementations that are valid according to this schema are given in Listing 1. Abstract element inheritance is described in Section 3.6.
- Importantly, new element specializations may be profiled in future and may be nested within `Entities` without requiring subsequent modification to this schema. The process for importing this XSD and deriving custom element specializations is described in Section 3.7
- All entity elements are defined globally within the XSD. In doing this, the GLUE 2 entity elements can be imported and reused wherever is required within other XML documents without having to conform to the constraints and ordering defined by `Entities`.
- Additional elements defined in other namespaces can be nested in pre-defined extension points.

3.1 Namespace

The Open Grid Forum published a document with guidelines for identifying names uniquely and uniform in the GGF/OGF domain [ogf-ns]. Based on this document, we have adopted the following namespace for the XML Schema realization of GLUE 2.0:

```
GLUE-XSD-NS ::= 'http://schemas.ogf.org/glue/' YYYY '/' MM '/spec_' M.N '_r' R
```

- `YYYY`: year of the normative document of the GLUE specification
- `MM`: month of the normative document of the GLUE specification
- `M.N`: M is the major version and N is the minor version of the GLUE conceptual model
- `R`: component to be used to specify the revision number of the XSD realization; this number SHOULD be incremented each time a new non-backwards compatible version is published

As a non-normative example, the namespace for the first release of the XSD document for the final GLUE 2.0 specification [glue-2] is:

```
To be updated on publishing:
```
~~http://schemas.ogf.org/glue/2009/03/spec_2.0_r1~~

3.2 Document Root Element (`Entities`)

- `Entities` is the only recognized Document Root element. For full interoperability, instance documents MUST define `Entities` as the Document Root.
- Entity elements are nested as siblings in an order which is determined according to the location of abstract elements defined within `Entities`.
- Different concrete elements MAY substitute for an abstract class in an unspecified order. For example, when substituting for the abstract `Domain` class, `AdminDomain` may appear before `UserDomain` and vice-versa.
- This imposes an element ordering which is more loosely defined compared to ordering by concrete elements. However, in doing this new entities may be conveniently substituted into predictable locations according to the placement of abstract classes. This provides a more strict element ordering compared to substituting new implementations in place of `xsd:any` extension points. Abstract classes also allow newly derived entities to be equal siblings within `Entities`.
- Entity elements that have no abstract super-class are placed according to their specific location within `Entities`.
- The multiplicity of all entity elements within `Entities` is zero-to-many.

Listing 1. Simplified GLUE 2 XML document structure. `Entities` is the document root. Entity elements are ordered according to the location of abstract elements defined within `Entities`. The star character (*) signifies a zero-to-many multiplicity.

```
<Entities>
       <!-- Locations and Contacts first: -->
       <Location/> *
       <Contact/> *

       <!–
       Abstract elements are defined in the XSD in the following order:
          <Domain/>
          <AbstractService/>
          <AbstractEndpoint/>
          <Share/>
          <Manager/>
          <Resource/>
          <AbstractActivity/>
          <Policy/>
       -->
       <!-- Elements that implement abstract <Domain/>
            (may substitute in any order): -->
       <AdminDomain/> *
       <UserDomain/> *
       <!-- Elements that implement abstract <AbstractService/>
            (may substitute in any order): -->
       <Service/> *
       <ComputingService/> *
       <StorageService/> *
       <!-- Elements that implement abstract <AbstractEndpoint/>
            (may substitute in any order): -->
       <Endpoint/> *
       <ComputingEndpoint/> *
       <!-- Elements that implement abstract <Share/>
            (may substitute in any order): -->
       <ComputingShare/> *
       <StorageShare/> *
       <!-- Elements that implement abstract <Manager/>
            (may substitute in any order): -->
       <ComputingManager/> *
       <StorageManager/> *
       <!-- Elements that implement abstract <Resource/>
            (may substitute in any order): -->
       <ExecutionEnvironment/> *
       <DataStore/> *
       <!-- Elements that implement abstract <AbstractActivity/>
            (may substitute in any order): -->
       <Activity/> *
       <ComputingActivity/> *
       <!-- Elements that implement abstract <Policy/>
            (may substitute in any order): -->
       <AccessPolicy/> *
       <MappingPolicy/> *

       <!-- Other concrete element references: -->
       <Benchmark/> *
       <ApplicationEnvironment/> *
       <ToComputingService/> *
       <ToStorageService/> *
       <StorageAccessProtocol/> *
       <StorageServiceCapacity/> *
       <StorageShareCapacity/> *
       <ApplicationHandle/> *
       <xsd:any/> *
</Entities>
```

3.3 Entity Elements

- Each UML class (or 'Entity' in GLUE 2 nomenclature) of the conceptual model maps into an XML element definition (an 'entity element'). These are given in Listing 1. For a comprehensive description of the attributes and semantics of each entity, please refer to the conceptual model [glue-2].
- In general, each attribute of a UML class in the conceptual model maps into an XML element definition (this is a general rule and applies also to both `ID` and `LocalID` attributes); an exception is made for the attributes `CreationTime` and `Validity` of the `Entity` class. Since they can be considered as metadata about GLUE-based description of entities, they are modeled as XML attributes.
- If a class or an attribute can be instantiated multiple times, then a separate XML element for each instance MUST be created.

As additional information, it should be noted that:

- Attributes which type is a timestamp are typed using `glue:DateTime` which is a restriction of the `xsd:dateTime` simple type to match the UTC Timezone: `yyyy'-'mm'-'dd'T'hh':'mm':'ssZ`
- If an information producer cannot define a value for a mandatory attribute, then you SHOULD use the placeholder values defined (see Annex A in [glue-2])

3.4 Enumerations

The GLUE 2 specification defines a set of attributes that form an enumeration. These enumerations belong to two main categories:

- Closed enumeration: a restricted list of values where the value of the attribute MUST belong to the set of defined values.
- Open enumeration: an unrestricted list of values where the value of the attribute MAY belong to the set of defined values. An open enumeration offers a partial list of values with defined semantics among which to choose. It also provides hints on how new values may be defined.

Closed enumerations are modeled as restrictions on a base type. By using the element `<enumeration>`, each allowed value can be defined. An element which type is a restricted string type in terms of a set of values is valid if and only if the value matches one of those defined. The following XSD fragment defines the enumeration for the `Endpoint.HealthState` attribute:

Listing 2. Sample schema for a Closed Enumeration type with a limited value range (`Endpoint.HealthState` attribute).

```
<simpleType name="EndpointHealthState_t">
    <restriction base="string">
        <enumeration value="critical"/>
        <enumeration value="ok"/>
        <enumeration value="other"/>
        <enumeration value="unknown"/>
        <enumeration value="warning"/>
    </restriction>
</simpleType>
```

For open enumerations, the natural approach would be to use the `union` capability of XSD [xsd-oe, xsd-ap]. Unfortunately, this is not well supported in current implementations of XML software libraries; therefore we decided to model them by using the `annotation` element. Each enumeration value is within an `appinfo` sub-element. Software validating an XML document according to the defined XSD for GLUE 2 SHOULD be instrumented in order to consider these values. The following XSD fragment presents the definition of the open enumeration for the `DataStore.Type` attribute:

Listing 3. Sample schema for an Open Enumeration type (`DataStore.Type` attribute).

```
<simpleType name="DataStoreType_t">
    <restriction base="string">
        <annotation>
            <appinfo>
                <enumeration value="disk"/>
                <enumeration value="optical"/>
                <enumeration value="tape"/>
            </appinfo>
        </annotation>
    </restriction>
</simpleType>
```

3.5 Associations

In the conceptual model, several associations are represented. They can be classified in terms of the multiplicity (one-to-one, one-to-many, many-to-many), in terms of the navigability (bi-directional, unidirectional) or in terms of the association type (binary, aggregation, composition, association class).

3.5.1 Associations Elements and ID Reference Elements

Associations between entities are modeled using ‘ID reference elements.’ Each of these elements refers to the unique ID value of a related element. ID reference elements are named after the referenced class with the suffix “ID”. They are grouped together under each entity’s `Associations` element. For instance, the `ComputingShare` can be linked to many `ComputingEndpointS`, many `ExecutionEnvironmentS` and a single `ComputingService`. The corresponding ID reference elements are therefore `ComputingEndpointID`, `ExecutionEnvironmentID and ComputingServiceID` as shown in Listing 4.

The `Associations` element is mandatory for every entity element and so MUST always be rendered. If a given entity has no mandatory relationships, then `Associations` MAY be empty provided the entity has no other optional relationships. An example of an empty `Associations` element is shown in Listing 5 which defines an orphan `Contact`. By mandating `Associations`, the entity’s relationships are clearly defined. If `Associations` itself was made optional, it would be possible to erroneously omit the `Associations` element even if relationships existed. If an entity element has associations with other elements, then the corresponding ID reference elements MUST be rendered within `Associations`.

Listing 4. Associations between entity elements are modeled using ID reference elements nested within the mandatory `Associations` element. ID reference elements specify the Id of related elements (as defined in their corresponding <ID>).

```
<Entities ...>
   ...
   <ComputingService BaseType="Service">
        <ID>urn:myservice1</ID> ...
   </ComputingService>

   <ComputingEndpoint BaseType="Endpoint">
        <ID>urn:myendpoint1</ID> ...
   </ComputingEndpoint>

   <ComputingEndpoint BaseType="Endpoint">
        <ID>urn:myendpoint2</ID> ...
   </ComputingEndpoint>

   <ComputingShare BaseType="Share">
        <ID>urn:mycomputingsshare1</ID>
        <ServingState>production</ServingState>
        <Associations>
            <ComputingEndpointID>urn:myendpoint1</ComputingEndpointID>
            <ComputingEndpointID>urn:myendpoint2</ComputingEndpointID>
            <ExecutionEnvironmentID>urn:myexenv1</ExecutionEnvironmentID>
            <ComputingServiceID>urn:myservice1</ComputingServiceID>
        </Associations>
    </ComputingShare>

   <ExecutionEnvironment BaseType="Resource">
        <ID>urn:myexenv1</ID> ...
   </ExecutionEnvironment>
...
</Entities>
```

Listing 5. The Associations element must always be rendered. If an entity element has no mandatory relationships, then its `Associations` element MAY be empty if no other optional associations exist.

```
    <Contact>
        <ID>urn:contact1</ID>
        <Detail>http://some.uri/embedding/contact/info</Detail>
        <Type>general</Type>
        <!-- This contact has no relationships,
        but Associations element must still be rendered as empty -->
        <Associations/>
    </Contact>
```

3.5.2 Document Style Choice - Flat vs. Nested

The 'flat' document style was chosen over a nested (hierarchical) approach for the following reasons.

- The GLUE 2 information model is not a pure tree structure. Elements with multiple parents and many-to-many relationships cannot be modeled by nesting alone. In these types of relationship, either element duplication or mixing both element ID references with nesting is necessary. For example, if the class participating in the "many" side of the relationship also participates in other associations, then only one of those associations can be mapped into a nested parent-child relationship. The other relationships must be modeled using element ID references. This style mixing adds considerable complexity. Conversely, modeling associations using ID reference elements imposes no such limitations and is consistent.

- Element nesting defines a strong parent-child relationship, where the life span of the child is strongly linked to that of the parent (UML composition). However, the entity relationships in GLUE 2 represent weaker UML aggregations where entities can exist in isolation with their own independent life spans. This is better suited to ID referencing approach.

- The flat style is more efficient when rendering the results of projection queries. Projection queries 'slice' the data and specify which information should be returned in the result set. In SQL queries, columns are typically specified but in our case we select entities, for example '`select from Endpoint where Endpoint.Type = 'X'`'). The efficiency advantage provided by the flat approach is due to the loose coupling provided by ID reference elements and the zero-to-many multiplicity of elements within `Entities`:

    - ID reference elements are lightweight – they establish that an element *has* immediate relationships with other elements without having to fully populate and render those elements.
    - Conversely, the nested approach must fully render its child and parent elements in order to show that these relationships actually exist. Fully populating those relations would be required for the sake of completeness; if a related element was not fully populated the data would be incomplete and subject to misinterpretation (elements MUST always be rendered in full including their immediate relationships). Furthermore, the nested approach is inherently recursive and so would require cascading to all decedents and ancestors in an entity graph.

- Minimizing XML bloat is important, especially when considering potentially thousands of entities listed by an information system in a large scale grid. Since the flat style is more efficient, it is more suitable for use by information systems which are the intended primary implementations of this schema.

- The loose coupling provided by element ID references means that query results can be split across multiple documents. For example, one `Entities` document could provide a list of service endpoints while another `Entities` document could provide contact information. This allows potentially sensitive information to be split into different documents. This would not be possible using nesting which defines strong parent-child relationships (composition) where all relations need to be captured in a single document for the sake of completeness (as described above).

3.5.3 Traversing Associations and Document Validation

ID referencing requires the associations are checked for correctness during document validation and when un-marshalling from XML into objects. To do this, the element IDs identified in the `Associations` element must be cross referenced to ensure they refer to the expected element types. This extra validation step is not enforced by the XSD schema. For example, it is necessary to check that a `ComputingEndpointID` element correctly references a `ComputingEndpoint` instance. If the referenced element is not of the required type, then the implementation MUST return an error.

3.6 Inheritance

3.6.1 Abstract Classes and Element Implementations

The main entity classes described in the conceptual model are defined as abstract XSD schema elements. These serve as global extension points and cannot be directly created in an XML instance document. For each abstract element, one or more implementation elements defined in the GLUE 2 namespace can be substituted in place of the abstract element. The abstract elements and their corresponding GLUE 2 implementations are listed in Table 1.

If the standard GLUE 2 element set does not adequately describe a Grid information model, the abstract elements can be substituted for new custom sub-type element specializations.

- Alternative element implementations MUST be defined in a new namespace (see Section 3.7). In doing this other standards, extending profiles and end users MAY define new substitutable elements that better describe their Grid entities as required. This follows the GLUE 2 conceptual model which was designed to facilitate extension and specialization of the core entity classes within a particular rendering. Indeed, the conceptual model states that for "Grid [entities] requiring a richer set of attributes, specific models MAY be derived by specializing from the [entity] class and adding new properties or relationships."
- Defining new sub-type element specializations allows new information models to be constrained using strongly typed XSD documents rather than solely relying on the loosely typed GLUE 2 extensibility mechanisms (e.g. string based key-value property bags).
- If new sub-type element specializations are defined, XML instance documents will need to be validated against both the base GLUE 2 XSD and the extending XSD schema(s).
- Since new sub-types must be defined under a new namespace, XML instance documents will NOT be GLUE 2 compliant unless those new types are profiled and subsequently incorporated into the GLUE specification at a later date.

3.6.2 Abstract Element Base Type

All substituting elements MUST either be the same as, or be derived from the same base type as the abstract element. This is enforced by the W3C XML Schema rule set. A substituting element MUST therefore implement;

a) The mandatory elements and attributes defined by the abstract element's base type.
b) The mandatory elements and attributes added by the extending sub-type (if any).
c) Selected optional elements and attributes added by the extending sub-type (if any).

Listing 6. The `AbstractService` element defines `Service_t` as its (base) type. This is specified in the GLUE 2 schema with the following excerpt:

```
<!-- An abstract service base type that is designed to be implemented/extended by
concrete service implementations. Service implementations must use
substitutionGroup="glue:AbstractService" -->

<element name="AbstractService" type="glue:ServiceBase_t" abstract="true"/>
```

Table 1: Abstract elements and their corresponding implementations. Each abstract element also defines a fixed BaseType attribute value to identify the base type.

Abstract Element	BaseType Attribute Value	GLUE 2 Implementation Elements
Domain	**Domain**	AdminDomain UserDomain
AbstractService	**Service**	Service ComputingService StorageService
AbstractEndpoint	**Endpoint**	Endpoint ComputingEndpoint StorageEndpoint
Share	**Share**	ComputingShare StorageShare
Manager	**Manager**	ComputingManager StorageManager
Resource	**Resource**	ExecutionEnvironment DataStore
AbstractActivity	**Activity**	Activity ComputingActivity
Policy	**Policy**	AccessPolicy MappingPolicy

3.6.3 BaseType Attribute

We also consider the possibility of querying all sub-types of a specific abstract super-class. In order to simplify this type of query, we introduce an XML attribute called `BaseType` whose value is fixed and corresponds to the name of the abstract super-class. This attribute is defined for all the entities and are listed in Table 2. Sample XPath queries are given below for querying for all types of Endpoint, Service and Domain.

Listing 8. Sample XPath queries with the `BaseType` attribute.

```
/Entities/*[@BaseType='Endpoint']
/Entities/*[@BaseType='Service']
/Entities/*[@BaseType='Domain']
```

3.6.4 Substitution Group

In an XML instance document, the GLUE 2 abstract elements can be substituted for any element that defines a corresponding `xsd:substitutionGroup` (and which derives from the same base type as the abstract element). For example, in the GLUE 2 XSD, the `AbstractService` element can be substituted for the elements that define the `AbstractService` substitution group. When validating an XML instance document against the GLUE 2 schema, the allowed substitutable elements include; `Service`, `ComputingService` or `StorageService`. This is specified in the GLUE 2 schema by the following excerpt. Notice that all of the substitutable elements also extend the `ServiceBase_t` complex type as this is the base type of `AbstractService`.

Listing 7. Elements that can substitute for an abstract element must extend the relevant base type and specify the appropriate substitution group. In this example, `Service`, `ComputingService` and `StorageService` all derive from `ServiceBase_t` type which is the base type of `AbstractService`.

```
<element name="AbstractService" type="glue:ServiceBase_t" abstract="true"/>

<!-- Concrete Service implementations that substitute AbstractService
    must be the same as, or be derived from, a ServiceBase_t type.  -->
<element name="Service" type="glue:Service_t"
substitutionGroup="glue:AbstractService"/>

<element name="ComputingService" type="glue:ComputingService_t"
substitutionGroup="glue:AbstractService"/>

<element name="StorageService" type="glue:StorageService_t"
substitutionGroup="glue:AbstractService"/>

...

<complexType name="Service_t">
     <complexContent>
          <extension base="glue:ServiceBase_t">
               <sequence>
                    ...elided...
               </sequence>
          </extension>
     </complexContent>
</complexType>

<complexType name="ComputingService_t">
     <complexContent>
          <extension base="glue:ServiceBase_t">
               <sequence>
                     ...elided...
               </sequence>
          </extension>
     </complexContent>
</complexType>

<complexType name="StorageService_t">
     <complexContent>
          <extension base="glue:ServiceBase_t">
               <sequence>
                   ...elided...
               </sequence>
          </extension>
     </complexContent>
</complexType>
```

3.7 Importing the GLUE 2 Schema to Define Extended Custom Sub Types

In order to derive custom elements that MAY substitute for the GLUE 2 abstract elements, the GLUE 2 XSD schema must be imported into a third party schema using an `xsd:import`. New elements can then be defined under the new target namespace of the extending schema.

As described above, new element implementations MUST;

- Be the same as, OR be derived from the same type as the abstract super class.
- Specify the name of the extended abstract element using an `xsd:substitutionGroup`.

An example XSD schema that imports the GLUE 2 schema to derive new sub-types is given in Listing 9. This schema defines the custom `MonitoredXService` type which provides supplementary monitoring information. Notice that the custom `MonitoredXService` specifies the GLUE 2 `AbstractService` element in its `substitutionGroup` and also extends from the GLUE 2 `ServiceBase_t` complex type.

A corresponding XML instance document is given in 10. Notice that the document references both the GLUE 2 XML schema and the extending schema within its document root definition. Also notice that the extending elements are qualified with the '`ext`' namespace prefix that identifies the namespace of the extension schema; '`http://www.extensions.ac.uk/sample`'. A similar approach can be taken for sub-typing all the other abstract elements whenever necessary.

3.8 Custom Associations

The example in Listings 9 and 10 demonstrates an important design feature; an entity's `Associations` element is not inherited from its super class. This is intentional and allows the sub-type to define an alternative strategy for modeling relationships. For the purpose of illustration, the `MonitoredXService` defines its `MonitoredXEndpoint` as a nested child element (rather than using an ID reference element). Indeed, this could be exploited in future to define an alternative set of GLUE 2 entity elements that use element nesting rather than ID references.

Listing 9 Sample XSD schema that imports the GLUE 2 schema and extends both the `AbstractService` and `AbstractEndpoint` elements with custom sub-types. A corresponding sample XML instance document is given in Listing 10.

```
<?xml version="1.0"?>
<xs:schema version="1.0"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:glue="http://schemas.ogf.org/glue/2009/03/spec_2.0_r1"
           xmlns:ext="http://www.gocdbextensions.ac.uk/sample"
           xmlns="http://www.gocdbextensions.ac.uk/sample"
           targetNamespace="http://www.gocdbextensions.ac.uk/sample">

    <!-- This XSD imports the base glue2 XSD and extends AbstractService and
    AbstractEndpoint in order to derive custom Service and Endpoint types. -->
    <xs:import namespace="http://schemas.ogf.org/glue/2009/03/spec_2.0_r1"
                  schemaLocation="glue2_2.xsd"/>

    <!-- For demonstration purposes, the MonitoredXEndpoint is defined as a
    child of the service so that the lifetime of the endpoint is strictly tied to
    its parent service.-->
    <xs:element name="MonitoredXService" substitutionGroup="glue:AbstractService">
        <xs:complexType>
            <xs:complexContent>
                <xs:extension base="glue:ServiceBase_t">
                    <xs:sequence>
        <xs:element ref="ext:Monitored" minOccurs="1" maxOccurs="1"/>
        <xs:element ref="ext:Beta" minOccurs="1" maxOccurs="1"/>
        <xs:element ref="ext:MonitoredXEndpoint" minOccurs="1" maxOccurs="unbounded"/>
                    </xs:sequence>
                </xs:extension>
            </xs:complexContent>
        </xs:complexType>
    </xs:element>

    <xs:element name="Monitored" type="glue:ExtendedBoolean_t" />
    <xs:element name="Beta" type="glue:ExtendedBoolean_t" />

    <!-- MonitoredXEndpoint must be the same as, or be derived from AbstractEndpoint.
    It adds supplementary monitoring information. -->
    <xs:element name="MonitoredXEndpoint" substitutionGroup="glue:AbstractEndpoint">
        <xs:complexType>
            <xs:complexContent>
                <xs:extension base="glue:EndpointBase_t">
                    <xs:sequence>
          <xs:element ref="ext:DowntimeClassification" minOccurs="1" maxOccurs="1"/>
          <xs:element ref="ext:DowntimeSeverity" minOccurs="1" maxOccurs="1"/>
                    </xs:sequence>
                </xs:extension>
            </xs:complexContent>
        </xs:complexType>
    </xs:element>

    <xs:element name="DowntimeClassification" type="ext:DowntimeClassification_t"/>
    <xs:element name="DowntimeSeverity" type="ext:DowntimeSeverity_t"/>

    <xs:simpleType name="DowntimeSeverity_t">
        <xs:restriction base="xs:string">
            <xs:enumeration value="OUTAGE"/>
            <xs:enumeration value="WARNING"/>
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="DowntimeClassification_t">
        <xs:restriction base="xs:string">
            <xs:enumeration value="SCHEDULED"/>
            <xs:enumeration value="UNSCHEDULED"/>
        </xs:restriction>
    </xs:simpleType>
</xs:schema>
```

Listing 10. Sample document that imports the both the GLUE 2 XSD and the sample XSD in Listing 9 in order to nest custom `MonitoredXService` and `MonitoredXEndpoint` types within `Entities`.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!--
The custom elements can be nested in the glue Entities element. This requires
no modification to the glue 2 xsd, but this doc must be validated against both
the base glue2 xsd and the extending xsd. -->
<Entities
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xmlns:ext='http://www.gocdbextensions.ac.uk/sample'
    xmlns='http://schemas.ogf.org/glue/2009/03/spec_2.0_r1'
    xsi:schemaLocation='
    http://www.gocdbextensions.ac.uk/sample sampleGlue2_2Extension.xsd
    http://schemas.ogf.org/glue/2009/03/spec_2.0_r1 glue2_2.xsd'>

    <AdminDomain BaseType="Domain">
        <ID>124</ID>
        <Associations>
            <ServiceID>urn:mymonitoredXservice</ServiceID>
            <ServiceID>urn:mystandardService</ServiceID>
        </Associations>
    </AdminDomain>

    <Service BaseType="Service">
        <ID>urn:mystandardService</ID>
        <Type>NormalService</Type>
        <QualityLevel>production</QualityLevel>
        <Associations>
            <EndpointID>123</EndpointID>
        </Associations>
    </Service>

    <!--
    Our custom service type is substitutable for the AbstractService.
    We can therefore nest this type of service within Entities.
    This allows future glue profiles to define new services/endpoints without
    needing to modify the base GLUE 2 schema.
    -->
    <ext:MonitoredXService BaseType="Service">
        <ID>urn:mymonitoredXservice</ID>
        <Type>ServiceX</Type>
        <QualityLevel>production</QualityLevel>
        <ext:Monitored>true</ext:Monitored>
        <ext:Beta>true</ext:Beta>
        <ext:MonitoredXEndpoint BaseType="Endpoint">
            <ID>12</ID>
            <URL>adf</URL>
            <InterfaceName>adf</InterfaceName>
            <QualityLevel>development</QualityLevel>
            <HealthState>ok</HealthState>
            <ServingState>production</ServingState>
            <ext:DowntimeClassification>SCHEDULED</ext:DowntimeClassification>
            <ext:DowntimeSeverity>OUTAGE</ext:DowntimeSeverity>
        </ext:MonitoredXEndpoint>
    </ext:MonitoredXService>

    <Endpoint BaseType="Endpoint">
        <ID>123</ID>
        <URL>uri://some.url.ac.uk/service</URL>
        <InterfaceName></InterfaceName>
        <QualityLevel>development</QualityLevel>
        <HealthState>ok</HealthState>
        <ServingState>production</ServingState>
        <Associations>
            <ServiceID>urn:mystandardService</ServiceID>
        </Associations>
    </Endpoint>
</Entities>
```

3.9 Extensibility

In the conceptual model, two main "hooks" are provided for extensions: the `Extension` class and the `OtherInfo` attribute (see Section 5.1 [glue-2]). In the XML Schema mapping, the `Extension` class is mapped as a child of the `Extensions` element. The `OtherInfo` attribute is mapped as an `OtherInfo` element. They are both available in all entity elements for the purposes of extension.

The XML Schema also adds additional extension points using the `xsd:any` element at specific locations within the document. Elements and attributes belonging to other namespaces may be substituted in place of the `xsd:any` element (we use the `lax` value for the `processContent` attribute of the `xsd:any` element and `##other` for the `namespace` attribute). The `xsd:any` element is provided in the `Extensions`, `Extension` and `Entity` elements. In the following example, we present a fragment showing how the extensibility options can be used:

Listing 11. Options for extension include the `OtherInfo`, `Extensions.Extension` elements and the `xsd:any` extension element.

```
<Entities...>
    ...
    <ExecutionEnvironment BaseType="Resource">
        <ID>executionEnvironment1</ID>
        <!-- Zero to many OtherInfo elements -->
        <OtherInfo>This is a powerful GPU system</OtherInfo>
        <OtherInfo>So is this one</OtherInfo>

        <!-- Extensions nests zero to many Extension elements -->
        <Extensions>
            <Extension>
                <LocalID>GeForge</LocalID>
                <Key>GeForge</Key>
                <Value>GeForge 7</Value>
            </Extension>

            <Extension>
                <LocalID>CoreLib</LocalID>
                <Key>CoreLib</Key>
                <Value>glibc:3.4.9</Value>
                <!-- xsd:any within Extension allows elements from other namespaces-->
                <typ:TextInfo xmlns:typ="http://unigrids.org/2006/04/types">
                    <typ:Name>StagingInPath</typ:Name>
                    <typ:Value>/user-home/in</typ:Value>
                </typ:TextInfo>
            </Extension>
            <!-- xsd:any within Extensions allows elements from other namespaces-->
            <typ:TextInfo xmlns:typ="http://unigrids.org/2006/04/types">
                <typ:Name>StagingInPath</typ:Name>
                <typ:Value>/user-home/in</typ:Value>
            </typ:TextInfo>
            <typ:TextInfo xmlns:typ="http://unigrids.org/2006/04/types">
                <typ:Name>StagingOutPath</typ:Name>
                <typ:Value>/user-home/out</typ:Value>
            </typ:TextInfo>
        </Extensions>
       ...
    </ExecutionEnvironment>
    ...
  <!-- xsd:any as last element within Entity allows elements from other namespaces-->
    <typ:TextInfo xmlns:typ="http://unigrids.org/2006/04/types">
        <typ:Name>StagingOutPath</typ:Name>
        <typ:Value>/user-home/out</typ:Value>
    </typ:TextInfo>
</Entities>
```

3.10 The Normative XML Schema Realization of GLUE 2.0

~~https://github.com/OGF-GLUE/XSD/blob/master/schema/GLUE2.xsd~~

To be imported into git after refining/updating:
http://redmine.ogf.org/dmsf/glue-wg?folder_id=31

4. Security Considerations

Security considerations related to the GLUE information model are discussed in section 9 of the GLUE specification [glue-2]. Additional considerations apply to the use of XML – for those, see [rfc-3470].

5. Author Information

Sergio Andreozzi
EGI.eu
Science Park 105
1098 XG Amsterdam
the Netherlands
sergio.andreozzi@egi.eu

Stephen Burke
Science and Technology Facilities Council
Rutherford Appleton Laboratory
Harwell Science and Innovation Campus
Chilton, Didcot, Oxfordshire, OX11 0QX (UK)
E-mail: s.burke@rl.ac.uk

Felix Nikolaus Ehm
CERN
Route de Meyrin 385
CH-1211 Geneva 23 (Switzerland)
E-mail: Felix.Ehm@cern.ch

Laurence Field
CERN
Route de Meyrin 385
CH-1211 Geneva 23 (Switzerland)
E-mail: Laurence.Field@cern.ch

Gerson Galang,
Australian Research Collaboration Service (ARCS)
Carlton South, Victoria (Australia)
E-mail: gerson.sapac@gmail.com

Balazs Konya,
Department of Physics, Lund University,
Professorsgatan 1, Box 118,
SE-221 00 Lund (Sweden)
E-mail: balazs.konya@hep.lu.se

Maarten Litmaath
CERN
Route de Meyrin 385
CH-1211 Geneva 23 (Switzerland)
E-mail: Maarten.Litmaath@cern.ch

Shiraz Memon
Jülich Supercomputing Centre (JSC)
Wilhelm-Johnen-Straße
52425 Jülich, Germany
Email: a.memon@fz-juelich.de

Paul Millar,
Deutsches Elektronen-Synchrotron (DESY),
Notkestraße 85,
22607 Hamburg (Germany)
E-mail: paul.millar@desy.de

John-Paul Navarro
University of Chicago/Argonne National Laboratory
Mathematics & Computer Science Division, Building 221
9700 S. Cass Avenue
Argonne, IL 60439 (USA)
E-mail: navarro@mcs.anl.gov

David Meredith
Scientific Computing Department
Science and Technology Facilities Council
Daresbury Laboratory
Warrington
Cheshire, WA4 4AD
E-mail: david.meredith@stfc.ac.uk

Adrian Taga
Warren Smith
Add address

6. Contributors & Acknowledgements

We gratefully acknowledge the contributions made to this document (in no particular order) by

7. Intellectual Property Statement

8. Disclaimer

9. Full Copyright Notice

10. References

[glue-wg] The GLUE Working Group of OGF,
https://forge.gridforum.org/sf/projects/glue-wg

[glue-uc] GLUE 2.0 Use Cases (early draft),
https://forge.gridforum.org/sf/go/doc14621

[glue-2] GLUE Specification v. 2.0, OGF GFD.147, 3 Mar 2009,
http://www.ogf.org/documents/GFD.147

[ogf-ns] Standardised Namespaces for XML infosets in OGF.
http://www.ogf.org/documents/GFD.84.pdf

[rfc-2119] Key words for use in RFCs to Indicate Requirement Levels,
http://www.ietf.org/rfc/rfc2119.txt

[rfc-3470] Guidelines for the Use of Extensible Markup Language (XML) within IETF Protocols,
http:/www.ietf.org/rfc/rfc3470.txt

[xsd-oe] XForms 1.0. Open Enumeration,
http://www.w3.org/TR/2002/WD-xforms-20020118/slice6.html #model-using-openenum

[xsd-ap] Advanced XML Schema Patterns for Databinding Version 1.0,
http://www.w3.org/TR/xmlschema-patterns-advanced/#group-Unions

Is this now required ?

Association End 1	Multiplicity	Association End 2
UserDomain	(1)←→(*)	UserDomain
AdminDomain	(1)←→(*)	Service
AdminDomain	(1)←→(*)	AdminDomain
AdminDomain	(1)←→(*)	Location
AdminDomain	(1)←→(*)	Contact
Service	(1)←→(*)	Service
Service	(1)←→(*)	Endpoint
Endpoint	(1)←→(*)	Activity
Endpoint	(1)←→(*)	AccessPolicy
Activity	(1)←→(1)	UserDomain
Activity	(1)←→(*)	Activity
ComputingEndpoint	(*)←→(*)	ComputingShare
ComputingEndpoint	(1)←→(*)	AccessPolicy
ComputingShare	(1)←→(*)	MappingPolicy
ExecutionEnvironment	(*)←→(*)	ComputingShare
ComputingActivity	(*)←→(1)	ComputingShare
ComputingActivity	(*)←→(1)	ComputingEndpoint
ComputingActivity	(*)←→(1)	ExecutionEnvironment
ComputingActivity	(1)←→(1)	UserDomain
ComputingActivity	(1)←→(*)	Activity
ComputingService	(1)←→(*)	ComputingEndpoint
ComputingService	(1)←→(*)	ComputingShare
ComputingService	(1)←→(*)	ComputingManager
ComputingService	(1)←→(*)	Service
ComputingService	(1)←→(1)	ToStorageService
ToStorageService	(1)←→(*)	StorageService
ExecutionEnvironment	(*)←→(*)	ApplicationEnvironment
ApplicationEnvironment	(1)←→(*)	ApplicationHandle
ComputingManager	(1)←→(*)	ExecutionEnvironment
ComputingManager	(1)←→(*)	ApplicationEnvironment
ComputingManager	(1)←→(*)	Benchmark
ExecutionEnvironment	(1)←→(*)	Benchmark
StorageService	(1)←→(*)	StorageEndpoint
StorageService	(1)←→(*)	StorageShare
StorageService	(1)←→(*)	StorageManager
StorageService	(1)←→(*)	StorageAccessProtocol
StorageService	(1)←→(*)	StorageServiceCapacity
StorageService	(1)←→(*)	ToComputingService
StorageAccessProtocol	(1)←→(*)	ToComputingService
StorageManager	(1)←→(*)	DataStore
StorageEndpoint	(*)←→(*)	StorageShare
StorageShare	(*)←→(*)	DataStore
StorageShare	(1)←→(*)	StorageShareCapacity
StorageService	(1)←→(*)	Service