



IST-2001-32133

GridLab - A Grid Application Toolkit and Testbed

## Triana Metadata Specification

---

Author(s):	Ian Taylor, Shalil Majithia, Matthew Shields, Ian Wang, Kelly Davis
Document Filename:	GridLab-3-D3_1-0001-MetaDataSpec
Work package:	WP3 Work-Flow Application Toolkit (TGAT)
Partner(s):	University of Wales, Cardiff
Lead Partner:	University of Wales, Cardiff
Config ID:	GridLab-3-D3_1-0001-1-0-DRAFT-A
Document classification:	INTERNAL

---

**Abstract:** This document specifies Triana Metadata. This document specifies Triana unit metadata, Triana algorithm metadata, Triana type metadata, Triana optimization metadata, and Triana monitoring metadata.



## Contents

<b>1</b>	<b>Status of this Document</b>	<b>3</b>
<b>2</b>	<b>Introduction</b>	<b>3</b>
<b>3</b>	<b>Triana Unit Metadata</b>	<b>5</b>
<b>4</b>	<b>Web Services Description Language ( A Quick Tour )</b>	<b>7</b>
4.1	WSDL Document Structure . . . . .	7
4.2	Types . . . . .	8
4.3	Messages . . . . .	9
4.3.1	Message Parts . . . . .	10
4.3.2	Abstract vs. Concrete Messages . . . . .	12
4.4	Port Types . . . . .	12
4.4.1	One-way Operation . . . . .	13
4.4.2	Request-response Operation . . . . .	13
4.5	Solicit-response Operation . . . . .	14
4.5.1	Notification Operation . . . . .	14
4.5.2	Names of Elements within an Operation . . . . .	15
4.5.3	Parameter Order within an Operation . . . . .	15
4.6	Bindings . . . . .	16
4.7	Ports . . . . .	16
4.8	Services . . . . .	17
<b>5</b>	<b>WSDL ( General Triana Extensions )</b>	<b>19</b>
5.1	Triana Types . . . . .	19
5.1.1	Triana XML Schema . . . . .	19
5.1.2	Triana Messages . . . . .	20
5.1.3	Triana Port Types . . . . .	21
5.2	Triana Binding Extensions . . . . .	22
<b>6</b>	<b>Optimization Metadata</b>	<b>25</b>
<b>7</b>	<b>Monitoring Metadata</b>	<b>27</b>
<b>8</b>	<b>Triana Algorithm Metadata</b>	<b>28</b>
<b>9</b>	<b>Web Services Flow Language (A Quick Tour )</b>	<b>29</b>
9.1	Flow Models . . . . .	29
9.2	Global Models . . . . .	29
9.3	Recursive Composition . . . . .	29
9.4	A World-Wind Tour of WSFL . . . . .	30
<b>10</b>	<b>Web Services Flow Language (Triana Specifics)</b>	<b>34</b>
<b>A</b>	<b>Appendix: WSDL Schema</b>	<b>36</b>



---

<b>B Appendix: WSDL Triana Specifics</b>	<b>42</b>
B.1 Triana Types . . . . .	42
B.2 Triana Messages . . . . .	43
B.3 Triana Port Types . . . . .	44
<b>C Appendix: WSFL Schema</b>	<b>45</b>
<b>D Appendix: WSFL Triana Specifics</b>	<b>53</b>
D.1 Triana Service Provider Types . . . . .	53

## 1 Status of this Document

This document is a draft version. As a working draft, this specification may be updated, replaced, or made obsolete at any time. It is distributed for discussion purposes only, and should not be used as a reference. Readers are encouraged to send comments to the Triana mailing list.

## 2 Introduction

Metadata is defined to be data about data. It is data which refers to other data and provides context to such data. So, for example, a dictionary entry can be thought of as data and metadata. The word to be defined, “dog” for example, is the data. The definition, “A domesticated canid” for example, is the metadata which refers to the data and provides it context. Within the context of Triana there are various data elements for which metadata is required.

Before describing the uses Triana has for metadata, let us quickly review the various elements which play a major role in Triana. As described in detail in the Triana requirements document, a *Triana algorithm* is constructed by an *application user* by “connecting together” various *Triana units*. A *Triana unit* is a collection of code which takes some input and produces some output. So, a *Triana algorithm* is simply the specification of the “flow” of data between various pre-existing *Triana units*. In addition, Triana allows for so-called “recursive composition.” In other words, Triana allows for an *application user* to create a *Triana algorithm* then take this *Triana algorithm* and “package” it as a *Triana unit*. This new *Triana unit* can then be used just as any other *Triana unit*. In particular, it can be used in another *Triana algorithm*. This process of “packaging” a *Triana algorithm* as a *Triana unit* is called *recursive composition*.

Now, after reviewing the various elements which play a major role in Triana, we are in a better position to examine several simple Triana use cases which require Triana employ metadata:

- A LIGO scientist wishes to reproduce the Triana calculations of a colleague which found the gravitational wave signature of a pair of inspiraling black holes. The scientist who originally ran the calculations needs to have some means of describing the various *Triana units* in the calculation, the manner in which the *Triana units* were connected for the calculation, the manner in which data flows between the various *Triana units* in the calculation, the *Triana algorithms* which are recursively composed for the calculation, and so on.

Metadata about *Triana units*, metadata about *Triana unit* “connections,” metadata about Triana data “flows,” metadata about *Triana algorithms*, metadata about Triana *recursive composition*, and so on, will be required for the two colleagues to exchange information about the calculation. Thus, Triana requires, at the very least, metadata about such elements.

- A Triana *application user* wishes to automatically “obtain” remote *Triana units* which fit a given description. This implies that each *Triana unit* is described in a such manner so as to allow Triana to search for such *Triana units*. This implies the requirement that the metadata which describes a *Triana unit* be such that it contains information which is searchable by Triana.
- A Triana *application user* wishes to “obtain” remote *Triana units* which fit a given description. This implies that each *Triana unit* is described in a such manner so as to allow

an *application user* to search on such *Triana units*. This implies the requirement that the metadata which describes a *Triana unit* be such that it contains information which is searchable by an *application user*.

- It will be difficult to reuse other people's *Triana units* and *Triana algorithms* without metadata to provide the context for these objects. A scientist considering reusing someone else's *Triana unit* or *Triana algorithm* will need to know things such as: what the it does, the allowed input types, the allowed output types, the "parameters," the author, where the help files are, and so on.
- A *application user* wishes to use a "living document." The "living document" concept illustrates the use of Metadata within Triana. A living document is a web page that contains a graphical representation of a Triana network which executes when a user starts it using the web interface. Such a living document shows the complete reconstruction of a *Triana algorithm*. To do this, we need to standardize metadata about *Triana units*, standardize metadata about how *Triana units* are "connected," standardize metadata about how Triana data "flows" between units, standardize the Triana data types, standardize metadata about *Triana algorithms*, and standardize metadata about *recursive composition* to name the most obvious standardization requirements. There will almost surely be others.
- As the number of models and components grows, metadata will provide the only scalable method for locating particular models and components. Experience in many fields shows that as a field grows, powerful search techniques are needed to enable researchers to find relevant resources. These search techniques require structured metadata.
- Metadata will also be essential for monitoring the execution of the *Triana algorithm* and extracting "Quality of Service" (QoS) parameters.

Metadata in Triana can be used in many different ways, such as:

- To support searches of a *Triana unit* repository
- To enable automatic discovery of remote *Triana units*
- To enable the user to reconstruct a *Triana algorithm*
- To enable the user to ascertain the status of a *Triana algorithm*

Whatever the structure the metadata takes it should be flexible and extensible because it is almost certain that we have not thought of all possible uses of Triana Metadata.

### 3 Triana Unit Metadata

The typical use cases for *Triana unit* metadata include:

1. Search for a *Triana unit* given a “parameter,” e.g. name, description, input/output types etc. e.g. a user might know what they want to do but they are not sure of the “name” of the *Triana unit* that can perform this type of operation. If we associated a “semantic description” with each unit, then this type of search can be realized.
2. To point out incompatible Triana data types when connecting *Triana units*.
3. To allow for flexible mapping of input types to output types when connecting *Triana units*.
4. To enable automatic translation from a “task graph” created in another dataflow or workflow environment into one that would be “equivalent” within Triana.
5. To allow the user to refer to a *Triana unit* by a pseudo name, not necessarily the class name.
6. To allow the user to refer to a *recursively composed Triana algorithm* as a *Triana unit*.
7. To enable the users to use the most recent version of the *Triana unit*. Versioning within units is important as it enables programmers to get the latest unit code and users need to know which version a unit is so that their results can be reconstructed accurately. There is no guarantee that newer versions of units are backwardly compatible (we would hope that they are though) so along with the main data flows, we need to reconstruct the algorithm of units along with their correct versions.

The use cases motivate a number of requirements for *Triana unit* metadata:

1. Name of unit
2. Number of input/output nodes
3. Description
4. Help files name and location
5. Input/output types
6. Parameters name/value

To motivate the format of the metadata for Triana let us examine once more a *Triana unit*. A *Triana unit* can be *local*, executing on the same computer as the Triana instance, or *remote*, executing on a computer which is not the same as the computer on which Triana is executing. ( In addition, a *local Triana unit* can be executing in such a manner so as to look like a *remote Triana unit*, to the local instance of Triana. )

As of this current moment in time, GridLab is planning to implement *remote Triana units* using WebServices. *Local Triana units* are to be implemented using various technologies. As of this current moment in time, there is a standardization effort WSDL; spearheaded by Ariba, IBM, and Microsoft as a submission to the W3C; to to define the metadata required for Web Services. To not “reinvent the wheel” we suggest using the Ariba, IBM, and Microsoft metadata standard WSDL to describe *remote Triana units*. Furthermore, to not construct an overly complicated

architecture for Triana dealing with metadata for *local* and *remote Triana units* through two differing specifications, we suggest using WSDL to describe *local Triana units* also. Now, let us examine in more detail WSDL through a quick quote of the WSDL specification [1].

## 4 Web Services Description Language ( A Quick Tour )

This section describes the core elements of the WSDL language.

### 4.1 WSDL Document Structure

A WSDL document is simply a **set of definitions**. There is a **definitions** element at the root, and definitions inside. The grammar is as follows:

```
<wsdl:definitions name="nmtoken"? targetNamespace="uri"?>
  <import namespace="uri" location="uri"/>*
  <wsdl:documentation .... /> ?
  <wsdl:types> ?
    <wsdl:documentation .... />?
    <xsd:schema .... />*
    <-- extensibility element --> *
  </wsdl:types>
  <wsdl:message name="nmtoken"> *
    <wsdl:documentation .... />?
    <part name="nmtoken" element="qname"? type="qname"?/> *
  </wsdl:message>
  <wsdl:portType name="nmtoken">*
    <wsdl:documentation .... />?
    <wsdl:operation name="nmtoken">*
      <wsdl:documentation .... /> ?
      <wsdl:input name="nmtoken"? message="qname"?>?
        <wsdl:documentation .... /> ?
      </wsdl:input>
      <wsdl:output name="nmtoken"? message="qname"?>?
        <wsdl:documentation .... /> ?
      </wsdl:output>
      <wsdl:fault name="nmtoken" message="qname"> *
        <wsdl:documentation .... /> ?
      </wsdl:fault>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="nmtoken" type="qname">*
    <wsdl:documentation .... />?
    <-- extensibility element --> *
    <wsdl:operation name="nmtoken">*
      <wsdl:documentation .... /> ?
      <-- extensibility element --> *
      <wsdl:input> ?
        <wsdl:documentation .... /> ?
        <-- extensibility element -->
      </wsdl:input>
```

```
<wsdl:output> ?
  <wsdl:documentation .... /> ?
  <-- extensibility element --> *
</wsdl:output>
<wsdl:fault name="nmtoken"> *
  <wsdl:documentation .... /> ?
  <-- extensibility element --> *
</wsdl:fault>
</wsdl:operation>
</wsdl:binding>
<wsdl:service name="nmtoken"> *
  <wsdl:documentation .... />?
  <wsdl:port name="nmtoken" binding="qname"> *
    <wsdl:documentation .... /> ?
    <-- extensibility element -->
  </wsdl:port>
  <-- extensibility element -->
</wsdl:service>
<-- extensibility element --> *
</wsdl:definitions>
```

Services are defined using six major elements:

- **types**, which provides data type definitions used to describe the messages exchanged.
- **message**, which represents an abstract definition of the data being transmitted. A message consists of logical parts, each of which is associated with a definition within some type system.
- **portType**, which is a set of abstract operations. Each operation refers to an input message and output messages.
- **binding**, which specifies concrete protocol and data format specifications for the operations and messages defined by a particular portType.
- **port**, which specifies an address for a binding, thus defining a single communication endpoint.
- **service**, which is used to aggregate a set of related ports.

These elements will now be described in detail.

## 4.2 Types

The types element encloses data type definitions that are relevant for the exchanged messages. For maximum interoperability and platform neutrality, WSDL prefers the use of XSD as the canonical type system, and treats it as the intrinsic type system.

```
<definitions .... >
  <types>
    <xsd:schema .... />*
  </types>
</definitions>
```

The XSD type system can be used to define the types in a message regardless of whether or not the resulting wire format is actually XML, or whether the resulting XSD schema validates the particular wire format. This is especially interesting if there will be multiple bindings for the same message, or if there is only one binding but that binding type does not already have a type system in widespread use. In these cases, the recommended approach for encoding abstract types using XSD is as follows:

- Use element form (not attribute).
- Don't include attributes or elements that are peculiar to the wire encoding (e.g. have nothing to do with the abstract content of the message). Some examples are soap:root, soap:encodingStyle, xmi:id, xmi:name.
- Array types should extend the Array type defined in the SOAP v1.1 encoding schema (<http://schemas.xmlsoap.org/soap/encoding/>) (regardless of whether the resulting form actually uses the encoding specified in Section 5 of the SOAP v1.1 document). Use the name ArrayOfXXX for array types (where XXX is the type of the items in the array). The type of the items in the array and the array dimensions are specified by using a default value for the soapenc:arrayType attribute. At the time of this writing, the XSD specification does not have a mechanism for specifying the default value of an attribute which contains a QName value. To overcome this limitation, WSDL introduces the arrayType attribute (from namespace <http://schemas.xmlsoap.org/wsdl/>) which has the semantic of providing the default value. If XSD is revised to support this functionality, the revised mechanism SHOULD be used in favor of the arrayType attribute defined by WSDL.
- Use the xsd:anyType type to represent a field/parameter which can have any type.

However, since it is unreasonable to expect a single type system grammar can be used to describe all abstract types present and future, WSDL allows type systems to be added via extensibility elements. An extensibility element may appear under the types element to identify the type definition system being used and to provide an XML container element for the type definitions. The role of this element can be compared to that of the schema element of the XML Schema language.

```
<definitions .... >
  <types>
    <!-- type-system extensibility element --> *
  </types>
</definitions>
```

### 4.3 Messages

Messages consist of one or more logical **parts**. Each part is associated with a type from some type system using a message-typing attribute. The set of message-typing attributes is extensible. WSDL defines several such message-typing attributes for use with XSD:

- **element**. Refers to an XSD element using a QName.
- **type**. Refers to an XSD simpleType or complexType using a QName.

Other message-typing attributes may be defined as long as they use a namespace different from that of WSDL. Binding extensibility elements may also use message-typing attributes.

The syntax for defining a message is as follows.

```
<definitions .... >
  <message name="nmtoken" > *
    <part name="nmtoken" element="qname"? type="qname"?/> *
  </message>
</definitions>
```

The message **name** attribute provides a unique name among all messages defined within the enclosing WSDL document.

The part **name** attribute provides a unique name among all the parts of the enclosing message.

#### 4.3.1 Message Parts

Parts are a flexible mechanism for describing the logical abstract content of a message. A binding may reference the name of a part in order to specify binding-specific information about the part. For example, if defining a message for use with RPC, a part may represent a parameter in the message. However, the bindings must be inspected in order to determine the actual meaning of the part.

Multiple part elements are used if the message has multiple logical units. For example, the following message consists of a Purchase Order and an Invoice.

```
<definitions .... >
  <types>
    <schema .... >
      <element name="PO" type="tns:POType"/>
      <complexType name="POType">
        <all>
          <element name="id" type="string"/>
          <element name="name" type="string"/>
          <element name="items">
            <complexType>
              <all>
                <element name="item" type="tns:Item" minOccurs="0"
                  maxOccurs="unbounded"/>
              </all>
            </complexType>
          </element>
        </all>
      </complexType>
    </types>
  </definitions>
```

```
<complexType name="Item">
  <all>
    <element name="quantity" type="int"/>
    <element name="product" type="string"/>
  </all>
</complexType>
<element name="Invoice" type="tns:InvoiceType"/>
<complexType name="InvoiceType">
  <all>
    <element name="id" type="string"/>
  </all>
</complexType>
</schema>
</types>
<message name="PO">
  <part name="po" element="tns:PO"/>
  <part name="invoice" element="tns:Invoice"/>
</message>
</definitions>
```

However, if the message contents are sufficiently complex, then an alternative syntax may be used to specify the composite structure of the message using the type system directly. In this usage, only one part may be specified. In the following example, the body is either a purchase order, or a set of invoices.

```
<definitions .... >
  <types>
    <schema .... >
      <complexType name="POType">
        <all>
          <element name="id" type="string"/>
          <element name="name" type="string"/>
          <element name="items">
            <complexType>
              <all>
                <element name="item" type="tns:Item" minOccurs="0" maxOccurs=
              </all>
            </complexType>
          </element>
        </all>
      </complexType>
      <complexType name="Item">
        <all>
          <element name="quantity" type="int"/>
          <element name="product" type="string"/>
        </all>
      </complexType>
```

```
<complexType name="InvoiceType">
  <all>
    <element name="id" type="string"/>
  </all>
</complexType>
<complexType name="Composite">
  <choice>
    <element name="PO" minOccurs="1" maxOccurs="1" type="tns:POType"/>
    <element name="Invoice" minOccurs="0" maxOccurs="unbounded" type="tns:Inv
  </choice>
</complexType>
</schema>
</types>
<message name="PO">
  <part name="composite" type="tns:Composite"/>
</message>
</definitions>
```

#### 4.3.2 Abstract vs. Concrete Messages

Message definitions are always considered to be an abstract definition of the message content. A message binding describes how the abstract content is mapped into a concrete format. However, in some cases, the abstract definition may match the concrete representation very closely or exactly for one or more bindings, so those binding(s) will supply little or no mapping information. However, another binding of the same message definition may require extensive mapping information. For this reason, it is not until the binding is inspected that one can determine "how abstract" the message really is.

#### 4.4 Port Types

A port type is a named set of abstract operations and the abstract messages involved.

```
<wsdl:definitions .... >
  <wsdl:portType name="nmtoken">
    <wsdl:operation name="nmtoken" .... /> *
  </wsdl:portType>
</wsdl:definitions>
```

The port type **name** attribute provides a unique name among all port types defined within in the enclosing WSDL document.

An operation is named via the **name** attribute.

WSDL has four transmission primitives that an endpoint can support:

- **One-way.** The endpoint receives a message.
- **Request-response.** The endpoint receives a message, and sends a correlated message.
- **Solicit-response.** The endpoint sends a message, and receives a correlated message.
- **Notification.** The endpoint sends a message.

WSDL refers to these primitives as operations. Although request/response or solicit/response can be modeled abstractly using two one-way messages, it is useful to model these as primitive operation types because:

- They are very common.
- The sequence can be correlated without having to introduce more complex flow information.
- Some endpoints can only receive messages if they are the result of a synchronous request response.
- A simple flow can algorithmically be derived from these primitives at the point when flow definition is desired.

Although request/response or solicit/response are logically correlated in the WSDL document, a given binding describes the concrete correlation information. For example, the request and response messages may be exchanged as part of one or two actual network communications.

Although the base WSDL structure supports bindings for these four transmission primitives, WSDL only defines bindings for the One-way and Request-response primitives. It is expected that specifications that define the protocols for Solicit-response or Notification would also include WSDL binding extensions that allow use of these primitives.

Operations refer to the messages involved using the message attribute of type QName. This attribute follows the rules defined by WSDL for linking.

#### 4.4.1 One-way Operation

The grammar for a one-way operation is:

```
<wsdl:definitions .... >
  <wsdl:portType .... > *
    <wsdl:operation name="nmtoken">
      <wsdl:input name="nmtoken"? message="qname"/>
    </wsdl:operation>
  </wsdl:portType >
</wsdl:definitions>
```

The **input** element specifies the abstract message format for the one-way operation.

#### 4.4.2 Request-response Operation

The grammar for a request-response operation is:

```
<wsdl:definitions .... >
  <wsdl:portType .... > *
    <wsdl:operation name="nmtoken" parameterOrder="nmtokens">
      <wsdl:input name="nmtoken"? message="qname"/>
      <wsdl:output name="nmtoken"? message="qname"/>
      <wsdl:fault name="nmtoken" message="qname"/>*
    </wsdl:operation>
  </wsdl:portType >
</wsdl:definitions>
```

The input and output elements specify the abstract message format for the request and response, respectively. The optional fault elements specify the abstract message format for any error messages that may be output as the result of the operation (beyond those specific to the protocol).

Note that a request-response operation is an abstract notion; a particular binding must be consulted to determine how the messages are actually sent: within a single communication (such as a HTTP request/response), or as two independent communications (such as two HTTP requests).

## 4.5 Solicit-response Operation

The grammar for a solicit-response operation is:

```
<wsdl:definitions .... >
  <wsdl:portType .... > *
    <wsdl:operation name="nmtoken" parameterOrder="nmtokens">
      <wsdl:output name="nmtoken"? message="qname"/>
      <wsdl:input name="nmtoken"? message="qname"/>
      <wsdl:fault name="nmtoken" message="qname"/>*
    </wsdl:operation>
  </wsdl:portType >
</wsdl:definitions>
```

The output and input elements specify the abstract message format for the solicited request and response, respectively. The optional fault elements specify the abstract message format for any error messages that may be output as the result of the operation (beyond those specific to the protocol).

Note that a solicit-response operation is an abstract notion; a particular binding must be consulted to determine how the messages are actually sent: within a single communication (such as a HTTP request/response), or as two independent communications (such as two HTTP requests).

### 4.5.1 Notification Operation

The grammar for a notification operation is:

```
<wsdl:definitions .... >
  <wsdl:portType .... > *
    <wsdl:operation name="nmtoken">
      <wsdl:output name="nmtoken"? message="qname"/>
    </wsdl:operation>
  </wsdl:portType >
</wsdl:definitions>
```

The **output** element specifies the abstract message format for the notification operation.

#### 4.5.2 Names of Elements within an Operation

The **name** attribute of the input and output elements provides a unique name among all input and output elements within the enclosing port type.

In order to avoid having to name each input and output element within an operation, WSDL provides some default values based on the operation name. If the name attribute is not specified on a one-way or notification message, it defaults to the name of the operation. If the name attribute is not specified on the input or output messages of a request-response or solicit-response operation, the name defaults to the name of the operation with "Request"/"Solicit" or "Response" appended, respectively.

Each fault element must be named to allow a binding to specify the concrete format of the fault message. The name of the fault element is unique within the set of faults defined for the operation.

#### 4.5.3 Parameter Order within an Operation

Operations do not specify whether they are to be used with RPC-like bindings or not. However, when using an operation with an RPC-binding, it is useful to be able to capture the original RPC function signature. For this reason, a request-response or solicit-response operation may specify a list of parameter names via the **parameterOrder** attribute (of type nmtokens). The value of the attribute is a list of message part names separated by a single space. The value of the parameterOrder attribute must follow the following rules:

- The part name order reflects the order of the parameters in the RPC signature
- The **return** value part is not present in the list
- If a part name appears in both the input and output message, it is an **in/out** parameter
- If a part name appears in only the input message, it is an **in** parameter
- If a part name appears in only the output message, it is an **out** parameter

Note that this information serves as a "hint" and may safely be ignored by those not concerned with RPC signatures. Also, it is not required to be present, even if the operation is to be used with an RPC-like binding.

## 4.6 Bindings

A binding defines message format and protocol details for operations and messages defined by a particular portType. There may be any number of bindings for a given portType. The grammar for a binding is as follows:

```
<wsdl:definitions .... >
  <wsdl:binding name="nmtoken" type="qname"> *
    <!-- extensibility element (1) --> *
    <wsdl:operation name="nmtoken"> *
      <!-- extensibility element (2) --> *
      <wsdl:input name="nmtoken"? > ?
        <!-- extensibility element (3) -->
      </wsdl:input>
      <wsdl:output name="nmtoken"? > ?
        <!-- extensibility element (4) --> *
      </wsdl:output>
      <wsdl:fault name="nmtoken"> *
        <!-- extensibility element (5) --> *
      </wsdl:fault>
    </wsdl:operation>
  </wsdl:binding>
</wsdl:definitions>
```

The **name** attribute provides a unique name among all bindings defined within in the enclosing WSDL document.

A binding references the portType that it binds using the **type** attribute. This QName value follows the linking rules defined by WSDL.

Binding extensibility elements are used to specify the concrete grammar for the input (3), output (4), and fault messages (5). Per-operation binding information (2) as well as per-binding information (1) may also be specified.

An operation element within a binding specifies binding information for the operation with the same name within the binding's portType. Since operation names are not required to be unique (for example, in the case of overloading of method names), the name attribute in the operation binding element might not be enough to uniquely identify an operation. In that case, the correct operation should be identified by providing the name attributes of the corresponding wsdl:input and wsdl:output elements.

A binding must specify exactly one protocol.

A binding must not specify address information.

## 4.7 Ports

A port defines an individual endpoint by specifying a single address for a binding.

```
<wsdl:definitions .... >
  <wsdl:service .... > *
    <wsdl:port name="nmtoken" binding="qname"> *
      <!-- extensibility element (1) -->
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

The **name** attribute provides a unique name among all ports defined within in the enclosing WSDL document.

The **binding** attribute (of type QName) refers to the binding using the linking rules defined by WSDL.

Binding extensibility elements (1) are used to specify the address information for the port.

A port must not specify more than one address.

A port must not specify any binding information other than address information.

## 4.8 Services

A service groups a set of related ports together:

```
<wsdl:definitions .... >
  <wsdl:service name="nmtoken"> *
    <wsdl:port .... /*
  </wsdl:service>
</wsdl:definitions>
```

The **name** attribute provides a unique name among all services defined within in the enclosing WSDL document.

Ports within a service have the following relationship:

- None of the ports communicate with each other (e.g. the output of one port is not the input of another).
- If a service has several ports that share a port type, but employ different bindings or addresses, the ports are alternatives. Each port provides semantically equivalent behavior (within the transport and message format limitations imposed by each binding). This allows a consumer of a WSDL document to choose particular port(s) to communicate with based on some criteria (protocol, distance, etc.).
- By examining it's ports, we can determine a service's port types. This allows a consumer of a WSDL document to determine if it wishes to communicate to a particular service based whether or not it supports several port types. This is useful if there is some implied relationship between the operations of the port types, and that the entire set of port types must be present in order to accomplish a particular task.

For more detail on the full WSDL syntax the interested reader can refer to the WSDL specification of the WSDL schema which is reproduced in the appendix as well as the WSDL specification [1].

## 5 WSDL ( General Triana Extensions )

A Triana unit, to work with Triana and in order to meet the metadata requirements outlined above, must implement the port types defined below. Each of these port types provides a functionality that was outlined above. In addition, beyond these requirements, each Triana unit must implement various other ports types which are described in the optimization metadata and monitoring metadata sections and local Triana units must bind using a Java WSDL binding extension.

### 5.1 Triana Types

Triana defines various types to be used in a WSDL description of a *Triana unit*. Each of these various types' existence can be derived directly from the requirements imposed by the various use cases outlined above.

#### 5.1.1 Triana XML Schema

The various Triana types are based upon a small set of types defined in a Triana specific XML schema. This schema, located at <http://www.gridlab.org/wp3/trianaunit.xsd>, has the following form:

```
<?xml version="1.0"?>
<schema targetNamespace="http://www.gridlab.org/wp3/schemas"
        xmlns="http://www.w3.org/2000/10/XMLSchema"
        xmlns:tns="http://www.gridlab.org/wp3/schemas">
  <simpleType name="guiweights">
    <restriction base="string">
      <enumeration value="LightWeight"/>
      <enumeration value="MiddleWeight"/>
      <enumeration value="HeavyWeight"/>
    </restriction>
  </simpleType>
  <element name="TrianaUnitName" type="string"/>
  <element name="TrianaGUIDescription">
    <complexType>
      <sequence>
        <element name="TrianaGUILanguage" type="language"/>
        <element name="TrianaGUIWeight" type="tns:guiweights"/>
      </sequence>
    </complexType>
  </element>
  <element name="TrianaIconHTMLGUI">
    <complexType>
      <sequence>
        <any namespace="http://www.w3.org/1999/xhtml" minOccurs="1"
            maxOccurs="unbounded" processContents="skip"/>
      </sequence>
    </complexType>
  </element>
</schema>
```

```
<element name="TrianaInformationHTMLGUI">
  <complexType>
    <sequence>
      <any namespace="http://www.w3.org/1999/xhtml" minOccurs="1"
          maxOccurs="unbounded" processContents="skip"/>
    </sequence>
  </complexType>
</element>
<element name="TrianaConfigurationHTMLGUI">
  <complexType>
    <sequence>
      <any namespace="http://www.w3.org/1999/xhtml" minOccurs="1"
          maxOccurs="unbounded" processContents="skip"/>
    </sequence>
  </complexType>
</element>
</schema>
```

The semantic interpretation of each of these type is rather obvious. Semantically, a TrianaUnitName is the name of a Triana unit. A TrianaGUIDescription consists describes what type of GUI to display. The TrianaGUIDescription consists of two parts a TrianaGUILanguage which describes the GUI language and a TrianaGUIWeight which describes the "weight" of the GUI (LightWeight for a small device such as a PalmPilot, HeavyWeight for something like a web browser, and MiddleWeight for somewhere inbetween). TrianaIconHTMLGUI is a type which contains HTML to render the icon of the Triana unit as it is to appear in a GUI. TrianaInformationHTMLGUI is a type which contains HTML to render the informational pane for the Triana unit which describes the unit and gives help for the unit. Finally, TrianaConfigurationHTMLGUI is a type which contains HTML to render the configuration pane used to configure the Triana unit.

### 5.1.2 Triana Messages

Built on the various XSD types defined in the previous section are various WSDL messages. Again, the existence of each of these messages and port types is dictated by the requirements on the Triana metadata which were previously detailed.

The various WSDL messages required of a Triana unit are given by the following WSDL document which is located at <http://www.gridlab.org/wp3/trianamessages.wsdl>:

```
<?xml version="1.0"?>
<definitions name="TrianaMessages"
  targetNamespace="http://www.gridlab.org/wp3/messages"
  xmlns:tns="http://www.gridlab.org/wp3/messages"
  xmlns:xsd1="www.gridlab.org/wp3/schemas"
  xmlns="http://schemas.xmlsoap.org/wsdl/">

  <import namespace="http://www.gridlab.org/wp3/schemas"
    location="http://www.gridlab.org/wp3/trianaunit.xsd"/>
```

```
<message name="TrianaUnitNameType">
  <part name="body" element="xsd1:TrianaUnitName"/>
</message>

<message name="TrianaGUIDescriptionType">
  <part name="body" element="xsd1:TrianaGUIDescription"/>
</message>

<message name="TrianaIconHTMLGUIType">
  <part name="body" element="xsd1:TrianaIconHTMLGUI"/>
</message>

<message name="TrianaInformationHTMLGUIType">
  <part name="body" element="xsd1:TrianaInformationHTMLGUI"/>
</message>

<message name="TrianaConfigurationHTMLGUIType">
  <part name="body" element="xsd1:TrianaConfigurationHTMLGUI"/>
</message>
</definitions>
```

The semantic interpretation of each of the various WSDL messages defined above is exactly the same as detailed in the XML schema as there is a one-to-one onto mapping between the two specifications.

### 5.1.3 Triana Port Types

Built upon the Triana messages defined in the previous section are various Triana port types. Each of these port types specifies a specific functionality that all Triana units must support. Each Triana unit, must be a WSDL service which implements these WSDL port types.

The various WSDL port types required of a Triana unit are given by the following WSDL document which is located at <http://www.gridlab.org/wp3/trianaporttypes.wsdl>:

```
<?xml version="1.0"?>
<definitions name="TrianaPortTypes"
  targetNamespace="http://www.gridlab.org/wp3/porttypes"
  xmlns:tns="http://www.gridlab.org/wp3/porttypes"
  xmlns:xsd1="http://www.gridlab.org/wp3/messages"
  xmlns="http://schemas.xmlsoap.org/wsdl/">

  <import namespace="http://www.gridlab.org/wp3/messages"
    location="http://www.gridlab.org/wp3/trianamessages.wsdl"/>
```

```
<portType name="TrianaUnitNamePortType">
  <operation name="GetTrianaUnitName">
    <output message="xsd1:TrianaUnitNameType"/>
  </operation>
</portType>

<portType name="TrianaIconHTMLGUIPortType">
  <operation name="GetTrianaIconHTMLGUI">
    <input message="xsd1:TrianaGUIDescriptionType"/>
    <output message="xsd1:TrianaIconHTMLGUIType"/>
  </operation>
</portType>

<portType name="TrianaInformationHTMLGUIPortType">
  <operation name="GetTrianaInformationHTMLGUI">
    <input message="xsd1:TrianaGUIDescriptionType"/>
    <output message="xsd1:TrianaInformationHTMLGUIType"/>
  </operation>
</portType>

<portType name="TrianaConfigurationHTMLGUIPortType">
  <operation name="GetTrianaConfigurationHTMLGUI">
    <input message="xsd1:TrianaGUIDescriptionType"/>
    <output message="xsd1:TrianaConfigurationHTMLGUIType"/>
  </operation>
</portType>
</definitions>
```

Semantically the `TrianaUnitNamePort` port type allows one to obtain the name of the Triana unit. The port type `TrianaIconHTMLGUIPortType` allows one to obtain the HTML icon for the Triana unit localized with the `TrianaGUIDescriptionType` to the processor and language specified. The port type `TrianaInformationHTMLGUIPortType` allows one to obtain the HTML informational GUI for the Triana unit localized with the `TrianaGUIDescriptionType` to the processor and language specified. The port type `TrianaConfigurationHTMLGUIPortType` allows one to obtain the HTML configuration GUI for the Triana unit localized with the `TrianaGUIDescriptionType` to the processor and language specified.

## 5.2 Triana Binding Extensions

The IBM has provided a standard binding extension for Java through their WSIF [3]. This allows one to bind a given port type's implementation to a method on an instance of a Java class. There are various examples of this which can be found in the IBM WSIF download [3]. Also, this binding extension is to become an extension to Java through JSR 110 [4].

Each use of this binding extension, as can be expected, depends on the name of the Java class and the names of its various methods. Hence, as such Triana classes do not as of yet exist, we can not explicitly write the WSDL document which creates the binding between a port type and a Java class. However, we can give an example of such from the WSFL specification [2] to give a flavor of such WSDL documents and how they will bind Java to a port type:

```
<definitions name="CreditCardVerifier"
  targetNamespace="http://example.com/creditCardVerification.wsdl"
  xmlns:java="http://schemas.xmlsoap.org/wsdl/java/">
  <types>
    <schema>
      <complexType name="creditCard">
        <element name="number" type="integer"/>
        <element name="name" type="string"/>
        <element name="expirationDate" type="gYearMonth"/>
      </complexType>
    </schema>
    <schema>
      <simpleType name="cardStatus">
        <restriction base="string">
          <enumeration value="STATUS_OK"/>
          <enumeration value="STATUS_INVALID_NUMBER"/>
          <enumeration value="STATUS_INVALID_NAME"/>
          <enumeration value="STATUS_EXPIRED"/>
        </restriction>
      </simpleType>
    </schema>
  </types>
  <message name="CardInformation">
    <part name="cardinfo" type="tns:creditCard"/>
  </message>
  <message name="CardStatus">
    <part name="status" type="tns:cardStatus"/>
  </message>
  <port type name="CreditCardVerificationPortType">
    <operation name="verifyCard">
      <input message="CardInformation"/>
      <output message="CardStatus"/>
    </operation>
  </port type>
  <binding name="LocalCardVerifier"
    type="tns:CreditCardVerificationPortType">
    <operation name="verifyCard">
      <java:operation javamethod="verifyCreditCard"/>
      <input>
        <java:typemapping name="tns:creditCard"
          class="com.example.verifier.CreditCard"
          serializer="com.example.verifier.CardSerializer"
          deserializer="com.example.verifier.CardSerializer"/>
      </input>
      <output>
        <java:typemapping name="tns:cardStatus"
          class="com.example.verifier.Status"
          serializer="com.example.verifier.StatusSerializer"
          deserializer="com.example.verifier.StatusSerializer"/>
      </output>
    </operation>
  </binding>
</definitions>
```

```
    </operation>
  </binding>
  <service name="CreditCardVerificationService">
    <port name="CreditCardVerificationPort"
      binding="LocalCardVerifier">
      <java:provider class="com.example.verifier.CardVerifier"/>
    </port>
  </service>
</definitions>
```

## 6 Optimization Metadata

Envisaged use cases:

- Users should be able to use the Grid transparently within Triana i.e. a user should need only to provide a *Triana algorithm* along with other user constraints, such as a maximum time for completion say. Mechanisms plugged into Triana should then determine from this information the best way that the *Triana algorithm* can be distributed onto the Grid.
- Users should have the option to specify the distribution of tasks i.e. the user can specify a *Triana unit* to run at particular hosts.
- Users should be able to specify the wire protocol each *remote* or *local Triana unit* uses to transmit information.
- Users should be able to specify which of a number of *Triana units* which implement the same WSDL port types is used based upon the start and stop time each *Triana unit* insures.
- Users should be able to specify which of a number of

These use cases imply the following additional requirements on the Triana unit metadata:

- Each Triana unit should register itself in a global registry, likely this will be some form of UDDI registry. [The majority of this functionality will be covered through the work of the GAT WP. ]
- Each network resource should provide information about its capabilities such as current bandwidth, maximum bandwidth, endpoints,...[ The majority of these requirements are to be covered as part of the GAT WP and the monitoring WP and appropriate “network API” will developed in such WP’s. ]
- Each host should provide information about its capabilities such as a operating system, CPU, network connection, flops, free persistent store, free memory, location...[ The majority of these requirements are to be covered as part of the GAT WP and the monitoring WP and appropriate “host API” will developed in such WP’s. ]
- Each host should provide for a means to migrate Triana units to itself which were there originally. In addition, and just importantly, each host should provide a means for removing such after they are no longer used. [ The majority of these requirements are to be covered as part of the GAT WP and the monitoring WP and appropriate “migration API” will developed in such WP’s. ]
- Each unit within Triana will implement an “optimization API” that will allow it to be quizzed about certain properties, for example, the amount of time required to complete a call to a port type. This information therefore can be used collectively to estimate the requirements from a complete algorithm or part of an algorithm. [ The majority of these requirements are to be covered as part of the GAT WP and the monitoring WP and appropriate “migration API” will developed in such WP’s. ]
- A “benchmarking API” for estimating the flops on a machine is required (the CPU speed the OS are irrelevant; we simply need to know how fast that machine will be for running a Triana unit). Giving this combination of information, an optimization algorithm can be

written to map a Triana network onto the Grid by splitting it up in the optimal way for the available resources. Users therefore, do not need to be involved in the distribution or parallelization of their code. [The majority of this functionality will be covered through the work of the GAT WP and the resources WP.]

- Caching issues not considered yet.

Form this enumeration of requirements we see that the majority of requirements imposed on the optimization metadata are met through the work of other work packages. This also implies that WP3 should work closely with other WP's to insure that the solutions which they devise to these requirements are compatible with the inner working of Triana. The easiest manner in which to do so would be to insure that WSDL is used to describe any API a Triana unit will be required to support. Currently, OGSA is thought to be the desired manner in which to describe various units, be they Triana or not. This implies that the each Triana unit will indeed be described by a WSDL interface and indeed this was/is the motivation for requiring Triana used WSDL and WSFL.

In addition, beyond the various port types and service provider types which the other WP's require each unit to implement we should mention that WP3 will also intend to follow very closely the development of WSEL as this emerging standard addresses many of the issues touched upon in this section.

## 7 Monitoring Metadata

Use Cases:

1. The user needs an estimation of how long it will take to run a Triana unit and a Triana algorithm.
2. Having started a Triana algorithm, the user wants to know the state of the execution of the Triana algorithm at any time.

Requirements:

1. Network Monitor - Monitor the bandwidth available and used by the Triana algorithm.
2. Host Monitoring - Monitor the CPU, disk, memory, storage, flops, Triana version, jdk, security,...
3. Job Monitoring - Monitor the progress, status (not started, started, stopped, finished),...

Currently, we are thinking of attacking this problem via two methods.

- Use the order of the algorithm directly
- Use self-learning, empirical estimation of the order of an algorithm

The self-learning, empirical estimation of the order of an algorithm does not need to know the direction relationship between its dimensionality and its timing. Calculate the flops required for the algorithm by using an empirical observation - this can be normalized using successive measurements. Also run the algorithm and divide the time taken by the flops.

$$T = T_{ALG} / (N * FLOPS)$$

where

$T_{ALG}$  = Timing of the algorithm (in seconds)

$T$  = Timing for unit data set on this machine

$N$  = dimensionality of the data set (number of elements in data set for vectors)

## 8 Triana Algorithm Metadata

The typical use cases for *Triana algorithm* metadata include:

1. *Triana algorithm* metadata can be used as a record of the “experiment” and this record can be used to reconstruct the “experiment” and also pass it around to other scientists for replication.
2. To enable automatic translation from a “task graph” created in another dataflow or workflow environment into one that would be “equivalent” within Triana.
3. To allow for a *recursively composed Triana algorithm* to be treated as a *Triana unit*.
4. To allow the user to refer to a *recursively composed Triana algorithm* by a pseudonym.

The use cases motivate a number of requirements for *Triana algorithm* metadata:

1. Record of *Triana units* comprising a *Triana algorithm*
2. Connections between the *Triana units* comprising *Triana algorithm*
3. Record of data flow between *Triana units* comprising a *Triana algorithm*
4. Location of *Triana units* comprising a *Triana algorithm* if appropriate
5. Manner in which to obtain *Triana units* for *Triana algorithm* “dynamic” or “static”

Also *recursive composition* motivates a number of requirements for *Triana algorithm* metadata

1. Name of *recursively composed Triana algorithm*
2. Description of *recursively composed Triana algorithm*
3. Input/output type of *recursively composed Triana algorithm*
4. Parameters name/value of *recursively composed Triana algorithm*
5. Number of input/output nodes of *recursively composed Triana algorithm*
6. Help files name and location of *recursively composed Triana algorithm*

As mentioned above, GridLab is planning to implement *remote Triana units* using WebServices. *Local Triana units* are to be implemented using Java with a WSDL interface. As of this current moment in time, there is a standardization effort WSFL, spearheaded by IBM, to define the metadata required for the composition of Web Services. We suggest adopting the IBM metadata standard WSFL to describe *Triana algorithms*. This will maximize the interoperability of Triana with other WebService workflow applications and insure that Triana can have the maximal utility with our without GAT. Now, let us examine in more detail WSFL through a short quote of the WSFL specification [2].

## 9 Web Services Flow Language (A Quick Tour )

The WSFL, Web Services Flow Language, is an XML language for the description of Web Services compositions. WSFL considers two types of Web Services compositions:

- The first type specifies the appropriate usage pattern of a collection of Web Services, in such a way that the resulting composition describes how to achieve a particular business goal; typically, the result is a description of a business process.
- The second type specifies the interaction pattern of a collection of Web Services; in this case, the result is a description of the overall partner interactions.

### 9.1 Flow Models

In the first case, a composition is created by describing how to use the functionality provided by the collection of composed Web Services. This is also known as flow composition, orchestration, or choreography of Web Services. WSFL models these compositions as specifications of the execution sequence of the functionality provided by the composed Web Services. Execution orders are specified by defining the flow of control and data between Web Services. For this reason, in this document, we will also use the term flow model to refer to the first type of Web Services compositions. Flow models can especially be used to model business processes or workflows based on Web Services.

### 9.2 Global Models

In the second case, no specification of an execution sequence is provided. Instead, the composition provides a description of how the composed Web Services interact with each other. The interactions are modeled as links between endpoints of the Web Services' interfaces, each link corresponding to the interaction of one Web Service with an operation of another Web Service's interface. Because of the decentralized or distributed nature of these interactions, we will use the term global model in this document to refer to this type of Web Services composition.

### 9.3 Recursive Composition

WSFL provides extensive support for the recursive composition of services: In WSFL, every Web Service composition (a flow model as well as a global model) can itself become a new Web Service, and can thus be used as a component of new compositions. The ability to do recursive composition of Web Services provides scalability to the language and support for top-down progressive refinement design as well as for bottom-up aggregation. For these reasons, recursive composition has been a central requirement in the design of the WSFL language.

## 9.4 A World-Wind Tour of WSFL

The purpose of a WSFL document is to define the composition of Web Services as a flow model or a global model. Both models have a declared public interface and an internal compositional structure. The composition assumes that the Web Services being composed support certain public interfaces, which can be specified as a single port type or as a collection of port types. We call this collection a *service provider type*.

The following code is a simplified example of a WSFL service composition defining a flow model called totalSupplyFlow. The syntax of many elements has been abbreviated in the interest of conciseness. The example assumes a set of WSDL port type and operation definitions as public interface of the service provider types referred to: the supplier and shipper service provider types are somehow assumed by the flow model; the totalSupply service provider type appears to be defined by the flow model, but it has been already defined somewhere else, which is perfectly valid. Note that the flow model imposes “sequencing constraints” for the execution of operations of the totalSupply service provider type.

```
<flowModel name="totalSupplyFlow" serviceProviderType="totalSupply">
  <serviceProvider name="mySupplier" type="supplier">
    <locator type="static" service="qualitySupply.com"/>
  </serviceProvider>
  <serviceProvider name="myShipper" type="shipper">
    <locator type="static" service="worldShipper.com"/>
  </serviceProvider>
  <activity name="processPO">
    <performedBy serviceProvider="mySupplier"/>
    <implement>
      <export>
        <target portType="totalSupplyPT" operation="sendProcOrder"/>
      </export>
    </implement>
  </activity>
  <activity name="acceptShipmentRequest">
    <performedBy serviceProvider="myShipper"/>
    <implement>
      <export>
        <target portType="totalSupplyPT" operation="sendSR"/>
      </export>
    </implement>
  </activity>
</flowModel>
```

```
<activity name="processPayment">
  <performedBy serviceProvider="mySupplier"/>
  <implement>
    <export>
      <target portType="totalSupplyPT"operation="sendPayment"/>
    </export>
  </implement>
</activity>
<controlLink source="processPO" target="acceptShipmentRequest"/>
<dataLink source="processPO" target="acceptShipmentRequest">
  <map sourceMessage="anINVandSR" targetMessage="anSR"/>
</dataLink>
</flowModel>
```

The totalSupplyFlow flow model specifies how to collaborate with two service provider types in order to offer to their joint customers a complete business process. Each of the two service providers used within the flow model is represented by a separate `<serviceProvider>` element. One service provider is of type supplier and is referred to as mySupplier in the flow model. The other service provider is of type shipper and is called myShipper. Both service providers contain “binding” information as well. This information is provided by means of a `<locator>` element, which specifies the actual service that will be used when the model is instantiated. In this case, binding information is “static,” but more dynamic binding schemes are possible.

The business process represented by the totalSupplyFlow flow model consists of three business tasks, called activities, that have to be performed in order to successfully complete the business process: A purchase order has to be processed, a shipment request must be accepted, and money has to be received. Each of these activities is specified by a separate `<activity>` element.

In our code example, the activities cannot be performed in any order, but there is a sequencing constraint between them: the processing of the purchase order by the supplier must precede the acceptance of the shipping request by the shipper; the money can be received at any time. The precedence rule is specified by simply connecting the two corresponding activities. Two kinds of connections are established, a control connection (through a `<controlLink>` element), and a data connection (through a `<dataLink>` element).

While the first connects the completion of one activity to the execution of another, the second connection represents a data exchange between the two. Note the `<map>` element nested inside the data link: it specifies what information needs to be transferred between the two linked activities. Also note that the separation of control flow and data flow is very helpful. For example, a service might only be enabled after the completion of another service without explicitly passing data from the former to the latter.

Web Services interact in a peer-to-peer manner. This pattern is immediately reflected by the interacting operations. For example, if a flow sends out a message via a notification operation, this operation corresponds to a one-way operation at a service provider. Pairs of corresponding operations in this sense are referred to as dual operations. In our example, the activity processPO has to send out a process order. For this purpose, the totalSupply service provider type declared by the flow model is assumed to include a port type totalSupplyPT with a sendProcOrder operation, which implements the activity. An `<implement>` element establishes this relation between an activity and its implementing operation. The service provider who is sup-

posed to interact with an activity's implementation (for example, to process the message sent) is defined through a <performedBy> element.

To define the public interface of the composition, the <flowModel> element includes a declaration of the supported service provider type as an attribute of the flow model, and a mapping of operations of the port types of this service provider type to activities of the flow model. This mapping is specified by an <export> element, which relates an activity of the flow model and an operation of its public interface. This mapping defines the effect of each operation by relating it to the execution of the internal composition. The public interface defines the interaction of a flow model with the "outside," that is, it specifies which messages are sent and which are used.

Typically, the operations of the public interface of the composition are not dual to any operation of the service providers to interact with, that is, messages are not simply sent to "anybody" or accepted by "anybody." Messages are related to a particular operation of a particular port type of the performing service provider. The relation between an operation of the public interface of a flow model and an operation of a service provider is established through a <plugLink> element. Thus, a plug link represents the inherent client/server structure of a Web Service.

In WSFL, plug links are typically specified within a <globalModel> element (although plug links can be specified "inline" within a flow model). Note that the advantage of separating plug links from flow models is that relations between operations of arbitrary port types or service provider types can be defined, whether they stem from a flow model or not. From a flow model perspective, a global model makes the interactions between service providers explicit. The following example specifies the interactions between a supplier, a shipper, and a total supplier.

```
<globalModel name="mySupplyChain" serviceProviderType="supplyChain">
  <serviceProvider name="mySupplier" type="supplier"/>
  <serviceProvider name="myShipper" type="shipper"/>
  <serviceProvider name="myTotalSupply" type="totalSupply">
    <export>
      <source portType="supplyLifecycle" operation="spawn"/>
      <target portType="manageChain" operation="order"/>
    </export>
  </serviceProvider>
  <plugLink>
    <source serviceProvider="myTotalSupply"portType="totalSupplyPT"
      operation="sendProcOrder"/>
    <target serviceProvider="mySupplier"portType="suppSvr"operation="procPO"/>
  </plugLink>
  <plugLink>
    <source serviceProvider="myTotalSupply"portType="totalSupplyPT"
      operation="sendPayment"/>
    <target serviceProvider="mySupplier"portType="suppSvr"operation="recPay"/>
  </plugLink>
  <plugLink>
    <source serviceProvider="myTotalSupply"portType="totalSupplyPT"
      operation="sendSR"/>
    <target serviceProvider="myShipper"portType="shipSvr"operation="recSR"/>
  </plugLink>
</globalModel>
```

In the example, the supplier service provider type is assumed to support the port type `suppSrv` with two operations: one for processing a purchase order, and one for receiving a payment (`processPO` and `recPay`). The supplier service provider type may also define restrictions on the sequencing of the two operations (for example, the execution of the first operation must precede the execution of the second). For this purpose, the service provider type could be defined as the public interface of another flow model (but this is not done in our example). The shipper service provider type is assumed to support the `shipSrv` port type including an operation for accepting and processing shipping requests (`recSR`). The `mySupplyChain` global model now plug links the operations of these two service provider types with the `totalSupply` service provider type declared by the `totalSupplyFlow` flow model.

As each composition, the `mySupplyChain` global model of the example declares a service provider type named `supplyChain`. This is done through an attribute of the `<globalModel>` element. The service provider `myTotalSupply` exports the operation `spawn` from its port type `supplyLifecycle` to the operation `order` of the port type `manageChain` that represents the public interface of the sample global model. The `spawn` operation is a lifecycle operation that allows the flow to kick off an instance of the `totalSupplyFlow` flow model: Thus, invoking the `order` operation, which is delegated to the `spawn` operation, will kick off the flow and will finally result in making use of the specified plug links. The first plug link specifies that the “`sendProcOrder`” operation of the public interface of the flow sends a message to the `procPO` operation of the `suppSrv` port type of the supplier. The other plug links are similar.

For more detail on the full WSFL syntax the interested reader can refer to the WSFL specification [2] or the WSFL schema which is reproduced in the appendix.

## 10 Web Services Flow Language (Triana Specifics)

Triana imposes very few extra constraints on WSFL as WSFL fits the various requirements that Triana's use cases impose on the *Triana algorithm* metadata. There are only two extra requirements that are of any note: one that is imposed by *recursive composition* and one that is imposed by the use of WSDL/WSFL to describe local Triana units. First let us look at the requirement that is imposed by *recursive composition*.

As mentioned previously, Triana requires that each Triana unit implement various port types. As to appear as a Triana unit, each recursively composed Triana algorithm is required to implement a service provider type, specified using the `serviceProviderType` attribute on the `flowModel` element, which contains all the port types required of a Triana unit.

This is accomplished by implementing the following service provider type whose WSFL document is located at <http://www.gridlab.org/wp3/trianaserviceprovidertypes.wsfl> :

```
<definitions name="TrianaServiceProviderTypes"
  targetNamespace="http://www.gridlab.org/wp3/serviceprovidertypes"
  xmlns:tns="http://www.gridlab.org/wp3/serviceprovidertypes"
  xmlns:ports="http://www.gridlab.org/wp3/porttypes"
  xmlns="!!!"/>

  <serviceProvider name="TrianaRecursivelyComposedServiceProviderType">
    <portType name="TrianaRecursivelyComposedPortType">
      <import portType="ports:TrianaUnitNamePortType"/>
      <import portType="ports:TrianaIconHTMLGUIPortType"/>
      <import portType="ports:TrianaInformationHTMLGUIPortType"/>
      <import portType="ports:TrianaConfigurationHTMLGUIPortType"/>
    </portType>
  </serviceProvider>
</definitions>
```

Let us next look at the requirement that is imposed by the use of WSDL/WSFL. Usually WSDL is used to describe *remote* services. However, there is no reason that it can not be used to describe local services as well. In fact that is what we will be doing with Triana. However, using WSDL to describe local services raises a number of issues.

Normally a binding associates a wire protocol or a data format with a port type. The various W3C bindings are currently all network enabled. ( For example SOAP and HTTP. ) No W3C means exists to bind a port type locally. There are at least three avenues one can take in confronting this problem when one is attempting to use WSDL and WSFL:

- Don't use WSDL/WSFL to describe local services
- Use WSDL/WSFL and make all local services appear to be remote services, i.e. use a SOAP binding and define to resulting port to point to localhost.
- Use a WSDL binding extension to create a new type of binding and use the `<internal>` WSFL tag. ( See section 6 of the WSFL specification [2]. In particular, pay attention to subsection 6.3 as it provides an example of a WSDL extension element for Java. )

This final method is the path that Triana will take as it is most in line with the spirit and letter of the WSDL and WSFL specifications. As mentioned previously, IBM has implemented a WSDL binding extension for Java in their WSIF [3]. In addition, a WSDL binding extension for Java is to become an extension to Java through JSR 110 [4] and thus become ubiquitous.

## A Appendix: WSDL Schema

```
<schema xmlns="http://www.w3.org/2000/10/XMLSchema"
  xmlns:wSDL="http://schemas.xmlsoap.org/wSDL/"
  targetNamespace="http://schemas.xmlsoap.org/wSDL/"
  elementFormDefault="qualified">
  <element name="documentation">
    <complexType mixed="true">
      <choice minOccurs="0" maxOccurs="unbounded">
        <any minOccurs="0" maxOccurs="unbounded"/>
      </choice>
      <anyAttribute/>
    </complexType>
  </element>
  <complexType name="documented" abstract="true">
    <sequence>
      <element ref="wSDL:documentation" minOccurs="0"/>
    </sequence>
  </complexType>
  <complexType name="openAtts" abstract="true">
    <annotation>
      <documentation>
        This type is extended by component types
        to allow attributes from other namespaces to be added.
      </documentation>
    </annotation>
    <sequence>
      <element ref="wSDL:documentation" minOccurs="0"/>
    </sequence>
    <anyAttribute namespace="##other"/>
  </complexType>
  <element name="definitions" type="wSDL:definitionsType">
    <key name="message">
      <selector xpath="message"/>
      <field xpath="@name"/>
    </key>
    <key name="portType">
      <selector xpath="portType"/>
      <field xpath="@name"/>
    </key>
    <key name="binding">
      <selector xpath="binding"/>
      <field xpath="@name"/>
    </key>
    <key name="service">
      <selector xpath="service"/>
      <field xpath="@name"/>
    </key>
  </element>
</schema>
```

```
<key name="import">
  <selector xpath="import"/>
  <field xpath="@namespace"/>
</key>
<key name="port">
  <selector xpath="service/port"/>
  <field xpath="@name"/>
</key>
</element>
<complexType name="definitionsType">
  <complexContent>
    <extension base="wsdl:documented">
      <sequence>
        <element ref="wsdl:import" minOccurs="0" maxOccurs="unbounded"/>
        <element ref="wsdl:types" minOccurs="0"/>
        <element ref="wsdl:message" minOccurs="0" maxOccurs="unbounded"/>
        <element ref="wsdl:portType" minOccurs="0" maxOccurs="unbounded"/>
        <element ref="wsdl:binding" minOccurs="0" maxOccurs="unbounded"/>
        <element ref="wsdl:service" minOccurs="0" maxOccurs="unbounded"/>
        <any namespace="##other" minOccurs="0" maxOccurs="unbounded">
          <annotation>
            <documentation>for extensibility elements</documentation>
          </annotation>
        </any>
      </sequence>
      <attribute name="targetNamespace" type="uriReference" use="optional"/>
      <attribute name="name" type="NMTOKEN" use="optional"/>
    </extension>
  </complexContent>
</complexType>
<element name="import" type="wsdl:importType"/>
<complexType name="importType">
  <complexContent>
    <extension base="wsdl:documented">
      <attribute name="namespace" type="uriReference" use="required"/>
      <attribute name="location" type="uriReference" use="required"/>
    </extension>
  </complexContent>
</complexType>
<element name="types" type="wsdl:typesType"/>
<complexType name="typesType">
  <complexContent>
    <extension base="wsdl:documented">
      <sequence>
        <any namespace="##other" minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

```
<element name="message" type="wsdl:messageType">
  <unique name="part">
    <selector xpath="part"/>
    <field xpath="@name"/>
  </unique>
</element>
<complexType name="messageType">
  <complexContent>
    <extension base="wsdl:documented">
      <sequence>
        <element ref="wsdl:part" minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
      <attribute name="name" type="NCName" use="required"/>
    </extension>
  </complexContent>
</complexType>
<element name="part" type="wsdl:partType"/>
<complexType name="partType">
  <complexContent>
    <extension base="wsdl:openAtts">
      <attribute name="name" type="NMTOKEN" use="optional"/>
      <attribute name="type" type="QName" use="optional"/>
      <attribute name="element" type="QName" use="optional"/>
    </extension>
  </complexContent>
</complexType>
<element name="portType" type="wsdl:portTypeType"/>
<complexType name="portTypeType">
  <complexContent>
    <extension base="wsdl:documented">
      <sequence>
        <element ref="wsdl:operation" minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
      <attribute name="name" type="NCName" use="required"/>
    </extension>
  </complexContent>
</complexType>
<element name="operation" type="wsdl:operationType"/>
<complexType name="operationType">
  <complexContent>
    <extension base="wsdl:documented">
      <choice>
        <group ref="wsdl:one-way-operation"/>
        <group ref="wsdl:request-response-operation"/>
        <group ref="wsdl:solicit-response-operation"/>
        <group ref="wsdl:notification-operation"/>
      </choice>
      <attribute name="name" type="NCName" use="required"/>
    </extension>
  </complexContent>
</complexType>
```

```
</complexContent>
</complexType>
<group name="one-way-operation">
  <sequence>
    <element ref="wsdl:input"/>
  </sequence>
</group>
<group name="request-response-operation">
  <sequence>
    <element ref="wsdl:input"/>
    <element ref="wsdl:output"/>
    <element ref="wsdl:fault" minOccurs="0" maxOccurs="unbounded"/>
  </sequence>
</group>
<group name="solicit-response-operation">
  <sequence>
    <element ref="wsdl:output"/>
    <element ref="wsdl:input"/>
    <element ref="wsdl:fault" minOccurs="0" maxOccurs="unbounded"/>
  </sequence>
</group>
<group name="notification-operation">
  <sequence>
    <element ref="wsdl:output"/>
  </sequence>
</group>
<element name="input" type="wsdl:paramType"/>
<element name="output" type="wsdl:paramType"/>
<element name="fault" type="wsdl:faultType"/>
<complexType name="paramType">
  <complexContent>
    <extension base="wsdl:documented">
      <attribute name="name" type="NMTOKEN" use="optional"/>
      <attribute name="message" type="QName" use="required"/>
    </extension>
  </complexContent>
</complexType>
<complexType name="faultType">
  <complexContent>
    <extension base="wsdl:documented">
      <attribute name="name" type="NMTOKEN" use="required"/>
      <attribute name="message" type="QName" use="required"/>
    </extension>
  </complexContent>
</complexType>
```

```
<complexType name="startWithExtensionsType" abstract="true">
  <complexContent>
    <extension base="wsdl:documented">
      <sequence>
        <any namespace="##other" minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
<element name="binding" type="wsdl:bindingType"/>
<complexType name="bindingType">
  <complexContent>
    <extension base="wsdl:startWithExtensionsType">
      <sequence>
        <element name="operation" type="wsdl:binding_operationType" minOccurs="0"
          maxOccurs="unbounded"/>
      </sequence>
      <attribute name="name" type="NCName" use="required"/>
      <attribute name="type" type="QName" use="required"/>
    </extension>
  </complexContent>
</complexType>
<complexType name="binding_operationType">
  <complexContent>
    <extension base="wsdl:startWithExtensionsType">
      <sequence>
        <element name="input" type="wsdl:startWithExtensionsType" minOccurs="0"/>
        <element name="output" type="wsdl:startWithExtensionsType" minOccurs="0"/>
        <element name="fault" minOccurs="0" maxOccurs="unbounded">
          <complexType>
            <complexContent>
              <extension base="wsdl:startWithExtensionsType">
                <attribute name="name" type="NMTOKEN" use="required"/>
              </extension>
            </complexContent>
          </complexType>
        </element>
      </sequence>
      <attribute name="name" type="NCName" use="required"/>
    </extension>
  </complexContent>
</complexType>
</element>
</sequence>
<attribute name="name" type="NCName" use="required"/>
</extension>
</complexContent>
</complexType>
<element name="service" type="wsdl:serviceType"/>
<complexType name="serviceType">
  <complexContent>
    <extension base="wsdl:documented">
      <sequence>
        <element ref="wsdl:port" minOccurs="0" maxOccurs="unbounded"/>
        <any namespace="##other" minOccurs="0"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

```
    <attribute name="name" type="NCName" use="required"/>
  </extension>
</complexContent>
</complexType>
<element name="port" type="wsdl:portType"/>
<complexType name="portType">
  <complexContent>
    <extension base="wsdl:documented">
      <sequence>
        <any namespace="##other" minOccurs="0"/>
      </sequence>
      <attribute name="name" type="NCName" use="required"/>
      <attribute name="binding" type="QName" use="required"/>
    </extension>
  </complexContent>
</complexType>
<attribute name="arrayType" type="string"/>
</schema>
```

## B Appendix: WSDL Triana Specifics

This appendix gathers the information related to the port types each Triana unit is required to implement.

### B.1 Triana Types

This is the XML schema of the various types which are required for Triana metadata:

```
<?xml version="1.0"?>
<schema targetNamespace="http://www.gridlab.org/wp3/schemas"
  xmlns="http://www.w3.org/2000/10/XMLSchema"
  xmlns:tns="http://www.gridlab.org/wp3/schemas">
  <simpleType name="guiweights">
    <restriction base="string">
      <enumeration value="LightWeight"/>
      <enumeration value="MiddleWeight"/>
      <enumeration value="HeavyWeight"/>
    </restriction>
  </simpleType>
  <element name="TrianaUnitName" type="string"/>
  <element name="TrianaGUILanguage" type="language"/>
  <element name="TrianaGUIWeight" type="tns:guiweights"/>
  <element name="TrianaIconHTMLGUI">
    <complexType>
      <sequence>
        <any namespace="http://www.w3.org/1999/xhtml"
          minOccurs="1" maxOccurs="unbounded"
          processContents="skip"/>
      </sequence>
    </complexType>
  </element>
  <element name="TrianaInformationHTMLGUI">
    <complexType>
      <sequence>
        <any namespace="http://www.w3.org/1999/xhtml"
          minOccurs="1" maxOccurs="unbounded"
          processContents="skip"/>
      </sequence>
    </complexType>
  </element>
</schema>
```

```
<element name="TrianaConfigurationHTMLGUI">
  <complexType>
    <sequence>
      <any namespace="http://www.w3.org/1999/xhtml"
          minOccurs="1" maxOccurs="unbounded"
          processContents="skip"/>
    </sequence>
  </complexType>
</element>
</schema>
```

## B.2 Triana Messages

This is the WSDL document detailing the various WSDL messages which are required for Triana metadata:

```
<?xml version="1.0"?>
<definitions name="TrianaMessages"
  targetNamespace="http://www.gridlab.org/wp3/messages"
  xmlns:tns="http://www.gridlab.org/wp3/messages"
  xmlns:xsd1="www.gridlab.org/wp3/schemas"
  xmlns="http://schemas.xmlsoap.org/wsdl/">

  <import namespace="http://www.gridlab.org/wp3/schemas"
    location="http://www.gridlab.org/wp3/trianaunit.xsd"/>

  <message name="TrianaUnitNameType">
    <part name="body" element="xsd1:TrianaUnitName"/>
  </message>

  <message name="TrianaGUIDescriptionType">
    <part name="body" element="xsd1:TrianaGUIDescription"/>
  </message>

  <message name="TrianaIconHTMLGUIType">
    <part name="body" element="xsd1:TrianaIconHTMLGUI"/>
  </message>

  <message name="TrianaInformationHTMLGUIType">
    <part name="body" element="xsd1:TrianaInformationHTMLGUI"/>
  </message>

  <message name="TrianaConfigurationHTMLGUIType">
    <part name="body" element="xsd1:TrianaConfigurationHTMLGUI"/>
  </message>
</definitions>
```

### B.3 Triana Port Types

This is the WSDL document detailing the various WSDL port types which are required for Triana metadata:

```
<?xml version="1.0"?>
<definitions name="TrianaPortTypes"
  targetNamespace="http://www.gridlab.org/wp3/porttypes"
  xmlns:tns="http://www.gridlab.org/wp3/porttypes"
  xmlns:xsd1="http://www.gridlab.org/wp3/messages"
  xmlns="http://schemas.xmlsoap.org/wsdl/">

  <import namespace="http://www.gridlab.org/wp3/messages"
    location="http://www.gridlab.org/wp3/trianamessages.wsdl"/>

  <portType name="TrianaUnitNamePortType">
    <operation name="GetTrianaUnitName">
      <output message="xsd1:TrianaUnitNameType"/>
    </operation>
  </portType>

  <portType name="TrianaIconHTMLGUIPortType">
    <operation name="GetTrianaIconHTMLGUI">
      <input message="xsd1:TrianaGUIDescriptionType"/>
      <output message="xsd1:TrianaIconHTMLGUIType"/>
    </operation>
  </portType>

  <portType name="TrianaInformationHTMLGUIPortType">
    <operation name="GetTrianaInformationHTMLGUI">
      <input message="xsd1:TrianaGUIDescriptionType"/>
      <output message="xsd1:TrianaInformationHTMLGUIType"/>
    </operation>
  </portType>

  <portType name="TrianaConfigurationHTMLGUIPortType">
    <operation name="GetTrianaConfigurationHTMLGUI">
      <input message="xsd1:TrianaGUIDescriptionType"/>
      <output message="xsd1:TrianaConfigurationHTMLGUIType"/>
    </operation>
  </portType>
</definitions>
```

## C Appendix: WSFL Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2000/10/XMLSchema"
xmlns:wSDL="http://schemas.xmlsoap.org/wSDL/"
xmlns:wsfl="http://schemas.xmlsoap.org/wsfl/"
targetNamespace="http://schemas.xmlsoap.org/wsfl/"
elementFormDefault="qualified">
<simpleType name="QNameList">
<list itemType="QName"/>
</simpleType>
<simpleType name="NCNameList">
<list itemType="NCName"/>
</simpleType>
<simpleType name="locatorTypeType">
<restriction base="string">
<enumeration value="static"/>
<enumeration value="local"/>
<enumeration value="any"/>
<enumeration value="UDDI"/>
<enumeration value="mobility"/>
</restriction>
</simpleType>
<simpleType name="selectionPolicyType">
<restriction base="string">
<enumeration value="first"/>
<enumeration value="random"/>
<enumeration value="user-defined"/>
</restriction>
</simpleType>
<simpleType name="bindTimeType">
<restriction base="string">
<enumeration value="startup"/>
<enumeration value="firstHit"/>
</restriction>
</simpleType>
<simpleType name="whenType">
<restriction base="string">
<enumeration value="deferred"/>
<enumeration value="immediate"/>
</restriction>
</simpleType>
<simpleType name="orderType">
<restriction base="string">
<enumeration value="LWW"/>
<enumeration value="RFW"/>
<enumeration value="random"/>
</restriction>
</simpleType>
```

```
<element name="definitions" type="wsfl:definitionsType">
  <unique name="flowModelName">
    <selector xpath="flowModel"/>
    <field xpath="@name"/>
  </unique>
</element>
<complexType name="definitionsType">
  <sequence>
    <element name="import" type="wsfl:importType"
      minOccurs="0" maxOccurs="unbounded"/>
    <element name="serviceProviderType"
      type="wsfl:serviceProviderTypeType"
      minOccurs="0" maxOccurs="unbounded"/>
    <element ref="wsfl:flowModel"
      minOccurs="0" maxOccurs="unbounded"/>
    <element ref="wsfl:globalModel"
      minOccurs="0" maxOccurs="unbounded"/>
  </sequence>
  <attribute name="targetNamespace" type="uriReference"/>
</complexType>
<complexType name="importType">
  <attribute name="namespace" type="uriReference" use="required"/>
  <attribute name="location" type="uriReference" use="required"/>
</complexType>
<complexType name="serviceProviderTypeType">
  <sequence>
    <element name="portType" type="wsfl:portTypeType"
      minOccurs="0" maxOccurs="unbounded"/>
    <element name="import"
      minOccurs="0" maxOccurs="unbounded">
      <complexType>
        <attribute name="portType" type="QName"/>
      </complexType>
    </element>
  </sequence>
</complexType>
<complexType name="portTypeType">
  <complexContent>
    <extension base="wsdl:portTypeType">
      <sequence>
        <element name="import"
          minOccurs="0" maxOccurs="unbounded">
          <complexType>
            <attribute name="portType" type="QName"/>
            <attribute name="operation" type="NCName"/>
          </complexType>
        </element>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

```
</sequence>
</extension>
</complexContent>
</complexType>
<element name="flowModel" type="wsfl:flowModelType">
  <key name="providerName">
    <selector xpath="serviceProvider"/>
    <field xpath="@name"/>
  </key>
  <key name="activityName">
    <selector xpath="activity"/>
    <field xpath="@name"/>
  </key>
  <unique name="controlLinkName">
    <selector xpath="controlLink"/>
    <field xpath="@name"/>
  </unique>
  <unique name="dataLinkName">
    <selector xpath="dataLink"/>
    <field xpath="@name"/>
  </unique>
  <keyref name="activityProviderRef" refer="providerName">
    <selector xpath="activity"/>
    <field xpath="@serviceProvider"/>
  </keyref>
  <keyref name="linkActivityRef" refer="activityName">
    <selector xpath="controlLink|dataLink"/>
    <field xpath="@source|@target"/>
  </keyref>
  <keyref name="implActivityRef" refer="providerName">
    <selector xpath="implement|import"/>
    <field xpath="@serviceProvider"/>
  </keyref>
</element>
<complexType name="flowModelType">
  <sequence>
    <element name="flowSource"
      type="wsfl:flowSourceType"
      minOccurs="0"/>
    <element name="flowSink"
      type="wsfl:flowSinkType"
      minOccurs="0"/>
    <element name="serviceProvider"
      type="wsfl:serviceProviderType"
      minOccurs="1" maxOccurs="unbounded"/>
    <group ref="wsfl:activityFlowGroup"/>
  </sequence>
```

```
<attribute name="name" type="NCName" use="required"/>
<attribute name="serviceProviderType" type="QName"/>
</complexType>
<group name="activityFlowGroup">
  <sequence>
    <element name="export" type="wsfl:exportType"
      minOccurs="0" maxOccurs="unbounded"/>
    <element name="activity" type="wsfl:activityType"
      minOccurs="0" maxOccurs="unbounded"/>
    <element name="controlLink" type="wsfl:controlLinkType"
      minOccurs="0" maxOccurs="unbounded"/>
    <element name="dataLink" type="wsfl:dataLinkType"
      minOccurs="0" maxOccurs="unbounded"/>
  </sequence>
</group>
<complexType name="serviceProviderType">
  <sequence>
    <element name="locator" type="wsfl:locatorType"
      minOccurs="0"/>
    <element name="export" type="wsfl:exportType"
      minOccurs="0" maxOccurs="unbounded"/>
  </sequence>
  <attribute name="name" type="NCName" use="required"/>
  <attribute name="type" type="QName" use="required"/>
</complexType>
<complexType name="locatorType">
  <sequence>
    <any namespace="##other"
      minOccurs="0" maxOccurs="unbounded"/>
  </sequence>
  <attribute name="type" type="wsfl:locatorTypeType" />
  <attribute name="service" type="QName"/>
  <attribute name="bindTime" type="wsfl:bindTimeType"/>
  <attribute name="selectionPolicy"
    type="wsfl:selectionPolicyType"/>
  <attribute name="activity" type="NCName"/>
  <attribute name="message" type="NCName"/>
  <attribute name="messagePart" type="NCName" />
  <attribute name="dataField" type="string"/>
  <attribute name="default" type="QName"/>
  <attribute name="invoke" type="string"/>
</complexType>
<complexType name="flowSourceType">
  <sequence>
    <element ref="wsdl:output"/>
  </sequence>
  <attribute name="name" type="NCName" use="required"/>
</complexType>
```

```
<complexType name="flowSinkType">
  <sequence>
    <element ref="wsdl:input"/>
  </sequence>
  <attribute name="name" type="NCName" use="required"/>
</complexType>
<complexType name="endPointType">
  <attribute name="serviceProvider" type="NCName"/>
  <attribute name="serviceProviderType" type="NCName"/>
  <attribute name="portType" type="QName" use="required"/>
  <attribute name="operation" type="NCName" use="required"/>
</complexType>
<complexType name="internalType">
  <complexContent>
    <extension base="wsfl:endPointType"/>
    <sequence>
      <element name="plugLink" type="wsfl:plugLinkType"/>
    </sequence>
  </extension>
</complexContent>
</complexType>
<complexType name="joinType">
  <attribute name="condition" type="wsfl:NCNameList"
  use="required"/>
  <attribute name="when" type="wsfl:whenType"
  use="default" value="deferred"/>
</complexType>
<complexType name="materializeType">
  <choice>
    <element name="mapPolicy">
      <complexType>
        <attribute name="order" type="wsfl:orderType"
        use="default" value="LWW"/>
      </complexType>
    </element>
    <element name="construction">
      <complexType>
        <attribute name="type" type="string"
        use="default" value="XSLT"/>
        <attribute name="location" type="string"/>
      </complexType>
    </element>
  </choice>
</complexType>
<complexType name="activityType">
  <complexContent>
    <extension base="wsdl:operationType">
      <sequence>
        <element name="performedBy">
```

```
<complexType>
  <attribute name="serviceProvider"
    type="NCName"/>
</complexType>
</element>
<element name="implement">
  <complexType>
    <choice>
      <element name="internal"
        type="wsfl:internalType">
      <element name="export"
        type="wsfl:exportType"/>
    </choice>
  </complexType>
</element>
<element name="join" type="wsfl:joinType"
  minOccurs="0"/>
<element name="materialize"
  type="wsfl:materializeType"
  minOccurs="0"/>
<any namespace="##other"
  minOccurs="0" maxOccurs="unbounded"/>
</sequence>
<attribute name="name" type="NCName"/>
<attribute name="exitCondition" type="string"/>
</extension>
</complexContent>
</complexType>
<complexType name="linkType">
  <attribute name="name" type="NCName"/>
  <attribute name="source" type="NCName"/>
  <attribute name="target" type="NCName"/>
</complexType>
<complexType name="controlLinkType">
  <complexContent>
    <extension base="linkType">
      <attribute name="transitionCondition" type="string"/>
      <attribute name="result" type="NCName"/>
    </extension>
  </complexContent>
</complexType>
<complexType name="dataLinkType">
  <complexContent>
    <extension base="linkType">
      <sequence>
        <element ref="map" minOccurs="0"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

```
</sequence>
</extension>
</complexContent>
</complexType>
<complexType name="plugLinkType">
  <sequence>
    <element name="source" type="wsfl:endpointType"
      minOccurs="0"/>
    <element name="target" type="wsfl:endpointType"/>
    <element ref="map" minOccurs="0"/>
    <element name="locator" type="wsfl:locatorType"
      minOccurs="0"/>
  </sequence>
</complexType>
<element name="map">
  <complexType>
    <attribute name="sourceMessage" type="NCName"/>
    <attribute name="targetMessage" type="NCName"/>
    <attribute name="sourcePart" type="NCName"/>
    <attribute name="targetPart" type="NCName"/>
    <attribute name="sourceField" type="NCName"/>
    <attribute name="targetField" type="NCName"/>
    <attribute name="converter" type="string"/>
  </complexType>
</element>
<complexType name="exportType">
  <sequence>
    <element name="source" type="endPointType"
      minOccurs="0"/>
    <element name="target" type="endPointType"/>
    <element ref="wsfl:map"
      minOccurs="0" maxOccurs="unbounded"/>
    <element name="plugLink" type="wsfl:plugLinkType"
      minOccurs="0"/>
  </sequence>
  <attribute name="lifecycleAction" type="NCName"/>
</complexType>
<element name="globalModel" type="wsfl:globalModelType">
  <unique name="globalProviderName">
    <selector xpath="serviceProvider"/>
    <field xpath="@name"/>
  </unique>
</element>
<complexType name="globalModelType">
  <choice>
    <sequence>
      <element name="serviceProvider"
        type="wsfl:serviceProviderType"
        maxOccurs="unbounded"/>
    </sequence>
  </choice>
</complexType>
```

```
<element name="plugLink" type="wsfl:plugLinkType"
minOccurs="0" maxOccurs="unbounded"/>
</sequence>
<element name="binding" maxOccurs="unbounded">
<complexType>
<element name="serviceProvider"
type="serviceProviderRefType"
maxOccurs="unbounded"/>
</complexType>
</element>
</choice>
<attribute name="name" type="NCName" use="required"/>
<attribute name="serviceProviderType" type="Qname"/>
<attribute name="ref" type="QName"/>
</complexType>
<complexType name="serviceProviderRefType">
<complexContent>
<restriction base="serviceProviderType"/>
<attribute name="type" use="prohibited"/>
</restriction>
</complexContent>
</complexType>
<simpleType name="state" base="string">
<enumeration value="Running"/>
<enumeration value="Suspended"/>
</simpleType>
<simpleType name="Success" base="string">
<enumeration value="OK"/>
</simpleType>
<element name="FlowInstanceID" type="string"/>
<element name="FlowInstanceState" type="wsfl:state"/>
<element name="FlowInstanceLastModificationTime" type="dateTime"/>
<element name="FlowInstanceCreationTime" type="dateTime"/>
<complexType name="SpawnResult">
<sequence>
<element ref="wsfl:FlowInstanceID"/>
<element ref="wsfl:FlowInstanceCreationTime"
minOccurs="0" maxOccurs="1"/>
</sequence>
</complexType>
<complexType name="EnquiryInput">
<sequence>
<element ref="wsfl:FlowInstanceID"/>
</sequence>
</complexType>
<complexType name="EnquiryResult">
<sequence>
<element ref="wsfl:FlowInstanceID"/>
<element ref="wsfl:FlowInstanceState"/>
</sequence>
</complexType>
```

```
<element ref="wsfl:FlowInstanceCreationTime"
minOccurs="0" maxOccurs="1"/>
<element ref="wsfl:FlowInstanceLastModificationTime"
minOccurs="0" maxOccurs="1"/>
</sequence>
</complexType>
<complexType name="Fault">
<sequence>
<element name="MainCode" type="integer"/>
<element name="SubCode" type="integer"
minOccurs="0" maxOccurs="1"/>
<element name="MessageText" type="string"
minOccurs="0" maxOccurs="1"/>
</sequence>
</complexType>
</schema>
```

## D Appendix: WSFL Triana Specifics

### D.1 Triana Service Provider Types

This is the WSFL document detailing the various WSFL service provider type which are required for Triana metadata:

```
<definitions name="TrianaServiceProviderTypes"
targetNamespace="http://www.gridlab.org/wp3/serviceprovidertypes"
xmlns:tns="http://www.gridlab.org/wp3/serviceprovidertypes"
xmlns:ports="http://www.gridlab.org/wp3/porttypes"
xmlns="!!!">

  <serviceProvider name="TrianaRecursivelyComposedServiceProviderType">
    <portType name="TrianaRecursivelyComposedPortType">
      <import portType="ports:TrianaUnitNamePortType"/>
      <import portType="ports:TrianaIconHTMLGUIPortType"/>
      <import portType="ports:TrianaInformationHTMLGUIPortType"/>
      <import portType="ports:TrianaConfigurationHTMLGUIPortType"/>
      ...
    </portType>
  </serviceProvider>
</definitions>
```

## References

- [1] <http://www.w3.org/TR/wsdl>
- [2] <http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf>
- [3] <http://www.alphaworks.ibm.com/tech/wsif>
- [4] <http://jcp.org/jsr/detail/110.jsp>