



IST-2001-32133

GridLab - A Grid Application Toolkit and Testbed

## Triana Metadata Specification

---

Author(s):	Ian Taylor, Shalil Majithia, Matthew Shields, Ian Wang
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. There are five issues considered in this document. Metadata requirements at unit level, Triana types XML serialization, Task graph metadata, metadata required for optimizing the Triana network and finally, metadata required for monitoring the network.



## Contents

<b>1</b>	<b>Status of this Document</b>	<b>2</b>
<b>2</b>	<b>Introduction</b>	<b>2</b>
<b>3</b>	<b>Unit Metadata</b>	<b>3</b>
<b>4</b>	<b>Triana Types Representation</b>	<b>4</b>
<b>5</b>	<b>Task Graph Metadata</b>	<b>4</b>
<b>6</b>	<b>Optimization Metadata</b>	<b>6</b>
<b>7</b>	<b>Monitoring Metadata</b>	<b>7</b>

## 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 usually defined as “data about data”. It is the supporting information that provides context to a resource, i.e. tool, task graph etc. Metadata in Triana includes information at various levels e.g. at the Unit, task graph, network level etc.

Triana needs metadata for several reasons:

- It will be difficult to reuse other people’s tools and task graphs without metadata to provide the context for these objects. A scientist considering reusing someone else’s tool or task graph will need to know things such as: what the tool does, the allowed input types, output types, parameters, the author, help files and so on.
- 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 flow and extracting Quality of Service (QoS) parameters.

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

- To support searches of a unit repository.
- To enable automatic discovery of tools published on remote websites.
- To enable the user to reconstruct a network
- To enable the user to ascertain the status of a network

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

The living document concept illustrates the use of metadata within Triana. A living document is a web page that contains an example of a data-flow network (simulated within Triana) that comes alive when a user clicks on it and instantiates Triana and shows the complete reconstruction of the experimental results. To do this, we need to be able to save units state in a standardized way, standardize the Triana data types into a common representation and record data flow information (within a task graph).

### 3 Unit Metadata

The typical use cases for Unit metadata include:

1. Search for a 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 types when connecting units.
3. To enable automatic translation from a task graph created in another dataflow or workflow environment into one that would be equivalent within Triana.
4. To allow the user to refer to a tool by a pseudo name, not necessarily the class name.
5. To enable the users to use the most recent version of the 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 unit metadata:

1. Name of unit and id;
2. Number of input/output nodes;
3. Description;
4. Help files name and location;
5. input/output types
6. Parameter name/value;
7. Class name;
8. Version no.

Below is an XML example of this information:

```
<?xml version="1.0" encoding="UTF-8"?>
<tool name="" toolid="">
  <version/>
  <javaclass name =""/>
  <inportnum/>
  <outportnum/>
  <description/>
  <input>
    <type/>
  </input>
  <output>
    <type/>
  </output>
  <helpfile/>
  <parameters>
    <param name="" value=""/>
  </parameters>
</tool>
```

## 4 Triana Types Representation

It is proposed to serialize Triana Types to XML documents to replace the ASCII conversions we have in Triana now. There are several open source programs available (Zeus [?], JSX [?], JATO [?], JAXB [?]). All four will be looked into and the best one fitting in with Triana requirements will be adopted.

## 5 Task Graph Metadata

A Task Graph within the context of Triana refers to an XML document which represents a group unit or the entire Triana Network. Potential use cases for task graph metadata:

1. a task graph will be used by OCL to execute the entire network as set out;
2. a task graph representing a group unit will be used to provide details on tools available;
3. a task graph 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.
4. a task graph can be used as a check-pointing mechanism to migrate if necessary.

For example, this interface could be as simple as:

```
String saveState();
// which returns a string of the checkpointed Triana i.e. to serialize Triana's state

Void restoreState(String state);
// to restore the state within Triana i.e. to deserialize
```

Requirements include:

1. name and id of task graph;
2. description
3. tasks comprising the task graph
4. connections between the tasks
5. layout metadata
6. metadata on host machine;

A simple task graph example is included below:

```
<?xml version="1.0" encoding="UTF-8"?>
<taskgraph>
  <name> </name>
  <description> </description>
  <parameters>
    <param name="" value= ""/>
  </parameters>
  <!-- List of tools participating in this task graph -->
  <tasks>
    <task toolname="" taskname="" taskid= "">
      <description> </description>
      <data>
        <inportnum> </inportnum>
        <outportnum> </outportnum>
        <input>
          <type> </type>
        </input>
        <output>
          <type> </type>
        </output>
      </data>
      <parameters>
        <inportnum> </inportnum>
        <outportnum> </outportnum>
        <input>
          <param name="" value=""/>
        </input>
        <output>
          <param name="" value=""/>
        </output>
        <param name="" value=""/>
      </parameters>
    </task>
  </tasks>
  <!-- List of connections between nodes on tools in this task graph -->
  <connections>
    <connection>
```

```
        <source taskid="" node=""/>
        <target taskid="" node=""/>
    </connection>
</connections>
<!-- information about how tools within this task graph are visually
    grouped and displayed -->
<layout>
    <task taskid=""/>
    <group>
        <task taskid=""/>
        <task taskid=""/>
    </group>
    <group>
        <group>
            <task taskid=""/>
            <task taskid=""/>
        </group>
        <task taskid=""/>
    </group>
</layout>
</taskgraph>
```

## 6 Optimization Metadata

Envisaged use cases:

1. Users should be able to use the Grid transparently within Triana i.e. a user should need only to provide a Triana task graph and a maximum time for completion. Mechanisms plugged into Triana should then determine from this task graph the best way that the algorithm can be distributed onto the Grid given this user constraint.
2. Users should have the option to specify the distribution of tasks i.e. the user can specify a task/task graph to run at particular hosts.

Requirements:

1. Each unit within Triana will implement a monitoring API that will allow it to be quizzed about certain properties, for example, the number of flops required to complete its task (which is related to the input dimension), the amount of memory required and other information such as disk space requirements and network load. This information therefore can be used collectively to estimate the requirements from a complete algorithm (or part of an algorithm).
2. A benchmarking API for estimating the JavaFlops 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 Java). 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.
3. caching issues not considered yet.

4. network routing issues not considered yet.

## 7 Monitoring Metadata

Use Cases:

1. the user needs an estimation of how long it will take to run a unit, a group unit, entire network on a particular machine.
2. having started a job, the user wants to know the state of the execution of the network at any time.

Requirements:

1. network monitor - bandwidth
2. host monitoring (cpu, disk, memory, storage, jvaflops, Triana version, jdk, security)
3. job monitoring - 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 or 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)

## References

- [1] <http://zeus.enhydra.org/>
- [2] <http://www.csse.monash.edu.au/~bren/JSX/>
- [3] <http://www.krumel.com/jato/>
- [4] <http://java.sun.com/xml/jaxb/>