

# Request Description Language (RDL)

<b>Version 0.4</b>	<b>2004/01</b>	Gabriele Carcassi, Michael Haddox-Schatz, Andy Kowalski, Jerome Lauret
<b>Version 0.5</b>	<b>2004/11</b>	David Alexander, Paul Hamill, Jerome Lauret, Levente Hajdu, David Stampf
<b>Version 0.51</b>	<b>2005/04</b>	David Alexander, Paul Hamill, Jerome Lauret, Levente Hajdu

<b>1</b>	<b>INTRODUCTION .....</b>	<b>2</b>
1.1	DOCUMENT OUTLINE .....	2
<b>2</b>	<b>USE CASES .....</b>	<b>3</b>
2.1	DIFFERENT DATASETS .....	3
2.2	DIFFERENT DATA PLACEMENT STRATEGY .....	3
2.3	DIFFERENT ARCHITECTURES .....	4
2.4	DIFFERENT DATASET SPLITTING METHODS .....	4
2.5	AUTOMATIC APPLICATION DEPLOYMENT .....	5
2.6	DIFFERENT APPLICATION SETUP .....	5
2.7	ALLOW IMPLEMENTATION WITH NON GRID COMPONENTS .....	5
<b>3</b>	<b>REQUEST DESCRIPTION LANGUAGE (RDL) FRAMEWORK .....</b>	<b>6</b>
3.1	OVERVIEW .....	6
3.2	RDL ELEMENTS .....	6
3.3	XML .....	7
3.4	REQUEST SPECIFICATION .....	7
3.5	APPLICATION SPECIFICATION .....	10
3.6	TASK SPECIFICATION .....	12
3.7	DATASET SPECIFICATION .....	14
3.8	RESOURCE SPECIFICATIONS .....	17
3.9	WORKFLOW .....	18
3.10	OPEN ISSUES .....	19
3.11	EXTENSION MECHANISM .....	19
<b>4</b>	<b>COMMON ELEMENTS .....</b>	<b>19</b>
4.1	CSH AND OTHER SHELLS .....	20
4.2	ROOT .....	21
4.3	DATASETS .....	22
<b>5</b>	<b>STAR ELEMENTS .....</b>	<b>23</b>
5.1	CSH APPLICATION AND TASK .....	24
5.2	STAR DATASETS .....	25
5.3	ROOT4STAR .....	26
<b>6</b>	<b>APPENDIX .....</b>	<b>27</b>
6.1	THE XML SCHEMA .....	27

# 1 Introduction

This work aims to define a language for job submission in which a user describes a set of tasks he wants to be carried out in the closest way possible to his requirements. It is basically an attempt to model user requests.

In our vision, the user won't specify a list of jobs, with the details about machines and resources to be used, but rather a set of high level requests for tasks to be performed. The submission system would make a plan on how to use the middleware to satisfy the requests, and translate them into jobs and low-level resources requirements. Therefore, there is no one to one correspondence between a request and a job: a single request might result in thousands of jobs or no jobs at all. The term "*Job Description Language*" is inappropriate, and we propose to call this specification the Request Description Language (RDL) to stress that this specification deals with user requests, and not jobs running on the grid or on a batch system. Furthermore, we use the term "abstract" request to mean that a single call may include more than one individual requests and to denote that there may be an inter-dependency or relationship between the requests in one call to a service that processes such requests.

A user request will likely be also in term of experiment specific elements, such as detector components, software components and metadata. The RDL will therefore need to be extensible to allow different experiments to interface with their specific systems (file catalogs, databases, applications). From this, derives also the need for a request-processing system that will allow communication with experiment specific services that will translate the language of the experiment independent requests into experiment specific information.

Another main constraint of the RDL is to allow the same request to be satisfied with different architectures, as we will specify in the use cases. Different groups might need to do things in different ways. Or the same group might need to do things in different ways in different times.

We recognize that this work will be useful only if it is done in conjunction with a modular implementation that allows different groups to finalize their experiment specific parts, and integrate their software platforms. In that case, different experiment will be able to not only share the specifications, but also tools to achieve the request translation, which will be similar but not always exactly the same.

This work in part derives from the experience on the STAR scheduler project, which had similar aims but limited for the STAR experiment. The system has been first deployed on August 2002, and was very well received by the STAR user base. The specification started as a common project between STAR and JLab. Some concepts and idea were inspired by DIAL, with which we also hope to achieve a common specification. We hope that other groups will join the discussion, since the value of a specification is in a wide acceptance.

## 1.1 Document outline

The document consists of 3 parts:

- Use cases, where we collect a series of functionalities that the specification needs to contain or different contexts in which the specification needs to hold.

- RDL framework, in which we present the concepts and mechanism of the specification. It defines the elements of an abstract request, and how to extend the specification.
- Common elements, in which we use the extension mechanism presented in the previous section to define request components that are common to different HENP experiments.

## **2 Use cases**

Here we collect a list of features we might want to have. While not all of them are feasible in a short term implementation, the specification should be able to hold for those future changes.

### **2.1 *Different datasets***

Currently, different experiments specify the dataset for a job in different ways. Among these:

- File lists: a series of physical files stored on media, typically distributed file systems (NFS, AFS, ...) or mass storage systems (HPSS, ...)
- Logical file lists: a series of logical names that a file catalog is able to map to a physical file.
- Queries to file catalog: a series of conditions, typically experiment's metadata, that a file catalog is able to translate in a list of physical or logical files
- Queries to databases: a series of conditions which can be used to retrieve the data directly from the database
- Named datasets: a name that can be translated, by a catalog or other mean, to a series of files or database entries

A suitable RDL must allow the description of the dataset in terms of the type of dataset definition allowed by the experiment, and only by that. That is, if the software architect of an experiment designed his system to use only one or two of these paradigms, the user should only use those.

The RDL must allow for different kind of datasets, and the system reading the request must be able to enforce type choice.

### **2.2 *Different data placement strategy***

There are different strategies used to make the data available for the job. Typical examples are:

- Already placed: the data is placed on well-known disk resources either by the user or by the administrator. The job will expect to find the data in a specified location.
- Placed before the job execution: the job will wait for another job which places the data, typically in a cache directory.

- Placed when required: when the job tries to open the file, the system will trigger a data placement if the file is not already there.
- From database: the data will be taken from a database, which is installed and configured by the administrator.

The RDL shouldn't make any assumption in how the data is placed since different experiments are doing things in slightly different ways, and a single experiment might want to change its framework. Request will be translated to the correct scheme by the request submission system.

## **2.3 *Different architectures***

The target architecture of a request can be:

- Cluster: a group of computer managed by a batch system
- Condor flock: a group of computer managed by a batch system usually more heterogeneous than a cluster, and usually not sharing a network file system
- Super computer: a large parallel machine
- Single machine: an experiment might allow the software and data to be installed on a standalone machine, but one should still be able to use the same RDL.

An experiment should be able to either custom tailor the RDL to fit all the architectures, or to specify extra requirements in case an application needs them to run on a particular architecture.

## **2.4 *Different dataset splitting methods***

There are different schemes to split data into smaller sections and assign them to a computing resource.

- Static job splitting: in this case the entire dataset is split into manageable smaller datasets, one assigned to each job, and all the jobs are submitted to the batch system. Failure recovery means to detect which jobs failed and restart them if possible.
- Dynamic job splitting: all or just part of the dataset is split, and jobs are submitted. Depending on the results of those jobs, the rest of the dataset is split, and other jobs are sent. Failure recovery can include reassembling the dataset of the failed jobs with the non assigned dataset before re-splitting.
- Consumer/producer: a master program, called producer, assigned a small portion of the dataset to different slave programs, the consumers. Every time a consumer finishes analyzing a dataset, goes back to the producer to ask for another. Failure recovery is handled by the producer that knows at each moment which part of the dataset has been processed and which hasn't.

The RDL shouldn't assume any specific framework, but should allow specifying extra tags in case an application is highly dependent on the model on which it is implemented.

## ***2.5 Automatic application deployment***

We want the submission system to be able to understand whether a request was made for an application that is not setup at a particular site. The submission system should:

- Reject requests of application that are not available/allowed at one site
- Install the application software and its dependencies in case is possible and is reasonable, in which case there are two sub-cases:
  - Permanent setup: the application is installed also for future executions
  - One time setup: the application is installed and then removed when the request is satisfied

No information about the software installation should be present in the request, though. The request must have only the information needed by the submission system to determine which application to use and what service to contact to be able to install it. Other middleware should provide the necessary information for the application setup.

## ***2.6 Different application setup***

It can be reasonable to assume that one might want to configure the same application in different way at different clusters or at different sites. Reasons can be:

- Different optimizations for batch jobs vs. interactive jobs
- Different dataset splitting methods: ex static job splitting vs. producer consumer.
- Different site requirement for software

One could have these setups permanently, as necessities or as choices for different circumstances, or simply in a transition phase. For example, if one wants to move from static job splitting to producer consumer, should be able to do it gradually, with minimal impact on users.

## ***2.7 Allow implementation with non grid components***

Especially at the beginning, in which all the pieces of the grid are not fully operational, each experiment or site must be able to incorporate ad-hoc intermediate solutions. Examples are:

- File catalogs: many experiment have their own
- Data placement utilities
- Batch systems (directly instead of through Globus)

## 3 Request Description Language (RDL) framework

### 3.1 Overview

Given the broad requirements, the RDL can't and doesn't want to address all the specific situations and provide tags and options for all the cases: it would be impossible and it would generate a gigantic specification that no one could implement completely.

The strategy is to identify what is common to all the situations, define only that, and provide a framework for each experiment or site to define their application tags and options.

For those elements which are used across experiments (i.e. shell scripts, root) we will propose a standard, more detailed request definition. Applications that are already standardized and experiment/site independent, makes it possible to achieve a request definition that is experiment/site independent. For other jobs, such as Monte Carlo simulations or experiment specific data production, it is more unlikely (although not impossible) to have a standard emerge.

### 3.2 RDL Elements

While the setup can be quite different as described in the use cases, there are always three concepts that remain the same and that constitute an abstract request. They are:

- **Request:** A distinct piece of work to be performed that matches up an Application and a Task with optional additional references to Dataset and ResourceSpecifications descriptions. As we note below, Workflow elements will describe relationships between Request elements.
- **Application:** a user will need to say which program is going to run for his request. The application is considered not written by the user, and is not changed at a request by request basis. It's the part of software that is managed, has release numbers and is typically already resident at the target site. If it is not resident, its installation has to be supervised, or at least authorized, by the site/cluster administrator.
- **Task:** a user will need to say what to do with the program. In some cases specifying the application will be sufficient, but in most cases it won't. For example, if I specify `root` as an application, I will need to specify my `macro`; if I use `csh`, I will need to give a script. The task is the user defined part of the program; it can change at a request by request basis. It will be typically need to be made available from the submission site to the target site in some way.

By themselves, Request, Application, and Task constitute the minimum information needed to define the provenance of the result. That is, to define how the result was constructed. This distinction will be useful for integration with other work (i.e. GriPhyn/Chimera)<sup>1</sup>.

There is some additional information that might be needed to aid the submission system on how to satisfy the request

---

<sup>1</sup> This approach was inspired by David Adam's work on DIAL.

- **Dataset:** for some requests, a user will need to say on what data the program should run.
- **ResourceSpecifications:** optional processing parameters such as required memory and processing time as well as references to input and output datasets.
- **WorkFlow:** information about request dependencies and priorities

### 3.2.1 Open issues

Workflow is not yet in the schema.

## 3.3 XML

We chose XML as the basis for the description for the following reason:

- It is the defacto core language of the Web and Grid Services
- It allows us to have a hierarchical structure, in which concepts can be refined as we go deeper in the tree. This structure maps to the need to have very general concepts of the request, application, and task that can be first refined for commonalities and then, if needed, refined in slightly different ways by each experiment.
- There are a number of tools and libraries to help us edit, parse and generate XML.
- It's being used already in STAR user JDL with success, providing a proof of concept.

## 3.4 Request specification

### 3.4.1 Description & Examples

First, let's give an example of a simple `AbstractRequest`:

```
<AbstractRequest>
  <Request appRef="A1" taskRef="T1"/>

  <Application ID="A1" name="root"/>

  <RootTask ID="T1">
    <Macro>myMacro.C</Macro>
  </RootTask>

</AbstractRequest>
```

This request specifies that the application `root` will run the macro `myMacro.C`. As described before, every `AbstractRequest` is composed of at least three mandatory parts: the request, the application, and the task.

Next, let's examine a more complex AbstractRequest composed of several requests:

```
<AbstractRequest version="V0.5">
  <Request appRef="A1" taskRef="T1" datasetRef="DS1"/>
  <Request appRef="A2" taskRef="T1" datasetRef="DS1"/>

  <Application ID="A1" name="root4star" ver="SL04k"/>
  <Application ID="A2" name="root4star" ver="dev"/>
  <Application ID="A3" name="root" ver="4.00.04"/>

  <RootTask ID="T1">
    <Macro>example.C</Macro>
    <Arguments>10.0,true</Arguments>
  </RootTask>

  <!-- My Data sets -->
  <LogicalDataset ID="DS1">
    <FileCatalog>
      <Query>production=2g,type=muDST</Query>
    </FileCatalog>
  </LogicalDataset>
</AbstractRequest>
```

### 3.4.2

### 3.4.3 Schema

### 3.4.4

As shown here, an AbstractRequest can contain multiple request, task, and application elements. Optional parts are: the dataset(s), the resource specifications and the workflow. The schema for an AbstractRequest is:

```
<xsd:element name="AbstractRequest">
  <xsd:complexType>
    <xsd:choice maxOccurs="unbounded">
      <xsd:element ref="Request"/>
      <xsd:element ref="Task"/>
      <xsd:element ref="Application"/>
      <xsd:element ref="Dataset" minOccurs="0"/>
      <xsd:element ref="ResourceSpecifications" minOccurs="0"/>
      <xsd:element name="WorkFlow" minOccurs="0"/>
    </xsd:choice>
    <xsd:attribute name="version" use="optional" default="V0.5">
      <xsd:simpleType>
        <xsd:restriction base="xsd:NMTOKEN">
          <xsd:enumeration value="V0.5"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
  </xsd:complexType>
</xsd:element>
<xsd:element name="Request">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="RequestName" minOccurs="0"/>
      <xsd:element name="RequestDescription" minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```



```

<xsd:element name="OutputHandling" minOccurs="0">
  <xsd:complexType>
    <xsd:choice maxOccurs="unbounded">
      <xsd:element name="Copy">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="Source"/>
            <xsd:element name="Destination"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="Register"/>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>
</xsd:sequence>
<xsd:attribute name="ID" type="xsd:ID"/>
<xsd:attribute name="taskRef" type="xsd:IDREF" use="required"/>
<xsd:attribute name="appRef" type="xsd:IDREF" use="required"/>
<xsd:attribute name="datasetRef" type="xsd:IDREF" use="optional"/>
<xsd:attribute name="resourceRef" type="xsd:IDREF" use="optional"/>
</xsd:complexType>
</xsd:element>

```

The idea is that the internal syntax of a task can be task-type specific, as the syntax of the dataset can be dataset specific. For example a `scriptTask` will contain a brief script, while a `LogicalDataset` will contain a series of logical file names. Experiments that will have similar requirements will hopefully merge to a joint specification of datasets and tasks.

Custom application, task and dataset types will be created by extending the base types. This part of the specification aims only at defining the basic elements. In Section 3.10 we will describe how to extend the basic elements, and provide specification for those elements that can be common to different HENP experiments.

### 3.4.5 Possible commonalities

The request element is a high-level abstraction that is independent of the implementation framework and says that the user wants to run this application with this specific set of task instructions.

### 3.4.6 Open issues

None.

### 3.4.7 Implementation framework consideration

Each request will need to be mapped to a specific queuing system and be setup properly to be run within the grid site's resource brokering system.

### 3.4.8 Motivation

A request is meant to be a flexible way of matching up applications and specific tasks that are specified at a high-level or user's point-of-view.

### 3.4.9

## 3.5 *Application specification*

### 3.5.1 Description & Examples

The application represents some software installed and configured at a remote site that is able to execute a user defined task. An application will be able to execute only some types of tasks. The definition of which tasks are supported is up to the application.

The application basically represents a consistent environment around the task. It could be mapped to different executables at different sites: the important thing is that they execute the same tasks with the same results.

An application can be a standard piece of software (such as `root`, `bash/csh/tsh`, `java`), an experiment specific one (such as `root4star`) or different configurations of another application.

In this document we only limit ourselves to the specification of the application inside a request, but an application in our minds has to define a lot more than that. It has to define:

- Software resources: defines which software packages have to be installed.
- Task verification: whether the task description is complete and valid for the application.
- Task deployment: defines how all the files, needed by the task to run, will be made available at the (possible) remote resource. For example, it uses AFS, file transfer, a CVS checkout, ...
- Task installation: how the task will be installed at the (possible) remote computing resource. Does it need to be compiled, initialized, ...
- Data retrieval: how the task will access the data. It can be already present at the remote site, copied before the execution, copied when the job requests it, ...
- Dataset splitting method: how the entire dataset can be split for the different jobs.
- Job interface: how a particular job is going to receive the decisions of the submission system. For example, how it's going to access its specific sub-dataset.

In some cases, a choice in one of this area will require the user to specify more options. These options will reside in the task specification. That is, to be able to execute the task, the application will need to know some information to copy the task (the bundle) or install it (on which OS can it compile). The application will then support only tasks that will have that required section.

To make this concept clearer, let's make two examples.

A `csh` application may define that: `csh` has to be installed, that a task is a script, that to run the task will compose a command line `/bin/csh script`, that to deploy the task

it has to copy it, that the data is already found at the target site, that the dataset consists of files, that it can assign any number of files it wants to one job, that the job will receive the file names through environment variable.

On the other hand, a root application might define that: root has to be installed, that a task is a macro containing a bundle with all the files needed to run that macro, that to install the task it has to copy it and compile it, that the macro must have a function `int doEvent(Event &event)`, and finally that this function will be called once for each event in the dataset.

As you see, application in this context is much more than a mere installed package. Some examples are:

```
<Application ID="A1" name="csh" />
<Application ID="A2" name="root" />
<Application ID="A3" name="root4star" ver="2.0" />
```

### 3.5.2 Schema

```
<xsd:element name="Application" type="applicationType">
</xsd:element>

<xsd:complexType name="applicationType">
  <xsd:attribute name="ID" type="xsd:ID" use="required"/>
  <xsd:attribute name="type" type="xsd:string" use="optional"/>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
  <xsd:attribute name="ver" type="xsd:string" use="optional"/>
</xsd:complexType>
```

An application is basically a unique ID, a name reference, and a version number. The ID is used within the `AbstractRequest` to identify the application. The submission system can use the name to check whether that application is present, and dispatch it to the correct version, or return an error if the application is not present. In the future, the scheduler can also contact an application distribution and installation service to install the application if it's not present.

The application definition allows extension in the form of extended types.

### 3.5.3 Possible commonalities

For common applications, we can share both the syntax (i.e. the application name) and the application module within the implementation framework.

### 3.5.4 Open issues

One might want to define namespaces to avoid collision of names between applications in different experiments. Probably this will be linked to the schema validation and the use of XML namespaces. The idea is that one would define his request in the experiment namespace.


There have been some concerns for the `Application`: people would think only about the executable, and not about all the other part an application is going to define. Other suggestions include `Environment`, `Executor`, `Shell`, or `Framework`.

### 3.5.5 Implementation framework consideration

Each application name needs to be mapped to a specific program and executable. Within the implementation framework every experiment will need a way to translate experiment specific application to a more neutral request (system JDL). This can be done through plug-ins or through web services, the details of which are beyond the scope of this document.

### 3.5.6 Motivation

The concept of application as standalone is useful for:

- Automatic package installation. By specifying at high level which application the user is going to use, the scheduler will be more easily able to check whether the software is correctly installed and even install it.
-  Decoupling of the command line. By not submitting the actual command line, we allow different site (or different farms within a site) to execute the task in a different way. This allows us to use different schemes, for example, when running interactive jobs or when running batch jobs.
- It makes it easier to divert requests to different resources on an application basis. (i.e. from today, all the reconstruction is done at that location)

## 3.6 Task specification

### 3.6.1 Description & Examples

The task describes what the user is going to do with the application and the data. The task description is not strictly application dependent, since different applications can use the same task: different application versions or different applications that map to the same application repackaged to include different reconstruction algorithms.

Typically, the task is an XML section sent to the application module, and it's the application module that is going to interpret it. The task is what the application will run, so its pretty opaque to the submission system.

Some examples of tasks are: a root macro

```
<RootTask ID="T1">
  <Macro>myMacro.C</Macro>
  <Arguments>"arg1", 10.0</Arguments>
</RootTask>
```

a java application

```
<JavaTask ID="T2">
  <ClassName>gov.bnl.star.MyClass</ClassName>
  <Jar>scheduler.jar</Jar>
</JavaTask>
```

a csh script

```
<ScriptTask ID="T3">
  <Script>myscript.sh</Script>
  <Arguments>arg1 arg2</Arguments>
</ScriptTask>
```

### 3.6.2 Schema

```
<xsd:element name="Task" type="taskType">
</xsd:element>

<xsd:complexType name="taskType">
  <xsd:sequence>
    <xsd:element name="STDIN" type="xsd:anyURI" minOccurs="0"/>
    <xsd:element name="STDOUT" type="xsd:anyURI" minOccurs="0"/>
    <xsd:element name="STDERR" type="xsd:anyURI" minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="ID" type="xsd:ID" use="required"/>
  <xsd:attribute name="type" type="xsd:string" use="optional"/>
</xsd:complexType>
```

Given the premises, in general the task doesn't define much. The type of the task will be the element name, which will be changed according to the type. We will discuss more about this in the next session. The type allows to check whether a given application support a given type.

The work of coming up with a common RDL is essentially finding common task elements.

For example, different experiment might share the notion of a root task, or might share, among different tasks, the notion of a bundle (i.e. files that need to be copied to the target machine). In section 4 we provide examples of this idea.

### 3.6.3 Possible commonalities

There can be many different common defined tasks. Ideally, all the across experiment applications can have common tasks. For example, csh can be considered the application, and a task would be a command line or a script: this would map directly to what is done through a batch system now, and it's a good place to start. Experiment specific tasks can be translated into csh tasks. Tasks for common HEP application, such as root or paw, can also be provided.

We also aim to provide a sort of inheritance for task definition to provide common sections, such as output definition, that would map to common functionalities, such as moving the output file to a storage system, or adding it to a file catalog.

### 3.6.4 Implementation consideration

The task is closely associated to the application module. That is, the application module should be able to verify a task (see if it has all the information needed), install it (e.g. move the appropriate files, compile it, and make it available on a network file system), run it (that is, provide a command line to actually start the job).

The way that the task will receive the job submission decisions (such as, how the dataset was split and which part was assigned to the specific job) is also application

specific. For example, say a csh job will need to know on which file it will run (FILELIST) and which JOBID is assigned to the job. This will likely be passed through environment variables. An experiment framework, instead, could pass that information directly to a framework component, behind the back of the user, which would get the data from a collection or an iterator.

Therefore, one has to keep in mind the target architectures for the implementation when developing a common syntax for a task.

### 3.6.5 Motivation

Having the task separate from the application allows us to share its description among different application, and in the end different experiments. For example, STAR has a modified version of root, called root4star. That would qualify as a different application, but the form of the task, a root macro, might be the same.

## 3.7 Dataset specification

### 3.7.1 Description & Examples

The dataset is the data input for the analysis. It can be specified at a metadata level (e.g. all the events with certain physics characteristic), at a logical level (e.g. a logical file in a file catalog) or at a physical level (e.g. a physical file on a particular storage system).

The dataset definition should be connected to the work being done in the analysis working group on the definition of the dataset properties. Some examples are:

```
<PhysicalDataset>
  <PhysicalFileNames>
    <File>/data01/myExp/data/run20345/file01.root</File>
    <File>/data01/myExp/data/run20345/file02.root</File>
    <File>/data01/myExp/data/run20345/file03.root</File>
    <File>/data01/myExp/data/run20346/file01.root</File>
    <File>/data01/myExp/data/run20346/file03.root</File>
    <File>/data01/myExp/data/run20346/file04.root</File>
  </PhysicalFileNames>
</PhysicalDataset>

<LogicalDataset ID="DS1">
  <FileCatalog>
    <Query>production=2g,type=muDST</Query>
  </FileCatalog>
</LogicalDataset>
```

### 3.7.2 Schema

As for the task, the dataset doesn't define much per se. We will share further syntax by defining types of datasets that are common among different experiments.

```
<xsd:element name="Dataset" type="datasetType"/>
<xsd:complexType name="datasetType">
```

```

    <xsd:attribute name="ID" type="xsd:ID" use="required"/>
  </xsd:complexType>
  <xsd:complexType name="catalogType">
    <xsd:sequence>
      <xsd:element name="Name" type="xsd:anyURI" minOccurs="0"/>
      <xsd:element name="Query" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:element name="GenericDataset" substitutionGroup="Dataset">
    <xsd:complexType>
      <xsd:complexContent>
        <xsd:extension base="genericDatasetType">
          <xsd:attribute name="type" use="required">
            <xsd:simpleType>
              <xsd:restriction base="xsd:NMTOKEN">
                <xsd:enumeration value="Logical"/>
                <xsd:enumeration value="Physical"/>
                <xsd:enumeration value="Virtual"/>
                <xsd:enumeration value="Mixed"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:attribute>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>
  </xsd:element>
  <xsd:complexType name="genericDatasetType">
    <xsd:complexContent>
      <xsd:extension base="datasetType">
        <xsd:choice>
          <xsd:element name="Name" type="xsd:anyURI" maxOccurs="unbounded"/>
          <xsd:element name="Id" type="xsd:long" maxOccurs="unbounded"/>
          <xsd:element name="DataSetIDRef" type="xsd:IDREF"
            maxOccurs="unbounded"/>
          <xsd:element name="Range" maxOccurs="unbounded">
            <xsd:complexType>
              <xsd:sequence>
                <xsd:element name="Minimum" type="xsd:anySimpleType"/>
                <xsd:element name="Maximum" type="xsd:anySimpleType"/>
              </xsd:sequence>
            </xsd:complexType>
          </xsd:element>
          <xsd:element name="Catalog" type="catalogType" maxOccurs="unbounded"/>
          <xsd:element name="Token" type="xsd:anyType" maxOccurs="unbounded"/>
        </xsd:choice>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="Text" type="textType" substitutionGroup="Dataset"/>
  <xsd:complexType name="textType">
    <xsd:complexContent>
      <xsd:extension base="datasetType">
        <xsd:sequence>
          <xsd:element name="Line" type="xsd:string" maxOccurs="unbounded"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="Environment" type="environmentType"
    substitutionGroup="Dataset"/>
  <xsd:complexType name="environmentType">
    <xsd:complexContent>
      <xsd:extension base="datasetType">

```

```

<xsd:sequence>
  <xsd:element name="EnvironmentVariable" maxOccurs="unbounded">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Name" type="xsd:string"/>
        <xsd:element name="Value" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:sequence>
</xsd:extension>
</xsd:complexContent>
</xsd:complexType>

```

### 3.7.3 Possible commonalities

Common types of dataset that can be shared are:

- Physical file name dataset: a dataset made of files located on a disk, either on the network or on local disk
- Logical file name dataset: a dataset made of logical file names connected to a particular file catalog
- File catalog metadata query: a dataset made of the result of a query to a file catalog. The syntax of the query would be file catalog dependent, and a module within the implementation is needed to transform the query in a list of physical/logical files. The specification of a common query language for file catalogs is beyond the scope of this work.

### 3.7.4 Open issues

Some data, such as calibration data, can be seen as part of the dataset, as a parameter of the task, or as a defined property of the application. It is unclear where to put it, and maybe the RDL should provide the flexibility for an experiment to decide in the specific case.

### 3.7.5 Implementation consideration

Dataset definition is intimately connected with the concept of dataset and request splitting. Request splitting is the crucial point of all the scheduling process: the dataset can potentially be divided according to the different resources used by the job (i.e. files, machines, CPU time, hard disk space for output), user characteristic (i.e. which user, what role) and site specific information (i.e. this site has a short queue, local batch system practices).

Constraint on the request splitting can be put by the type of dataset, but also by the specific task and application. Since some application will be experiment specific, an implementation need to remain flexible on how the dataset is handled, and provide a framework to insert custom dataset splitters.



### 3.7.6 Motivation

The data input is an orthogonal section to the application and task, and different experiment might have different ways to store, access and specify the dataset.

## 3.8 Resource Specifications

### 3.8.1 Description & Descriptions

The resource specification gives processing parameters such as required memory and processing time. This information allows the scheduler to estimate the resources that will be required to perform the request, and queue it accordingly. Since resource types and unit will vary between tasks, the RDL attempts to define a general schema for communicating minimum, maximum, and size of resources, without specific reference to a particular unit.

An example of the ResourceSpecifications is:

```
<ResourceSpecifications ID="R1">
  <MemoryPerProcess>
    <AverageSizeOfUnit>1.0</AverageSizeOfUnit>  <!-- 1 MB -->
    <MaximumUnitsEstimated>20</MaximumUnitsEstimated>  <!-- 20 MB -->
  </MemoryPerProcess>
  <Time>600</Time>
</ResourceSpecifications>
```

### 3.8.2 Schema

```
<xsd:element name="ResourceSpecifications">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="MemoryPerProcess" minOccurs="0"/>
      <xsd:element ref="InputResource" minOccurs="0"/>
      <xsd:element ref="OutputResource" minOccurs="0"/>
      <xsd:element name="Time"/>
      <xsd:any namespace="##any" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="ID" type="xsd:ID" use="required"/>
  </xsd:complexType>
</xsd:element>
```

The ResourceSpecifications element contains a mandatory Time element, optional MemoryPerProcess, InputResource, InputFile, InputTracks, InputEvents, and OutputResource, outputFiles, OutputTracks, and OutputEvents elements are defined below:

```
<xsd:element name="MemoryPerProcess" type="resourceType"/>
<xsd:element name="InputResource" type="resourceType"/>
```

```

<xsd:element name="InputFiles" type="resourceType"
  substitutionGroup="InputResource"/>
<xsd:element name="InputTracks" type="resourceType"
  substitutionGroup="InputResource"/>
<xsd:element name="InputEvents" type="resourceType"
  substitutionGroup="InputResource"/>
<xsd:element name="OutputResource" type="resourceType"/>
<xsd:element name="OutputFiles" type="resourceType"
  substitutionGroup="OutputResource"/>
<xsd:element name="OutputTracks" type="resourceType"
  substitutionGroup="OutputResource"/>
<xsd:element name="OutputEvents" type="resourceType"
  substitutionGroup="OutputResource"/>

```

These elements are of type `resourceType`, which defines a resource as something that may have a size, minimum, maximum, or a rate of usage over time, independent of the units in which it is actually measured:

```

<xsd:complexType name="resourceType">
  <xsd:sequence>
    <xsd:element name="MininumUnitsRequired" type="xsd:long"
      minOccurs="0"/>
    <xsd:element name="MaximumUnitsEstimated" type="xsd:long"
      minOccurs="0"/>
    <xsd:choice minOccurs="0">
      <xsd:element name="AverageSizeOfUnit" type="xsd:float"/>
      <xsd:element name="UnitsPerSeconds" type="xsd:float"/>
    </xsd:choice>
  </xsd:sequence>
</xsd:complexType>

```

Thus, an element of type `resourceType` always contains an `AverageSizeOfUnit`. It may contain either a `UnitsPerSeconds` element, or a sequence of `MinimumUnitsRequired` and `MaximumUnitsEstimated`.

Other resource types that are defined for use in `ResourceSpecifications` include `InputFiles`, `InputTracks`, `InputEvents`, `OutputFiles`, `OutputTracks`, `OutputEvents`. These are also of type `resourceType`.

## 3.9 WorkFlow

### 3.9.1 Description & Examples

The work flow provides information to help the scheduler manage dependencies between multiple requests. Its schema is not yet defined, but the following pseudo-schema shows the type of information that may be represented.

```

MergeRequests { id=[xs:ID] }
RequestID { idref=[xs:IDref] }
....
ChooseBetweenRequests { id=[xs:ID] }
  RequestID { idref=[xs:IDref] }
SetupRequestDependancy { id=[xs:ID] }
  RequestToBeDoneFirstID { idref=[xs:IDref] }

```

### 3.10 Open issues

The Workflow tags have not yet been completed.

### 3.11 Extension mechanism

As we said, the basic specification provides only fundamental concepts and nothing more. The rest of the specification builds on these to provide common specification for common applications, tasks and datasets. *Notice that if two experiments do not share any applications, or tasks, and do not even agree on how a dataset should be defined, there is no chance that they will share a common RDL.* A common RDL, or common parts of it, can exist only if the intent of different groups is similar. We believe that such similarities exist between the different HENP experiments, some of which on all the PPDG members.

The extension mechanism is based on the extension scheme in XML: it allows us to substitute an element with another element of a derived type. For example, we want to specify the scriptTaskType as a derivation of the taskType, and we want to allow the scriptTask to replace task. We add to the specification:

```
<xsd:element name="scriptTask" type="scriptTaskType"
  substitutionGroup="task"/>
<xsd:complexType name="scriptTaskType">
  <xsd:complexContent>
    <xsd:extension base="taskType">
      <xsd:sequence>
        <xsd:element name="script" type="xsd:string"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

With the same mechanism, we could specify a specialScriptTaskType that derives from the scriptTask, and so on.

#### 3.11.1 Open issue

We would like to declare difference pieces of different schema definitions without modifying the original schema file. Is this possible?

## 4 Common elements

In this section we go into the details of which part of a request can be standardized for broad collaboration. What we envision is that these commonalities will be defined as a per need basis, and might even evolve over time. Subsequent specification, made also by other group, should be allowed to add or extend different elements. This section also function as an example on how this work can be done.

## 4.1 CSH and other shells

A good starting point to illustrate how to define an application and a task is to define a csh application and a csh task. It is a good starting point for the following reason:

- It's a standard application, and it's probably useful to have a RDL ready for it
- It's something well understood
- It's close to what one specifies to a batch system

The application, therefore, is csh itself, and the task is other a script, or a file containing the script. For example, we would expect something like:

```
<Application ID="A1" name="csh"/>
<ScriptTask ID="T1">
  <Script>
    echo Here is the time:
    time
  </Script>
</ScriptTask>
```

or

```
<Application ID="A1" name="csh"/>
<ScriptTask ID="T1">
  <Script>myscript.csh</Script>
</ScriptTask>
```

The application follows directly the general syntax, but the task does not. In fact, we need a script element that tells the script to execute, or the file that contains the script. We need a schema modification to accept these new components.

```
<xsd:complexType name="scriptTaskType">
  <xsd:complexContent>
    <xsd:extension base="taskType">
      <xsd:sequence>
        <xsd:element name="Script">
          <xsd:complexType>
            <xsd:simpleContent>
              <xsd:extension base="xsd:string"/>
            </xsd:simpleContent>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="Arguments" type="xsd:string" minOccurs="0"
          maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:element name="ScriptTask"
  type="scriptTaskType" substitutionGroup="Task"/>
```

This allows us to substitute a task element, which was declared as taskType, with a scriptTask element, of type scriptTaskType, which extends taskType. In this way, the parser is able to validate the XML request, and we are able to add new task, application and dataset types. In much the same way, we can extend the scriptTaskType, and we can allow to specify the parts of the request that are common, while leaving to each experiment the freedom to add their experiment specifics tags.

[TODO: still have to understand how to chain XML schema definitions]

Also notice that the same definition of a script task applies for other script interpreters: bash, tsch, perl, python... The only change needed is the name of the application

The specification, though, does not end here. The job submission system will have to make some decisions: namely, will have to get the entire dataset, and split it into multiple chunks and assign it to each job. Within the script one will need to know which particular dataset is assigned to the particular instance of the script to be able to do something with it. We need a mechanism to do that, and this mechanism is specific to the application.

Within a shell script, it comes naturally to use environment variables. Let's assume that the csh application will accept datasets that consists of files. We can define the following:

- \$JOIBID - A unique number for the job
- \$INPUTFILECOUNT - The number of input files
- \$INPUTFILExx - The input filename

Within the script, one can use these variables to get which data needs to be analyzed.

The application needs also to define how the inputs and outputs will be transferred, and also which inputs and outputs. As we said in the use cases, this can be done in a number of ways. The csh application will need the script itself, which is already specified, but it might also need to redirect the standard streams. For example:

```
<scriptTask>
...

<STDIN>myfile.in</STDIN>
<STDOUT>myfile.out</STDOUT>
<STDERR>myfile.err</STDERR>
</scriptTask>
```

## 4.2 ROOT

Root is a common application among HENP experiments, so it makes sense to have a common specification for it. Here is an example:

```
<Application ID="A1" name="root"/>

<RootTask ID="T1">
  <Macro>myMacro.C</Macro>
  <Arguments>234, myFileList</Arguments>
```

```
</RootTask>
```

or:

```
<Application ID="A1" name="root"/>

<RootTask ID="T1">
  <Macro>myMacro.C(234, "myFileList")</Macro>
</RootTask>
```

## 4.3 Datasets

Another common area is the dataset definition. There are different kinds of dataset that can be shared. For example:

- Physical file dataset
- Logical file dataset

These datasets have types that are extended from the base Dataset type, as shown below.

```
<PhysicalDataset>
  <PhysicalFileNames>
    <File>/data01/myExp/data/run20345/file01.root</File>
    <File>/data01/myExp/data/run20345/file02.root</File>
    <File>/data01/myExp/data/run20345/file03.root</File>
    <File>/data01/myExp/data/run20346/file01.root</File>
    <File>/data01/myExp/data/run20346/file03.root</File>
    <File>/data01/myExp/data/run20346/file04.root</File>
  </PhysicalFileNames>
</PhysicalDataset>

<LogicalDataset ID="DS1">
  <FileCatalog>
    <Query>production=2g,type=muDST</Query>
  </FileCatalog>
</LogicalDataset>
```

Here is an XML schema fragment specifying the PhysicalDataset type, an extension of Dataset:

```
<xsd:element name="PhysicalDataset" type="physicalDatasetType"
  substitutionGroup="Dataset"/>

<xsd:complexType name="physicalDatasetType">
  <xsd:complexContent>
    <xsd:extension base="datasetType">
      <xsd:choice maxOccurs="unbounded">
        <xsd:element name="ReplicaCatalog" type="catalogType"/>
        <xsd:element name="PhysicalFileNames" type="filelistType"/>
      </xsd:choice>
    </xsd:extension>
  </xsd:complexContent>
```

```
</xsd:complexType>
```

First, it tells that a dataset can be described by a `PhysicalDataset` of type `physicalDatasetType`. Then it describes the content of a `PhysicalDataset`, which consists of either a `ReplicaCatalog` element of type `catalogType`, or a `PhysicalFileNameselement` of type `filelistType`. The types `catalogType` and `filelistType` are defined separately:

```
<xsd:complexType name="catalogType">
  <xsd:sequence>
    <xsd:element name="Type" type="xsd:string" minOccurs="0"/>
    <xsd:element name="Name" type="xsd:anyURI" minOccurs="0"/>
    <xsd:element name="Query" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
```

Elements of the type `catalogType` consist of a `Query` element, which generates a STAR catalog query, and optional `Type` and `Name` elements. Elements of the type `filelistType` consist of a list of `File` elements.

```
<xsd:complexType name="filelistType">
  <xsd:sequence>
    <xsd:element name="File" type="xsd:anyURI" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
```

Since we defined the `catalogType` and `filelistType` as separate types, we can use the same definitions in different context. For example, a `LogicalDataset` consists of either a list of files or a catalog query. The schema fragment to specify the `logicalDataset` type would be:

```
<xsd:element name="LogicalDataset" type="logicalDatasetType"
  substitutionGroup="Dataset"/>

<xsd:complexType name="logicalDatasetType">
  <xsd:complexContent>
    <xsd:extension base="datasetType">
      <xsd:choice maxOccurs="unbounded">
        <xsd:element name="FileCatalog" type="catalogType"/>
        <xsd:element name="LogicalFileNames" type="filelistType"/>
      </xsd:choice>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

[TODO a real specification agreed over different experiment]

## 5 STAR elements

The STAR scheduler already comes with the concept of a request that maps to multiple jobs. The request itself, though, is not structured according the proposed framework. Here we describe a possible way to reorganize the STAR scheduler request

attributes according to the proposed scheme. For a complete description of the current STAR specification, refer to the scheduler manual at:

<http://www.star.bnl.gov/STAR/comp/Grid/scheduler/manual.htm>.

This constitutes the minimum required from STAR to be able to submit jobs, and can be implemented in a short time by replacing the front end of the scheduler itself.

[TODO: tag arrangement is completely open to discussion]

## 5.1 CSH application and task

In STAR there are basically two sets of application that we will be addressing: CSH and Root. Let's start with CSH, since the STAR scheduler users specify a small CSH script in their request.

There are a couple of assumptions we will take, which are characteristic of the STAR setup:

- At each site where the jobs will be submitted, the STAR software infrastructure is present, and accessible via PATH in the same way.
- The file:/ URLs in a request are relative to the site where the request is sent.

The assumptions go together in the sense that they are symptoms of the same cause: we don't have yet a reliable way to transfer bundles. Once suitable GRID middleware is in place, we can review the file specification and add the notion of a bundle to the request.

A request with all the parameters might look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<AbstractRequest>
  <Request appRef="A1" taskRef="T1" datasetRef="DS1" resourceRef="R1"/>

  <Application ID="A1" name="csh"/>

  <ScriptTask ID="T1">
    <Script>myScript.csh</Script>
    <Arguments>10.0, true</Arguments>
    <STDIN>file:/star/u/carcassi/scheduler/out/$JOBID.in</STDIN>
    <STDOUT>file:/star/u/carcassi/scheduler/out/$JOBID.out"</STDOUT>
    <STDERR>file:/star/u/carcassi/scheduler/out/$JOBID.err"</STDERR>
  </ScriptTask>

  <LogicalDataset ID="DS1">
    <FileCatalog>
      <Query>production=2g,type=muDST</Query>
    </FileCatalog>
  </LogicalDataset>

  <ResourceSpecifications ID="R1">
    <MemoryPerProcess>
      <AverageSizeOfUnit>1.0</AverageSizeOfUnit> <!-- 1 MB -->
      <MaximumUnitsEstimated>5</MaximumUnitsEstimated> <!-- 5 MB -->
    </MemoryPerProcess>
  </ResourceSpecifications>
</AbstractRequest>
```



```

    <Time>600</Time>
  </ResourceSpecifications>
</AbstractRequest>

```

The elements of this request:

Request	The job to be executed, containing an application, a task, a dataset, and a resource specification, all identified by ID.
Application	The application to run, 'csh'.
ScriptTask	The script 'myScript.csh' will be run with the provided arguments. The standard streams are redirected to the URIs given by the STDIN, STDOUT, and STDERR elements. The environment variable \$JOBID will be replaced by the actual job ID assigned by the scheduler.
LogicalDataset	Specifies the input data to the application. Here, a catalog query generates the input data.
ResourceSpecifications	Time (600 sec) and memory (20 MB) resources the request is expected to consume.

## 5.2 STAR Datasets

The current scheduler interface allows the input to be a mix of files, filelists and queries. It is likely that this won't be needed in the future, and it will be better to have 3 different separate kinds of dataset: files, filelists and catalog queries.

So, for STAR, we will need 4 kinds of datasets: one describing a list of files, one describing a filelist (which is a separate file containing a list of files), one describing a series of queries to the metadata catalog and one allowing a mix of the three, for compatibility to our current specification.

(TODO: Bring this section up to date with the current RDL schema.)

### 5.2.1 fileListDataset

The first two can be combined in a single entity that allows two different specifications

```

<fileListDataset>
  <file
URL="file:/star/data15/reco/productionCentral/FullField/P02ge/2001/322/st_physi
cs_2322006_raw_0015.MuDst.root"/>
  <file
URL="file:/star/data15/reco/productionCentral/FullField/P02ge/2001/322/st_physi
cs_2322006_raw_0016.MuDst.root"/>
  <file
URL="file:/star/data15/reco/productionCentral/FullField/P02ge/2001/322/st_physi
cs_2322006_raw_0017.MuDst.root"/>
</fileListDataset>

<fileListDataset URL=" file:/star/u/user/username/filelists/mylist.list" />

```

## 5.2.2 catalogDataset

The catalog dataset allows retrieving a list of files based on the metadata.

```
<catalogDataset orderBy="production" >
  <catalog
    URL="catalog:star.bnl.gov?production=P02gd,filetype=daq_reco_mudst,filetype=MC_
    reco_MuDst" nFiles="all" />
  <catalog
    URL="catalog:star.bnl.gov?production=P03gd,filetype=daq_reco_mudst,filetype=MC_
    reco_MuDst" nFiles="all" />
</catalogDataset>
```

The syntax of the catalog URL is specific to the catalog, in this case the STAR catalog.

## 5.2.3 starDataset

The starDataset will allow a mix of the three elements.

```
<starDataset>
  <catalog
    URL="catalog:star.bnl.gov?collision=&collision;;trgsetupname=&trigger;;filetype
    =MC_reco_MuDst,storage=NFS" nFiles="all" />
  <file
    URL="file:/star/data15/reco/productionCentral/FullField/P02ge/2001/*/*.MuDst.ro
    ot"/>
  <filelist URL="filelist:/star/u/user/username/filelists/mylist.list"/>
</starDataset>
```

## 5.3 Root4star

The requests for Root would use a RootTask definition, and a new application name:

```
<AbstractRequest>
  <Request appRef="A1" taskRef="T1"/>

  <Application ID="A1" name="root4star" ver="dev"/>

  <RootTask ID="T1">
    <Macro>myMacro.C</Macro>
    <Arguments>"$FILELIST", "$SCRATCH/myAnalysis_$JOBID.root", 21
  </Arguments>
    <STDIN>file:/star/u/carcassi/scheduler/out/$JOBID.in</STDIN>
    <STDOUT>file:/star/u/carcassi/scheduler/out/$JOBID.out</STDOUT>
    <STDERR>file:/star/u/carcassi/scheduler/out/$JOBID.err</STDERR>
  </RootTask>
</AbstractRequest>
```

The only difference is that the script element is substituted by the macro element, which includes a macro call. Also, the fileListSyntax attribute since becomes a matter of the root4star application to specify.

## 6 Appendix

### 6.1 The XML schema

This is the XML schema that includes all the schema fragments defined during the document.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="AbstractRequest">
    <xsd:complexType>
      <xsd:choice maxOccurs="unbounded">
        <xsd:element ref="Request"/>
        <xsd:element ref="Task"/>
        <xsd:element ref="Application"/>
        <xsd:element ref="Dataset" minOccurs="0"/>
        <xsd:element ref="ResourceSpecifications" minOccurs="0"/>
        <xsd:element name="WorkFlow" minOccurs="0"/>
      </xsd:choice>
      <xsd:attribute name="version" use="optional" default="V0.50">
        <xsd:simpleType>
          <xsd:restriction base="xsd:NMTOKEN">
            <xsd:enumeration value="V0.50"/>
            <xsd:enumeration value="V0.5"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="Request">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="RequestName" minOccurs="0"/>
        <xsd:element name="RequestDescription" minOccurs="0"/>
        <xsd:element name="OutputHandling" minOccurs="0">
          <xsd:complexType>
            <xsd:choice maxOccurs="unbounded">
              <xsd:element name="Copy">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element name="Source"/>
                    <xsd:element name="Destination"/>
                  </xsd:sequence>
                </xsd:complexType>
              </xsd:element>
              <xsd:element name="Register"/>
            </xsd:choice>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="ID" type="xsd:ID"/>
      <xsd:attribute name="taskRef" type="xsd:IDREF" use="required"/>
      <xsd:attribute name="appRef" type="xsd:IDREF" use="required"/>
      <xsd:attribute name="datasetRef" type="xsd:IDREF" use="optional"/>
      <xsd:attribute name="resourceRef" type="xsd:IDREF" use="optional"/>
    </xsd:complexType>
  </xsd:element>


```

```

<xsd:element name="Application" type="applicationType"/>
<xsd:complexType name="applicationType">
  <xsd:attribute name="ID" type="xsd:ID" use="required"/>
  <xsd:attribute name="name" type="xsd:string" use="optional"/>
  <xsd:attribute name="ver" type="xsd:string" use="optional"/>
</xsd:complexType>

<xsd:element name="Task" type="taskType"/>
<xsd:complexType name="argumentType">
  <xsd:simpleContent>
    <xsd:extension base="xsd:string">
      <xsd:attribute name="type" type="xsd:anySimpleType" use="optional"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
<xsd:complexType name="taskType">
  <xsd:sequence>
    <xsd:element name="STDIN" type="xsd:anyURI" minOccurs="0"/>
    <xsd:element name="STDOUT" type="xsd:anyURI" minOccurs="0"/>
    <xsd:element name="STDERR" type="xsd:anyURI" minOccurs="0"/>
    <xsd:element name="TaskArguments" type="xsd:string" minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="ID" type="xsd:ID" use="required"/>
  <xsd:attribute name="type" type="xsd:string" use="optional"/>
</xsd:complexType>
<xsd:complexType name="rootTaskType">
  <xsd:complexContent>
    <xsd:extension base="taskType">
      <xsd:sequence>
        <xsd:element name="Macro">
          <xsd:complexType>
            <xsd:simpleContent>
              <xsd:extension base="xsd:string"/>
            </xsd:simpleContent>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="Argument" type="argumentType" minOccurs="0"
          maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="RootTask" type="rootTaskType" substitutionGroup="Task"/>
<xsd:complexType name="scriptTaskType">
  <xsd:complexContent>
    <xsd:extension base="taskType">
      <xsd:sequence>
        <xsd:choice>
          <xsd:element name="EmbeddedScript" type="xsd:string"/>
          <xsd:element name="ScriptName">
            <xsd:complexType>
              <xsd:simpleContent>
                <xsd:extension base="xsd:string"/>
              </xsd:simpleContent>
            </xsd:complexType>
          </xsd:element>
        </xsd:choice>
        <xsd:element name="Argument" type="argumentType" minOccurs="0"
          maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

```

```

        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
<xsd:element name="ScriptTask" type="scriptTaskType"
    substitutionGroup="Task"/>
<xsd:complexType name="javaTaskType">
    <xsd:complexContent>
        <xsd:extension base="taskType">
            <xsd:sequence>
                <xsd:element name="ClassPath" type="xsd:anyURI" minOccurs="0"/>
                <xsd:choice>
                    <xsd:element name="ClassName" type="xsd:anyURI"/>
                    <xsd:element name="MainJar" type="xsd:anyURI"/>
                </xsd:choice>
                <xsd:element name="Argument" type="argumentType" minOccurs="0"
                    maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
<xsd:element name="JavaTask" type="javaTaskType" substitutionGroup="Task"/>

<xsd:element name="Dataset" type="datasetType"/>
<xsd:complexType name="datasetType">
    <xsd:attribute name="ID" type="xsd:ID" use="required"/>
</xsd:complexType>
<xsd:complexType name="catalogType">
    <xsd:sequence>
        <xsd:element name="Name" type="xsd:anyURI" minOccurs="0"/>
        <xsd:element name="Query" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:element name="GenericDataset" substitutionGroup="Dataset">
    <xsd:complexType>
        <xsd:complexContent>
            <xsd:extension base="genericDatasetType">
                <xsd:attribute name="type" use="required">
                    <xsd:simpleType>
                        <xsd:restriction base="xsd:NMTOKEN">
                            <xsd:enumeration value="Logical"/>
                            <xsd:enumeration value="Physical"/>
                            <xsd:enumeration value="Virtual"/>
                            <xsd:enumeration value="Mixed"/>
                        </xsd:restriction>
                    </xsd:simpleType>
                </xsd:attribute>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
</xsd:element>
<xsd:complexType name="genericDatasetType">
    <xsd:complexContent>
        <xsd:extension base="datasetType">
            <xsd:choice>
                <xsd:element name="Name" type="xsd:anyURI" maxOccurs="unbounded"/>
                <xsd:element name="Id" type="xsd:long" maxOccurs="unbounded"/>
                <xsd:element name="DataSetIDRef" type="xsd:IDREF"
                    maxOccurs="unbounded"/>
                <xsd:element name="Range" maxOccurs="unbounded">
                    <xsd:complexType>
                        <xsd:sequence>
                            <xsd:element name="Minimum" type="xsd:anySimpleType"/>

```

```

        <xsd:element name="Maximum" type="xsd:anySimpleType"/>
    </xsd:sequence>
</xsd:complexType>
</xsd:element>
<xsd:element name="Catalog" type="catalogType"
    maxOccurs="unbounded"/>
    <xsd:element name="Token" type="xsd:anyType" maxOccurs="unbounded"/>
</xsd:choice>
</xsd:extension>
</xsd:complexContent>
</xsd:complexType>
<xsd:element name="Text" type="textType" substitutionGroup="Dataset"/>
<xsd:complexType name="textType">
    <xsd:complexContent>
        <xsd:extension base="datasetType">
            <xsd:sequence>
                <xsd:element name="Line" type="xsd:string" maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
<xsd:element name="Environment" type="environmentType"
    substitutionGroup="Dataset"/>
<xsd:complexType name="environmentType">
    <xsd:complexContent>
        <xsd:extension base="datasetType">
            <xsd:sequence>
                <xsd:element name="EnvironmentVariable" maxOccurs="unbounded">
                    <xsd:complexType>
                        <xsd:sequence>
                            <xsd:element name="Name" type="xsd:string"/>
                            <xsd:element name="Value" type="xsd:string"/>
                        </xsd:sequence>
                    </xsd:complexType>
                </xsd:element>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>

<xsd:element name="ResourceSpecifications">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="MemoryPerProcess" minOccurs="0"/>
            <xsd:element ref="InputResource" minOccurs="0"/>
            <xsd:element ref="OutputResource" minOccurs="0"/>
            <xsd:element name="Time"/>
            <xsd:any namespace="##any" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="ID" type="xsd:ID" use="required"/>
    </xsd:complexType>
</xsd:element>
<xsd:complexType name="resourceType">
    <xsd:sequence>
        <xsd:element name="MininumUnitsRequired" type="xsd:long"
minOccurs="0"/>
        <xsd:element name="MaximumUnitsEstimated" type="xsd:long"
minOccurs="0"/>
        <xsd:choice minOccurs="0">
            <xsd:element name="AverageSizeOfUnit" type="xsd:float"/>
            <xsd:element name="UnitsPerSeconds" type="xsd:float"/>
        </xsd:choice>
    </xsd:sequence>
</xsd:complexType>

```

```
        </xsd:sequence>
    </xsd:complexType>
    <xsd:element name="MemoryPerProcess" type="resourceType"/>
    <xsd:element name="InputResource" type="resourceType"/>
    <xsd:element name="InputFiles" type="resourceType"
        substitutionGroup="InputResource"/>
    <xsd:element name="InputTracks" type="resourceType"
        substitutionGroup="InputResource"/>
    <xsd:element name="InputEvents" type="resourceType"
        substitutionGroup="InputResource"/>
    <xsd:element name="OutputResource" type="resourceType"/>
    <xsd:element name="OutputFiles" type="resourceType"
        substitutionGroup="OutputResource"/>
    <xsd:element name="OutputTracks" type="resourceType"
        substitutionGroup="OutputResource"/>
    <xsd:element name="OutputEvents" type="resourceType"
        substitutionGroup="OutputResource"/>
</xsd:schema>
```