

AJDL: Analysis Job Description Language

Version 0.30
David Adams
March 7, 2004

Introduction

One of the most difficult and most important challenges for grid computing is support of end-user analysis, i.e. enabling ordinary users to make use of distributed computing resources to produce and manipulate large distributed data samples. There are many types of data, many applications and many users and we cannot expect to satisfy them all with a single system. However there is pattern of interaction with the grid that that we expect to be broadly applicable: a user defines a transformation and an input dataset and uses the grid to apply the former to the latter to produce another dataset. The new dataset is examined locally (i.e. off the grid), the application or dataset are modified and the process is repeated until a satisfactory result is obtained. The new dataset may then be used input to a subsequent processing step.

Components

It is useful to specify the transformation with two components: an application and the task that carries the user's configuration of that application. This configuration may include input parameters, configuration files and even code to be compiled and dynamically linked. The application typically defines an entry point into a large software system that must be installed on the machines where the processing takes place.

This is a job-based model where the user's initial request specifying application, task and dataset defines a high-level job. Processing is typically distributed by splitting this job into a collection of sub-jobs, carrying them out, and then merging their results to form the output dataset. This splitting may be recursive with the sub-jobs split into sub-sub-jobs, etc. Users should be able to monitor the progress of jobs (and sub-jobs) and be allowed some control, e.g. to kill or suspend/resume processing.

The processing of a high level job is a complicated task. In addition to splitting and merging, one must be concerned with staging input data, placing output data and monitoring sub-jobs to handle failures, slow-running jobs, etc. Jobs may be subject to constraints such as maximum allowed resource usage or response time. Processing should be coordinated with that of other jobs to ensure that a high priority job is not blocked by a previously submitted low priority job.

For these reasons we introduce a separate component that has the responsibility to carry out this processing. We call this the scheduler or, if it has a service interface the analysis service.

We anticipate the need for some job configuration information such as the user's role, resource budget, maximum response time, preferred location for output data, etc. We introduce an additional component, JobConfiguration, to carry this data.

Putting these all together, we have a list of the types of components that make up the AJDL (Analysis Job Description Language). The list is as follows:

- Application – Specifies the software used to do the processing
- Task – User configuration of the application
- Dataset – Specifies a collection of data
- JobConfiguration – Configuration for a job instance
- Job – Particular instance of application, task and dataset producing an output dataset
- Scheduler – Carries out the processing of jobs

Users will need to store, retrieve and select these objects. Selection may be based on internal characteristics or assigned metadata. To facilitate these operations, we also outline some catalog services.

Nature of the components

For most or all of the above types, we do not expect to be able to come up with a single definition that will be applicable across all realms. Instead these realms will introduce appropriate subtypes. For example ATLAS might introduce a POOL event collection dataset and an athena-based reconstruction application. There may be compound, globus, condor and LSF job types.

An important goal of AJDL is to define generic interfaces to the above components so that associated tools and services may be implemented generically so that particular tool or service can be used with components defined in different realms. Most importantly, it should be possible to implement a generic scheduler that can be used with a broad range of applications and datasets. Similarly it should be possible to create a job monitor that works with many different types of jobs.

It will often be useful to allow for partial specialization of the basic types to make it possible to implement generic tools and services that work for that subtype. For example, an event dataset (i.e. one made up of events than can be processed independently) is very useful for particle physics and many other areas.

It is obvious that most concrete types must have persistent representations and it is natural to express them in a data language such as XML. For definiteness we refer to XML in the following but the reader may substitute his or her own choice of data language.

However, *it is too restrictive to assume that a single schema for a generic type can be used with all subtypes to define all generic properties*. This may arise from different conventions in different realms, e.g. 32-bit identifiers in one and 128-bit in another. More importantly, some generic properties will require calculations that depend on the specialization; e.g. fetching the list of event ID's in an event dataset may require reading its (specialization-specific) data files.

Thus we are led to a polymorphic model: we identify the properties associated with generic types and then require concrete subtypes to provide the methods to return these properties. We do not expect a simple inheritance tree (as supported by XML schema) to suffice; for example, a dataset may have events, data in files, 64-bit ID's or any combination of these properties.

It is allowable to carry some properties in the generic types and for the appearance of these properties to be optional. When a generic property is missing from the corresponding type, it must be possible to discover the property using the subtype. Discovering a property may simply be a matter of parsing the XML or may require calling a function which has access to the component's data (i.e. a method). The XML schema and function are those corresponding to the type carrying the property.

Programmatic interface

Providers of tools and services are expected to access AJDL components using their generic interfaces. In object-oriented languages it is natural to introduce classes corresponding to the AJDL generic types. These classes have methods corresponding to the properties associated with the corresponding AJDL type. Bindings are provided for all anticipated languages, in particular C++, python and possibly java.

This interface must also include means to write the XML representation of a component and to create an object with the generic interface from the XML description of a concrete type.

User interface

Users may directly access AJDL components using python, java or CINT/C++. We also anticipate providing a command line interface for some subset of properties. The input for these commands would be file-based XML descriptions.

Web service interface

AJDL should provide means to expose some of the generic interfaces as web services. At a minimum this includes the scheduler. A scheduler implemented in any of the above languages can thusly be deployed as an analysis service. The AJDL specification must include WSDL for those types that are exposed as web services.

Implementation of subtypes

Users of AJDL tools and services will typically need to provide concrete subtypes for dataset and application. This includes XML schema and all functions required to override those in the base types. The functions may be implemented in any supported implementation language. An AJDL-provided wrapping mechanism will then make them available in the other languages. This might be a language-to-language wrapping or might use a command line executable or web service as an intermediary. Initially we anticipate supporting only C++ as the implementation language but later may add python or java.

In cases such as the scheduler where AJDL provides a web service binding, the provider may directly implement (in any language) a web service meeting the AJDL WSDL. AJDL provides the means to access this service from any of the generic type interface bindings.

<i>process(Task, Dataset) : string</i>
--

Generic type interfaces

The following sections list and specify interfaces for the AJDL generic types and a few useful concrete types. In addition to the properties indicated, each type must provide yet-to-be-defined means to access its type information and unique identifier (string representation). The methods associated with each type are shown in the figures. Italics indicate abstract classes or methods.

Application

The generic application provides methods to return the commands (most likely scripts) process a dataset and to build a task. They return empty strings if the script is not accessible. The methods are not implemented here.

<i>Application</i>
<i>process(Task, Dataset) : string</i>
<i>build_task(Task) : string</i>

PreInstalledApplication

A preinstalled application holds a package name and version and searches standard locations for an instance of that application.

PreInstalledApplication
name() : string
version() : string
process(Task, Dataset) : string
build_task(Task) : string

Schedulers configured to work with fixed applications may use the name and version to select or verify the application. In this case, the process and build_task methods may return blank.

PackagedApplication

A packaged application makes use of a package management system or service to locate the package with the scripts. It also adds a new method install to install that package and any others required.

<i>build_task(Task) : string</i>

The type is abstract and subtypes specify and provide means of using the associated package management scheme.

PackagedApplication
<i>install() : Status</i>

Task

A task carries data that a user can use to configure an application. These might include runtime parameters or code to be compiled and called from the application. For

simplicity, we assume this data is contained in a collection of named physical files. The path names for these files are relative to a working directory.

Task
<i>files()</i> : <i>NameList</i>
<i>extract(string dir)</i> : <i>Status</i>

There is a method to return the list of file names and a method to extract these files into a specified directory.

EmbeddedTask

In EmbeddedTask, the contents of the files are carried as part of the task data.

EmbeddedTask
Files() : <i>NameList</i>
extract_files(DirName) : <i>NameList</i>

Dataset

Dataset is the most complicated component type. It specifies the data to be processed where the data may be of almost any type or format. As with the task, the analysis service is not concerned with the data itself but may have the responsibility of running jobs close to the data or moving data to the sites where the jobs are run.

Dataset
<i>is_locked()</i> : <i>bool</i>
<i>is_virtual()</i> : <i>bool</i>

From the point of view of distributed analysis, the most important action taken on a dataset is splitting. This defines the boundaries for the job splitting that makes distributed analysis possible. Subtype information is required to split a dataset.

Typically datasets undergoing analysis are locked, i.e. immutable, but we allow for the possibility (e.g. during construction) that they are not. An analysis service may simply require that input datasets be locked and reject jobs that do not meet this requirement.

A dataset may be virtual, i.e. some action must be taken to transform it into a concrete dataset before an application can be run. This can occur because the dataset has not yet been constructed, because it is a virtual dataset ID appearing in a dataset replica catalog, or because it is a query to be applied against a dataset selection catalog.

A few important subtypes are described in the following but the list is far from exhaustive. Applications will generally require another subtype to provide the data interface natural to the application.

VirtualDataset

A virtual dataset implements the Dataset method and obviously returns true when asked if it is virtual. A scheduler receiving a virtual dataset may consult a dataset replica catalog to find a concrete instance or a virtual data catalog to find the prescription to create a replica.

VirtualDataset
is_locked() : <i>bool</i>
is_virtual() : <i>bool</i>

CompoundDataset

A compound dataset is made up of other datasets. It provides a method to return its list of constituents. A common strategy for splitting datasets is to use this list of constituents, i.e. to split it into the same pieces used to create the original dataset. The constituent datasets may themselves be compound datasets leading to a tree structure. Splitting may descend this tree until sub-datasets of the desired size or number are found.

A method is provided to append a dataset to the list of constituents. The dataset must be unlocked for this to succeed. The type is abstract. Subtypes specify the data layout and check the consistency of datasets when appending.

CompoundDataset
<i>append(Dataset) : status</i>
<i>constituents() : DatasetList</i>

EventDataset

An event dataset is one whose data includes a collection of events that can each be processed independently. Thus the event boundaries are natural boundaries for splitting. The term event of course arises from its use in HEP but other realms have datasets with similar behavior. Analysis services that process event datasets i.e. split along event boundaries and process events independently form a restricted but widely applicable category.

The type is abstract.

EventDataset
<i>event_count() : int</i>
<i>event_ids() : EventIdList</i>
<i>select_events(EventIdList) : int</i>

LogicalFileDataset

A logical file dataset is one for which at least part of the data is found in a set of logical files. The list can be used by the application or analysis service to stage files in advance of processing or move them to permanent storage after processing. Although it is natural to split datasets along file boundaries, this subtype does not have enough information to make such a split. Typically the constituents of compound datasets will reflect file boundaries. In any case, the information here can be used to check whether split datasets fall along these boundaries.

LogicalFileDataset
<i>logical_files() : LfnList</i>

Configuration

The final piece in specifying a job is the configuration which is not part of the essential provenance but provides hints for running the job. It could include authentication and authorization information (user identity and role), desired and maximum acceptable response time, accounting information (who to charge and maximum to spend), etc.

Job

The final component type is the job itself. It holds the input application, task, dataset and configuration and the generated result. It also holds the status and start, stop and update times. Subtypes can be used to extend this information. Low level jobs that call an

application would add the machine where the job was run, the clock and CPU times, maximum memory consumed, etc. Further subtypes might be used to specify the job management system, e.g. LsfJob might add a queue name.

Job
application() : Application
task() : Task
dataset() : Dataset
configuration() : JobConfiguration
start_time() : Time
update_time() : Time
stop_time() : Time
result() : Dataset
status() : JobStatus
subjobs() : JobIdList

Job also carries a list of ID's for the sub-jobs if the job is split. This list will be empty if there are none implying that this job directly called the application. This information could be moved to a subtype CompoundJob.

Scheduler

The table at the right presents the methods of the scheduler (aka analysis service).

Scheduler
has_application(Application) : bool
has_task(Application, Task) : bool
install_task(Application, Task) : Status
submit(Application, Task, Dataset, Configuration) : JobId
job(JobId) : Job
kill(JobId) : Status

As indicated in the following section, an ID can be substituted for any of the base types (e.g. Dataset replaced with DatasetId) if the service has access to a repository (see below) holding the object.

Object ID's

An object of any type carries a unique ID. This implies that all the above generic types additionally have a method id() that returns this value. It also implies we can sensibly use an ID in place of any object if we have a repository service to retrieve the object description with the ID.

Catalogs

We require means to store, retrieve and select objects of the AJDL types. The following sections describe relevant catalog services.

Repository

A repository service is one where an object with an ID can be stored and then later retrieved using the ID. In the table, Xyz stands for any of the base object types (Application, Task, Dataset, Configuration, Job or Scheduler). We will need all of these.

XyzRepository
store(Xyz) : Status
retrieve(XyzId) : Xyz
remove(XyzId) : Status

Replica catalog

A replica catalog stores associations between a reference ID and a collection of replica ID's. The primary case of interest is the dataset replica catalog.

XyzReplicaCatalog
insert_reference(XyzId) : Status
insert_association(XyzId, XyzId) : Status
remove_reference(XyzId) : Status
remove_association(XyzId, XyzId) : Status
reference_count() : int
has_reference(XyzId) : bool
has_replica(XyzId) : bool
get_replicas(XyzId) : XyzIdList
get_reference(XyzId) : XyzId

Metadata catalog

A metadata catalog, also known as a selection catalog, associates metadata with object ID's and allows user to query for ID's based on the metadata.

XYZMetadataCatalog
insert_id(XyzId) : Status
insert_metadatum(XyzId, NameValue) : Status
insert_metadata(XyzId, NameValueList) : Status
size() : int
ids() : XYZIdList
get_metadata(XyzId) : NameValueList
select(Query) : XYZMetadataCatalog

This is clearly of interest for datasets and tasks and may be useful other generic types.

Virtual data catalog

The virtual data catalog records the prescriptions for creating datasets, i.e. the application, task, input dataset for each output dataset. Both the input and output datasets appearing in this catalog are virtual. In order to have complete provenance, we must also record starting datasets and datasets formed by merging other datasets.

This is a special case of DatasetMetadataCatalog and other dataset metadata might simply be merged into this catalog.

Job catalog

The provenance in the previous table is the essential or abstract provenance. It does not include concrete information such as the splitting into sub-jobs and time and location where each sub-job was run. This information appears in the job catalog. With our job definition and assuming sufficient detail in the job subclasses, the job repository contains all this information.

It might be convenient to extract some subset of this data into a job selection (metadata) catalog.

Catalog summary

The following catalogs have been identified as useful:

- Application repository
- Task repository
- Dataset repository
- Configuration repository
- Job repository – serves as job catalog

Dataset replica catalog
Task metadata catalog
Dataset metadata catalog – probably merged into the following
Virtual data catalog – records dataset provenance

Comments

Chimera

The GriPhyN Chimera project is developing a virtual data language similar to the model described here. It is not service-based but is expected to include datasets in its next version. (The current release manipulates files.) There is considerable overlap between the models: our application and task are closely related to their transformation and transformation parameters. Discussions are taking place to try to merge the models or at least come to agreement on defining terms such as dataset.

Dataset content

There is a long discussion on dataset properties in the document “Datasets for the Grid” available at <http://www.usatlas.bnl.gov/ADA/docs>. One of those properties that have been neglected here is content. In the case of event datasets it specifies what kind of data (RAW, ESD, AOD; tracks, jets, electrons ...) is present for each event. If this information is added to the dataset interface and the application and task interfaces are extended similarly, then we determine at a generic level whether a particular dataset is suitable for processing with a given application/task.

It also becomes possible to select the part of a dataset corresponding to a specified sub-content, e.g. the AOD part of a dataset that includes RAW, ESD, AOD and TAG. We can reduce the number of entries in the dataset provenance catalog by including only complete datasets, i.e. those for which all ancestor data is included in the dataset. The type information in the application/task is used to select the relevant data to deliver for processing.

Conclusions

Another pass at defining a high level job language has been presented. It is based on the C++ model that evolved in the DIAL project. It is intended as a step toward defining a broad-based standard interface that would allow different realms to share analysis tools and services.