# The AstroGrid Common Execution Architecture (CEA)

*Paul Harrison, Noel Winstanley & John Taylor, AstroGrid (www.astrogrid.org)*

## Introduction

The Common Execution Architecture (CEA) is an attempt to create a reasonably small set of interfaces and schema to model how to execute a typical Astronomical application within the Virtual Observatory (VO). In this context an application can be any process that consumes or produces data, so in existing terminology could include

- A Unix command line application
- A database query
- A web service

The CEA has been primarily designed to work within a web services calling mechanism, although it is possible to have specific language bindings using the same interfaces. For example Astrogrid has a Java implementation of the interfaces that can be called directly from a Java executable.

## Motivation

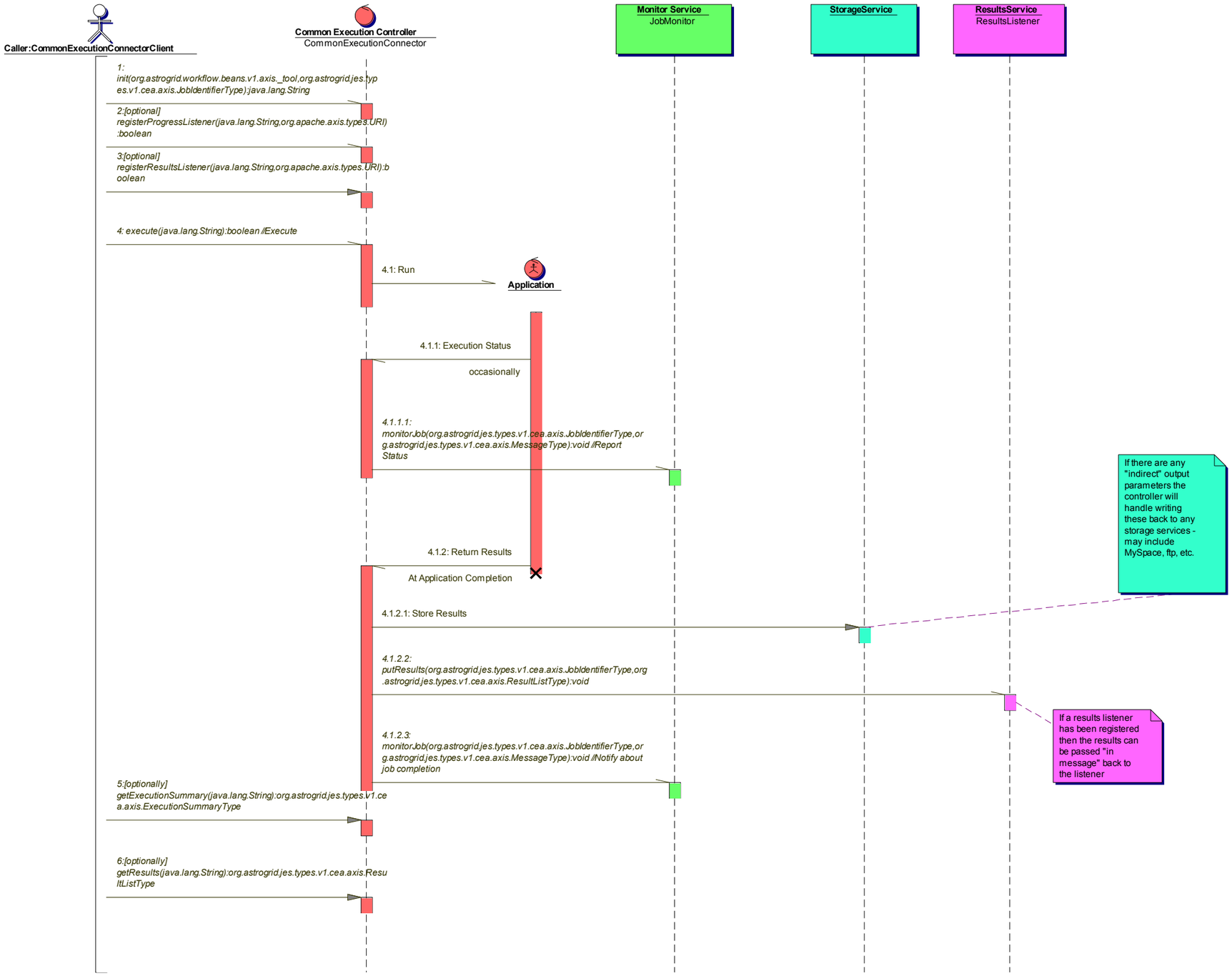
The primary requirements motivating the creation of this architecture are;

- To create a uniform interface and model for an application and its parameters. This has twin benefits;
  1. It allows VO infrastructure writers a single model of an application that that have to code for.
  2. Application writers know what they have to implement to be compatible with a VO infrastructure.
- To provide a higher level description than WSDL1.1 can offer.
  - Restrict the almost limitless possibilities allowed by WSDL into a manageable subset.
  - Provide specific semantics for some astronomical quantities.
  - Provide extra information not allowed in WSDL - e.g. default values, descriptions for use in a GUI etc.
- To provide extensions with the VOResource schema (See the IVOA WG) that can describe a general application
- To provide asynchronous operation of an application - This is essential as the call tree that invokes the application cannot be expected to be active for extremely long lasting operations - e.g. a user from a web browser invokes a data-mining operation that takes days
  - Provide callback for notification of finishing.
  - Provide polling mechanisms for status.
- To allow for the data flow to not necessarily have to follow the call tree. In a typical application execution the results are returned to the invoking process - In a VO scenario, it can be useful if the application can be instructed to pass the results on to a different location

## Origins

Amongst the VO specifications there was no existing model for applications that was defined at the level at which this design attempts to address. In the VOResource schema an application is defined as a Service with the interface definition. The interface definition either relies on referring to a WSDL definition of the service, or on other schema extending the service definition to provide some specific detail as in the case of a Simple Image Access service. There is no general definition of an application in the resource.

It is clear that the WSDL model of an interface has had a large influence on the design of the CEA, but it should be remembered that the CEA is intentionally layered on top of WSDL so that CEA controls the scope and semantics of operations. There is only one WSDL definition for all applications, so as far as web services are concerned the interface is constant. CEA works by transporting meta information about the application interface within this constant WSDL interface.




## Components

- **Application** - This is the process that is to be executed. It is defined as a process that can consume or create data. So this can include Unix command line tools, database queries, web services etc.
- **Common Execution Controller** - this is the component that implements the *CommonExecutionConnector* interface, and actually controls the execution of the application. There can be various specialisms of this service, such as the CommandLineApplicationController, which can be configured to invoke a general Unix command line tool, a WebServiceApplicationController, which can be configured to act as a proxy to call a general web service in a uniform manner
- **Invoking process**
- **Monitoring Service** - This is a service that the Common Execution Controller can report status to - it can of course be
- **Storage Service** - this is the mechanism by which the application can return its results in the indirect parameter mode (see indirect parameters).
- **Results Service**

This sequence diagram illustrates how the various components of the CEA system interact when an application is executed. The steps are

1. The invoking process calls the init method of the CommonExecutionConnector interface, which is implemented by the component known as the CommonExecutionController. This will set up the execution environment for the the application and will return immediately with an executionID which is the identifier by which the *CommonExecutionController keeps track* of this particular execution instance. The parameters to this call are
    - A Tool object - This is described in more detail below.
    - JobIdentifier - this is the identifier by which the *invoking process uses to keep track* of this particular execution instance.
2. the invoking process then has the opportunity to register two classes of listener
    - Status Monitor - this is the endpoint of the service that implements the JobMonitor interface that the ExecutionController can call to inform the monitoring process of the status of the execution instance.
    - Results Listener - this is the endpoint of a service that implements the ResultsListener port so that the ExecutionController can report the results of the application execution once they are ready
3. Then the execute operation should be invoked and the CommonExecutionController will then start the application.
4. The application can then optionally return status information to the CommonExecutionController which will then pass this on to the Monitor Service.
5. When the application completes it will inform the CommonExecutionController which will then pass the indirect results on to the storage service, the direct results back to any results listeners and inform the monitor service that the application has finished.

Some point of note;

- The monitoring/resultListening services could equally be the same as the invoking service - they are shown as conceptually separate, as the endpoint of this service is passed in as an argument to the registering call. Indeed if required there could be many status and results listeners for a single application execution.
- The only guaranteed status message that the monitoring service will receive is the one informing it that the application has finished (or failed). The application might be capable of sending intermediate messages whilst it is sill executing, but this is not required.
- The results of the application are not necessarily returned directly to the invoking process. For "indirect" output parameters, the final destination for the result data is implicit in the specification of the output parameters, and it is the responsibility of the ExecutionController to ensure that they get to the desired storage service.
- The results will also always be passed to the resultsListener if registered. In the case of an indirect parameter, then only the URI that specifies the location will returned, otherwise the full value will be returned.

# Interfaces

The 3 WSDL ports that are used to interact within CEA are  briefly described below;

## CommonExecutionConnector

This is the main port that is used to communicate with the application. The main operations in this port are;

- **init** - this will initialize the application environment - returns and executionId by which
- **registerResultsListener** - any number of services can register themselves as wanting to receive the results from the run when they are available as long as they implement the ResultsListener port below
- **registerProgressListener** - any number of services can register themselves as wanting to receive status messages during the run as long as they implement the JobMonitor port below
- **execute** - will actually start the asynchronous execution of the application specified in the init call.
- **queryExecutionStatus** - this call can be used to actively obtain the execution status of a running application, rather than passively waiting for it as a JobMonitor
- **abort** - will attempt to abort the execution of an application
- **getExecutionSummary** - request summary information about the application execution
- **getResults** - actively request the results of the application execution, rather than passively waiting for them as a ResultsListener.
- **returnRegistryEntry** - this returns the registry entry for the particular CommonExecutionConnector instance - this will probably be removed from this interface to be replaced by the equivalent operation in the standard VO service definitions.

## JobMonitor

The only operation in the `JobmMonitor` port is the `monitorJob` operation, which expects to receive a message with the job-identifier-type (as specified in the original init operation of the CommonExectutionConnector port) and a status message

## ResultsListener

The only operation is the `putResults` on the `ResultsListener` port. This accepts a message that contains a job-identifier-type and a result-list-type, which is just a list of parameterValues.

## Objects

The objects that participate in CEA can be split into two groups

1. Those used to describe the application in the registry
    - *Application* the overall application, which has a series of
    - *Parameter* which are the detailed descriptions of the parameters and their types.
2. Those used to describe the application in the WSDL interface
    - *Tool* - An instance of an application with real parameter values.
    - *ParameterValue* which is used to pass a values to a Tool.

### Application

As this model depicts an application in CEA is really quite a simple entity consisting of 1 or more interfaces that consist of 0 or more input parameters and 0 or more output parameters.

### Parameter Definition

The description of the parameters and the parameter values are probably the heart of the CEA. It is the model for the parameters that allow us to add semantic meaning, and to give the flexibility in how the parameters are transported. The implementation is still in its infancy, but it is hoped that the parameter definition will be extended to encompass any data models that the VO produces.

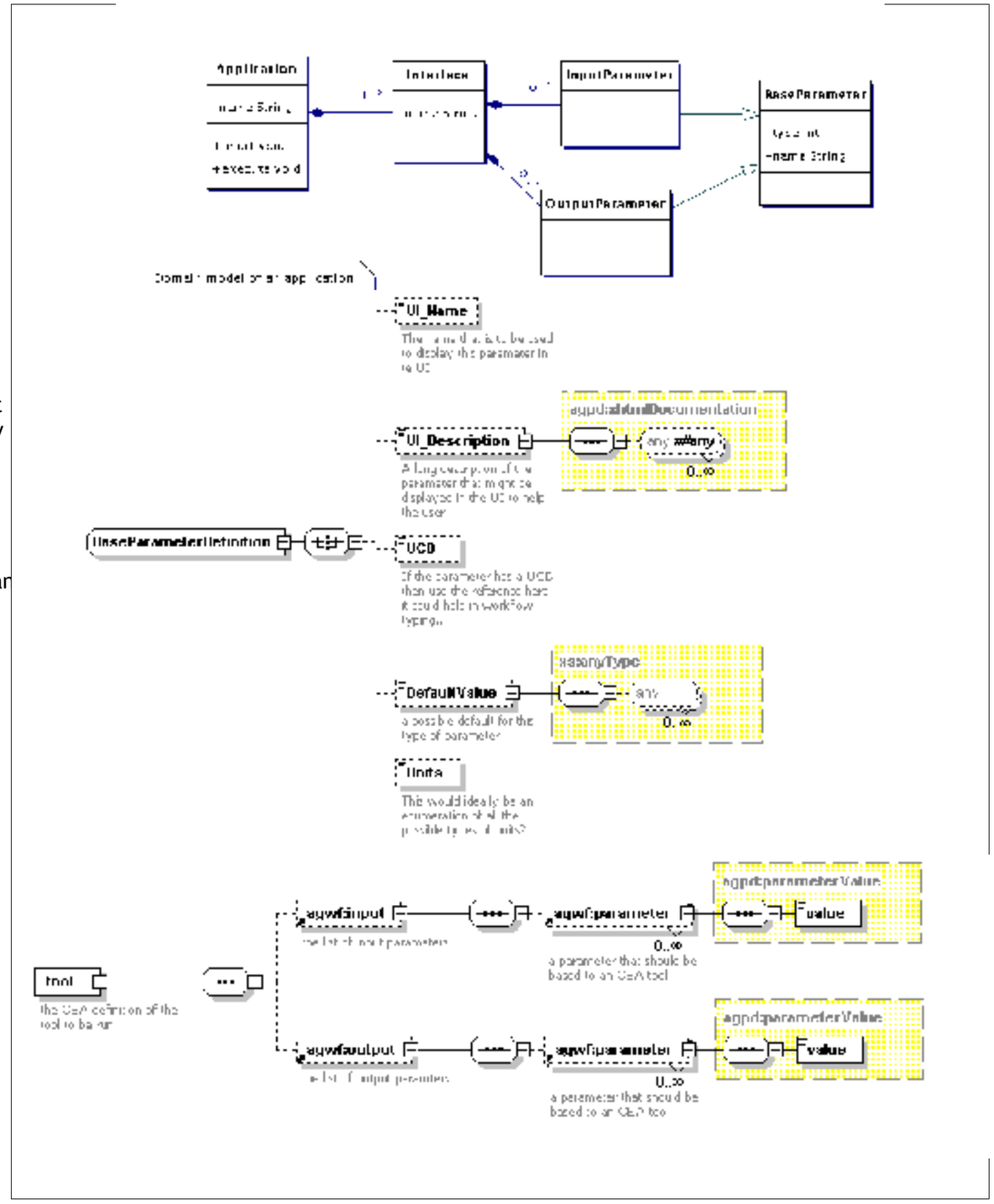
### Tool

The tool represents the full collection of parameters that are passed to a particular interface of an application and the results that are returned.

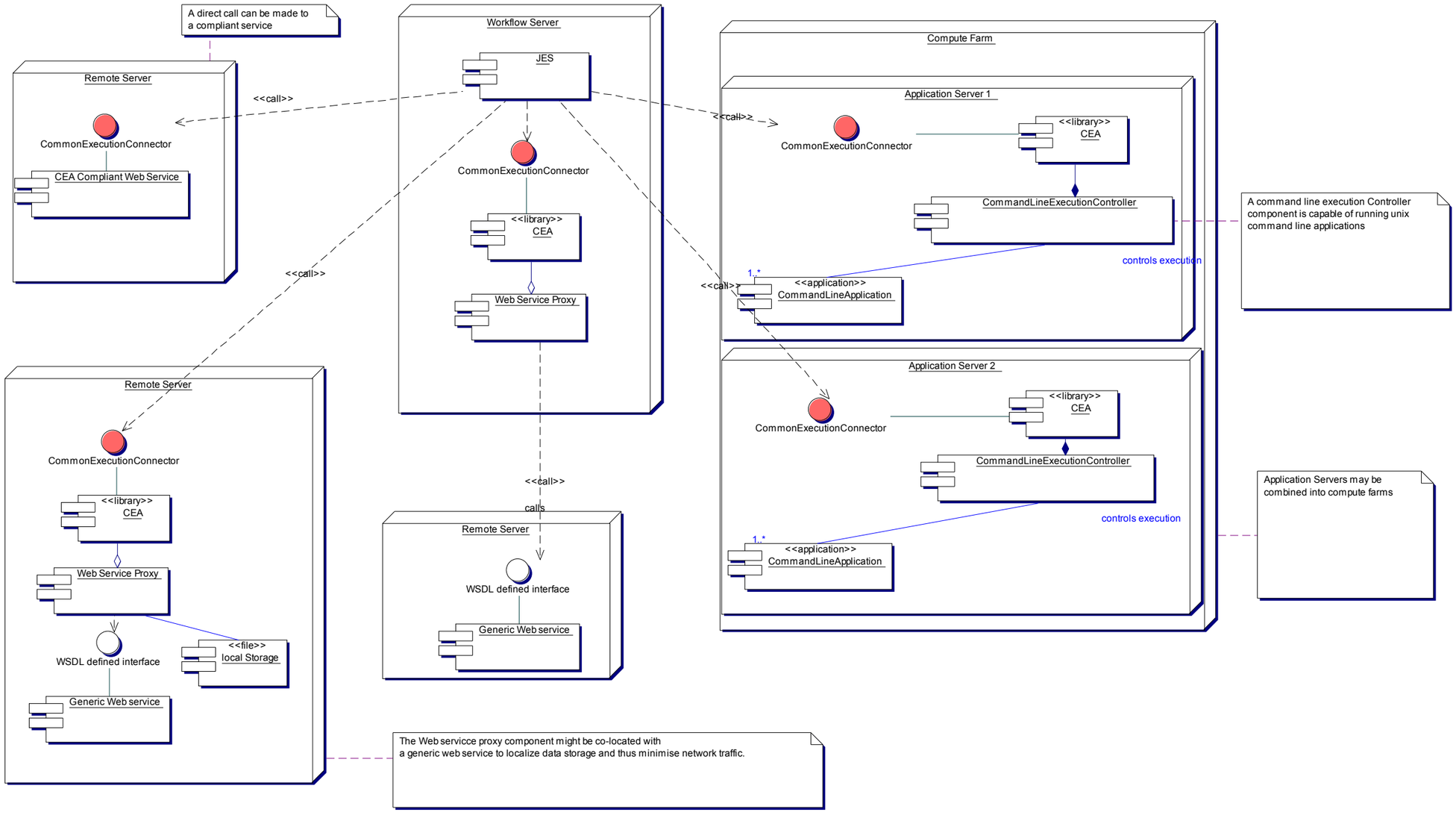
### ParameterValue

The parameterValue model is simple but powerful representation of the parameters that are passed to an application. The parameterValueelement has 2 attributes

- name
- indirect This describes whether the value element of the parameter should be used as is (indirect=”false"), or if the value of the parameter represents a URI from which the actual value should be fetched (indirect="true"). It has not been defined what the minimum set of transport mechanisms a service should understand to be CEA compliant, but the different sorts of transport mechanism are expected to include
    - SOAP messages
    - http get/put
    - SOAP attachments
    - ftp/gridftp
    - MySpace
    - local filestore
    - etc.



# Deployment




This deployment shows some of the features of using the CEA

On the right hand side of the diagram there are command line applications that are wrapped by specialized CommonExecutionControllers that allow the workflow engine to use the CommonExecutionConnector interface to communicate

- There is a webservices proxy component that can act as an adaptor between a generic web service and the CommonExecutionConnector interface
- On the left of the diagram the webservices proxy is localised with a web service so that the results returned by the webservice can be stored locally thus minimising network traffic

# Astrogrid Implementation

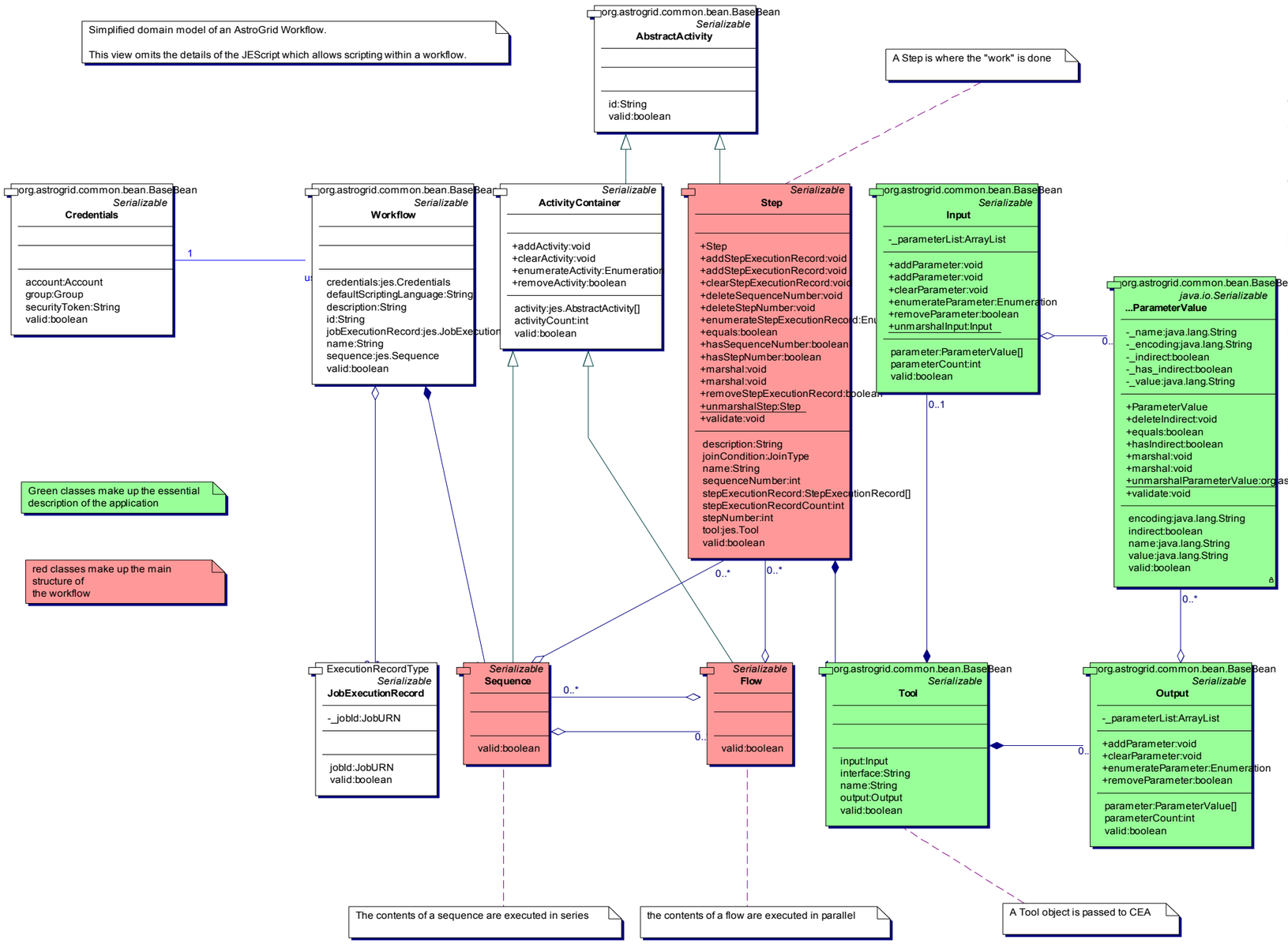
## CEA

AstroGrid has created a set of software that implements the CEA, which is documented at http://www.astrogrid.org/maven/docs/snapshot/applications/. The main server components that implement the CommonExecutionConnectorweb service are called ExecutionControllers and there are currently two specialized adapter controllers for

- Wrapping legacy command line applications in the framework.
- Wrapping legacy GET/POST style HTTP applications in the framework.

There are plans for a further adapter for other web services themselves as well as the possibility of creating new applications that directly implement the CEA.

## Job Execution System (JES)/Workflow

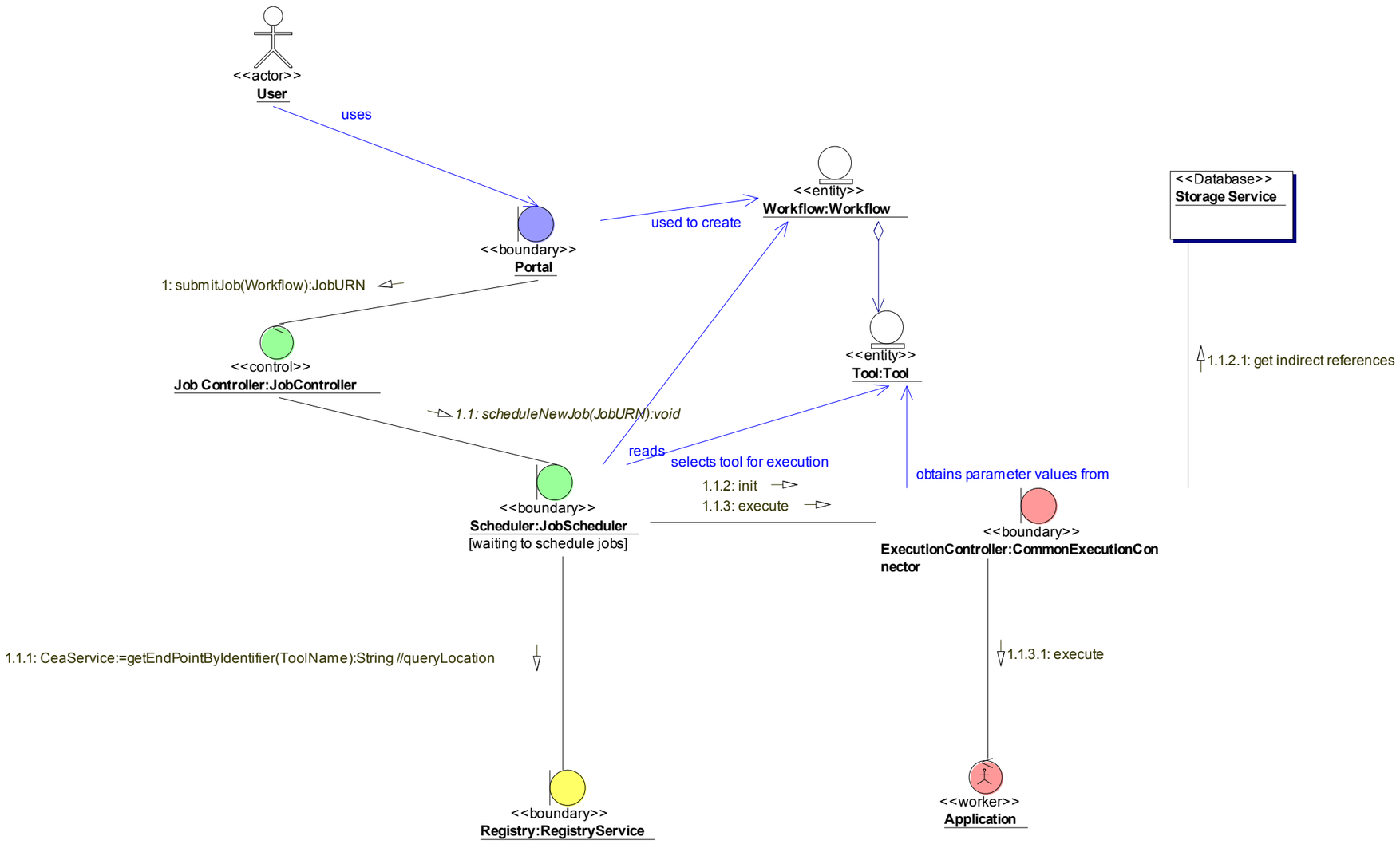



The CEA only specifies how to call applications, there is another component with the responsibility for actually organising the order of execution of a sequence of applications – this is JES (http://www.astrogrid.org/maven/docs/SNAPSHOT/jes/). The job system in AstroGrid is driven from the workflow which has a class structure as shown below.

The unit of execution within the workflow is a Step, which can itself be part of a Sequence (a set of actions performed in series) or a Flow (a set of actions performed in parallel). The Step contains a Tool which is the full CEA description of a particular execution instance of an application. The workflow has an XML representation which makes it easy to manipulate and build workflows with existing tools such and XML editors. This diagram does not in fact show the full complexity of the workflow as it also includes a scripting component known as JEScript (based on the Java scripting language Groovy - ) which allows control structures such as loops and conditionals to be placed in the workflow.

# Interaction between CEA and JES

The diagram below shows some of the typical interactions that occur between the subcomponents of CEA and JES when a user submits a workflow.



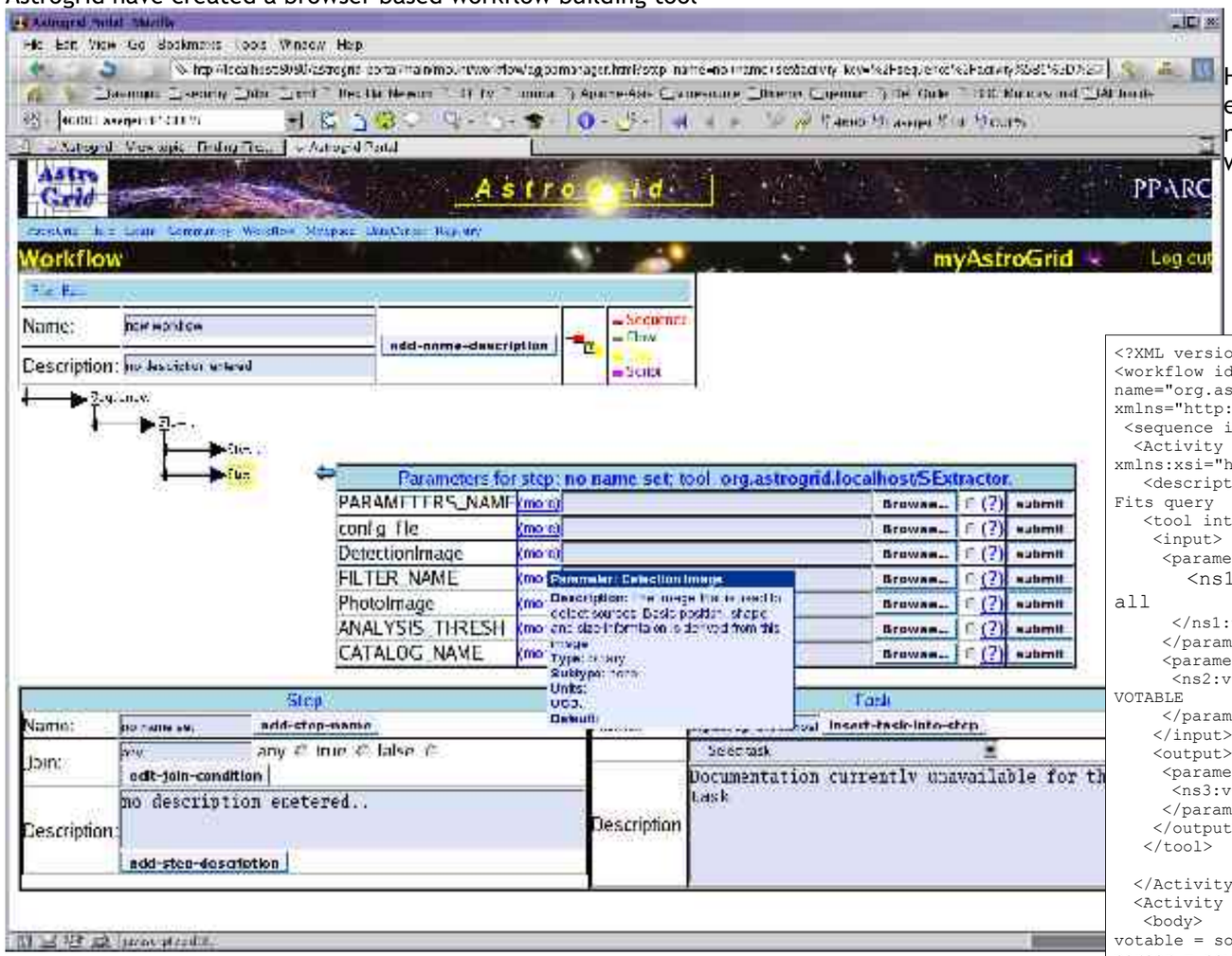

The user is able to design the workflow in the AstroGrid portal and then submit it to the JobController which will simply queue the job for execution. The JobScheduler will then pick this up and attempt to run each of the steps in the workflow. To execute a particular step the JobScheduler has to query the Registry to find out which ExecutionController can run a particular application. The call to the ExecutionController then occurs in two phases

- The init phase, where the parameter values are passed and any indirect parameters are copied from a storage service – e.g. MySpace or ftp.
- the execute phase, where the application is actually run in an asynchronous fashion.

The JobScheduler knows that a particular application has finished by registering itself as a status listener with the ExecutionController (not shown on the diagram). There can be arbitrary numbers of status listeners and results listeners for a particular execution run of an application.

## Astrogrid Workflow Builder

Astrogrid have created a browser based workflow building tool




However a workflow is completely defined by itsXML representation, so it is possible to edit the document in a text editor. An example workflow is presented below. It should be noted that this example there is some JEScript that parses a VOTableto extract some URLs which are then passed on as parameters to the next step.

```
<?XML version="1.0" encoding="UTF-8"?>
<workflow id="N10001"
name="org.astrogrid.workflow.externaldep.itn6.solarevent.CompositeFitsVotableParsingConcatWorkflowTest"
xmlns="http://www.astrogrid.org/schema/AGWorkflow/v1">
 <sequence id="N10005">
  <Activity id="N10006" joinCondition="any" name="org.astrogrid.localhost/trace_dsa" result-var="source"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="step">
   <description>
Fits query   </description>
   <tool interface="adql" name="org.astrogrid.localhost/trace_dsa">
    <input>
     <parameter indirect="false" name="Query">
       <ns1:value xmlns:ns1="http://www.astrogrid.org/schema/AGParameterDefinition/v1"> select
all
      </ns1:value>
     </parameter>
     <parameter indirect="false" name="Format">
      <ns2:value xmlns:ns2="http://www.astrogrid.org/schema/AGParameterDefinition/v1">
VOTABLE      </ns2:value>
     </parameter>
    </input>
    <output>
     <parameter indirect="false" name="Result">
      <ns3:value xmlns:ns3="http://www.astrogrid.org/schema/AGParameterDefinition/v1"/>
     </parameter>
    </output>
   </tool>

  </Activity>
  <Activity id="N10024" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="script">
   <body>
votable = source.Result; // access result of previous step
parser = new XmlParser(); //create new parser
nodes = parser.parseText(votable); //parse votable into node tree
urls = nodes.depthFirst().findAll{it.name() == 'STREAM'}.collect{it.value()}.flatten(); // filter node tree on
'STREAM', project value
print(urls); // show what we've got
concatStep = jes.getSteps().find {it.getName() == 'concat-step'}; // find next step in workflow
inputs = concatStep.getTool().getInput(); // get to set of input parameters
inputs.clearParameter(); // clear what's there already
urls.each { p = jes.newParameter(); p.setName('src'); p.setIndirect(true); p.setValue(it); inputs.addParameter(p);} //
add a new parameter for each url
   </body>
  </Activity>
  <Activity id="N10029" joinCondition="any" name="concat-step" result-var="finalResults"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="step">
   <tool interface="basic" name="org.astrogrid.localhost/concat">
    <input>
     <parameter name="src">
      <ns6:value xmlns:ns6="http://www.astrogrid.org/schema/AGParameterDefinition/v1"/>
     </parameter>
    </input>
    <output>
     <parameter indirect="true" name="result">
      <ns7:value xmlns:ns7="http://www.astrogrid.org/schema/AGParameterDefinition/v1">
ivo://org.astrogrid.localhost/myspace#frog/CompositeFitsVotableConcatWorkflowTest.dat      </ns7:value>
```