*I*nternational

*V*irtual

*O*bservatory

*A*lliance

Design and Implementation of the AstroGrid Workflow system

Version 1.00

IVOA Note 2006 February 27

Author(s):

Noel Winstanley nw@jb.man.ac.uk

Abstract

This note describes the AstroGrid Workflow system, including details of the implementation. The successes and shortcomings of the current system are identified and improvements are suggested. Some general recommendations to the IVOA are made, based on the experience gained. It is hoped that this document will benefit further research and development of workflow for the virtual observatory.

Status of This Document

This is a Note. The first release of this document was 27 February 2006

This is an IVOA Note expressing suggestions from and opinions of the authors. It is intended to share best practices, possible approaches, or other perspectives on interoperability with the Virtual Observatory. It should not be referenced or otherwise interpreted as a standard specification.

A list of current IVOA Recommendations and other technical documents *can be found at* http://www.ivoa.net/Documents/.

Acknowledgements

The first prototypes of an AstroGrid Workflow system were produced by Jeff Lusted. The design was refined in tandem with Paul Harrison's development of the CEA architecture. The user interface is the work of Phil Nicholson.

Contents

1 Introduction....2
2 Workflow Execution Environment....3
2.1 Common Execution Architecture (CEA) for Applications....4
2.2 MySpace....5
3 The Workflow Document....5
3.1 Sequencing & Iterating....6
3.2 Workflow Variables....6
3.3 Script Expressions....7
3.4 Branching....7
3.5 Step....7
3.6 Error Handling....9
3.7 Scripting....9
4 User Environment....10
4.1 Workflow Builder....11
4.2 Progress Monitor....12
4.3 Transcript Viewer....13
4.4 MySpace Browser....14
5 Implementation of JES....14
5.1 Scheduler....16
6 Challenges....17
7 Suggestions for IVOA....18

1 Introduction

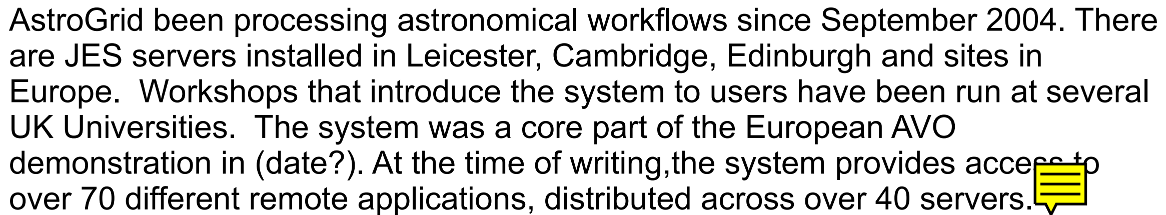
AstroGrid Workflow is a multi-user batch system for the unattended execution of potentially long-running astronomical workflows.

The input is a *workflow document,* authored by a user, that describes which remote *applications* — data collections and processing packages — are to be used. These applications may be **distributed** throughout the virtual observatory.

The workflow document is passed to a *Job Execution Server* (JES) for execution. After this the user has no further input and need not remain online. The JES orchestrates the workflow: it evaluates flow-of-control logic, and dispatches invocations in **parallel** to the required applications.

The results and intermediate products of the workflow are stored in *MySpace,* a distributed file-system[1], from where they can be accessed later by the user. The user can also retrieve a *workflow transcript* that reports how the workflow was executed.

AstroGrid been processing astronomical workflows since September 2004. There are JES servers installed in Leicester, Cambridge, Edinburgh and sites in Europe. Workshops that introduce the system to users have been run at several UK Universities. The system was a core part of the European AVO demonstration in (date?). At the time of writing,the system provides access to over 70 different remote applications, distributed across over 40 servers.



Scientists working with AstroGrid have produced a selection of *parametrized workflows[2]* - pre-packaged workflow templates that prompt the user for input parameters before being executed by a JES. These perform tasks such as colour-cutting[3] and redshift-making[4]. This is a convenient way of publishing useful workflows; allows novice users to gain experience in the system; and provides starting points for developing new workflows.

2 Workflow Execution Environment

Workflow has always been a designed-in part of the AstroGrid architecture. To execute a workflow a JES cooperates with other AstroGrid Services — for resource resolving, application execution, and data storage. Each of these services complies with IVOA standards where available - in areas that lack standards, proposals have been made [5], [6] based on experiences gained assembling the AstroGrid system.

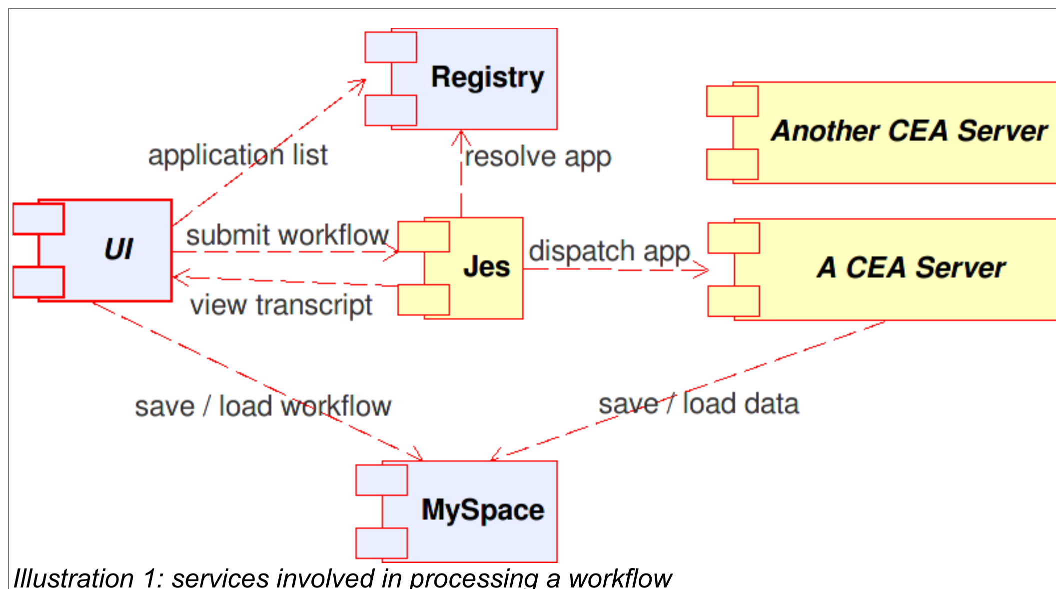


Illustration 1: services involved in processing a workflow

The diagram illustrates the stages in processing a workflow. The user interface constructs a workflow document (or loads a previously saved document) and submits it to a JES.

The JES creates a new job to execute the workflow and a globally-unique identifier for this job is returned to the user interface. When the user wishes to track the progress of the job, this identifier can be used to query the JES for the workflow transcript: a copy of the original workflow document annotated with execution progress and messages from remote applications.

A workflow document contains a set of *activities*. Activities may either be a *container* or a *primitive*. Container activities, such as loops, conditionals, sequences and parallelizations, compose together further sets of activities. The JES evaluates these constructs to determine which parts of the workflow to execute next.

The most important primitive activity is a *step*: this represents an invocation of a remote application - for example running an ADQL[7] query against 2MASS or running HyperZ over data in a VOTable[8]. The step activity describes which application to invoke, and the parameters to supply to it. Some of the input parameters may be references to MySpace files; similarly the output parameters may point to MySpace files in which to store the results.

2.1 Common Execution Architecture (CEA) for Applications

JES can execute any application that adhere to the *Common Execution Architecture* (CEA) which is specified in an IVOA Note[9]. This section covers the aspects of CEA that are relevant to AstroGrid Workflow.

The CEA standard defines the webservice interfaces, message protocols, and formats that a executable application must support, and also defines how applications should described in an VO Registry. It ensures a uniform calling convention and meta-data for all applications, which simplifies the JES service.

An application may be provided by more than one *CEA server* - this gives redundancy and load-sharing. The JES dispatches a step by first using the registry to resolve the application to a list of CEA servers that provide it. JES selects a server, records it's choice in the workflow transcript, and then instructs the CEA server to asynchronously execute the application with the required set of parameters. The application is executed when convenient to the CEA server — it may enforce a queuing or scheduling policy to conserve resources. Once the step has been dispatched, the JES can continue processing other activities, and other workflows, until it receives notification from the CEA server.

The CEA defines webservice interfaces[10],[11] that CEA servers must use to notify a controlling process (in this case, a JES) when the status of a task changes. This is used when: the application starts running; completes; or ends in error. The server may also choose to report further progress or logging information back to the controller using this interface.

When a JES receives a notification from a CEA server it is recorded in the workflow transcript. If the notification states the application has completed or failed, JES examines the workflow and dispatches further steps. This repeats until the workflow is completed.

2.2 MySpace

AstroGrid MySpace is a location-independent distributed storage system. It provides an individual hierarchical file system for each user. CEA servers know how to read and write to MySpace, which in the workflow system is used as a 'buffer' for intermediate data, and as a place to leave final results. Before the CEA application starts, the CEA Server will fetch any input parameters stored in MySpace. Similarly, the results may be saved back to MySpace when the application completes.

Using MySpace in this way ensures that the minimum of information passes between JES and CEA servers - the bulk of the data flows between CEA and MySpace servers. This reduces the traffic and load on the JES server, ensuring it remains responsive.

3 The Workflow Document

Workflow documents are written in XML whose structure is formally defined by a set of schema published in [12].The root element is named **workflow**.

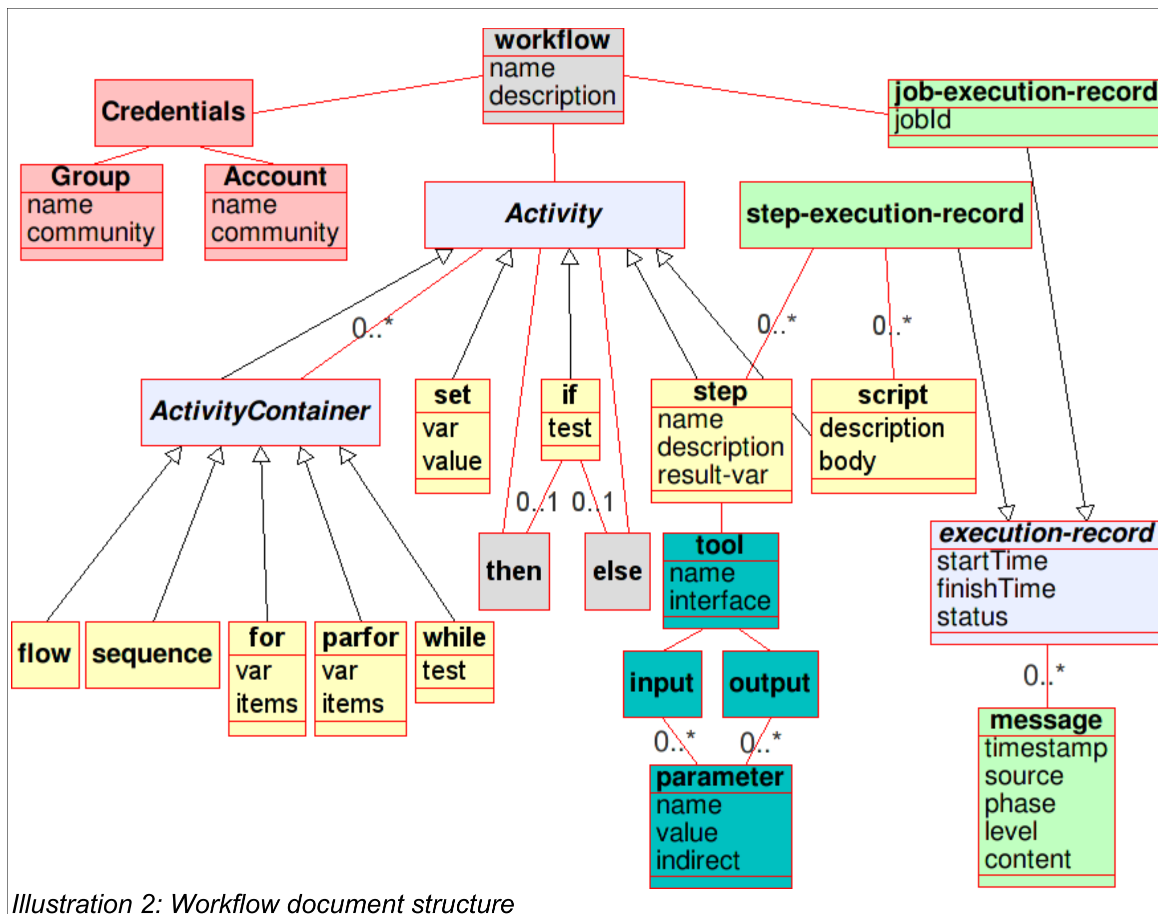


Illustration 2: Workflow document structure

In the figure, elements are denoted by boxes — the element name is emboldened, with attributes, if any, listed below. Elements with italicized names are 'abstract' - these are templates for other elements, but do not occur in the workflow document themselves. Red links denote a containment relationship, while black links denote an extends ('is kind of') relationship. Similar elements are grouped by colour.

The workflow element can contain three elements: a **Credentials** that identifies the user on whose behalf to run the workflow; an **Activity** that describes the operations the workflow must perform; and a **job-execution-record** that will record information about the overall execution of the workflow.

3.1 Sequencing & Iterating

Activity is an abstract element - it does not occur within the workflow document. Instead, one of the elements that extends from it (those coloured yellow) is expected in it's place. The root activity in a workflow is typically an **ActivityContainer** - an activity that contains a further set of child activities, such as:

- **flow** – indicates that the set of child activities may be executed in any order, or even in parallel. This activity completes when all it's children have completed.
- **sequence** – executes the child activities sequentially, in the order they appear in the document.
- **for** – execute the child activities for each item in the list **items**, binding the current value to the workflow variable named **var**[1].
- **parfor** – semantically identical to **for**, but operationally differs by executing each iteration simultaneously, completing when all iterations have completed.
- **while** – execute the child activities repeatedly while the condition **test** remains true.

The **flow** and **parfor** constructs are useful, but can only be used in circumstances where there is no data dependencies between the parallelized activities. Also note that they are advisory, not mandatory parallelizations - the JES is free to decide how best to execute the activity set.

3.2 Workflow Variables

A workflow may contain workflow variables — named variables that bind to strings. These variables are useful for storing constants, intermediate results, and for controlling which path through the workflow to take.

The **set** activity binds a string value to the variable named in **var** - the variable will be created if it does not already exist. The **for** and **parfor** activities also binds a value to a variable - but these loop variables are only visible to it's child

1 **for** behaves similarly to the 'for' in Python, not like the 'for' in C, which generates an arithmetic sequence.

activities. Finally, the **step** activity can bind it's results of an application execution to a variable - which will be explained later.

3.3 Script Expressions

Workflow variables become much more useful when used in conjunction with *script expressions*. These can occur within any attribute in the workflow document and are denoted as `${expr}`. At runtime the expression is evaluated and the result substituted into the attribute. This allows, for example, computed values to be passed as input parameters to CEA applications.

Script expressions are written in a rich language that allows references to previously defined workflow variables, numerical operations, string manipulation, and calls to library functions.

Store the value 'hello' in the variable 'a'

```
<set var="a" value="hello" />
```

Store the value 'HELLO WORLD' in the variable 'shout'

```
<set var="shout" value="${a.toUpper()} WORLD" />
```

Store the value '1' in variable 'b'

```
<set var="b" value="1" />
```

Store the values '1 + 1' in variable 'c'

```
<set var="c" value="${b} + 1" />
```

Store the value '2' in variable 'd'

```
<set var="d" value="${b + 1}" />
```

Example 1: setting workflow variables from script expressions

3.4 Branching

The **if** activity allows conditional execution. The **test** attribute must contain a script expression that evaluates to a boolean. (The **while** activity introduced earlier works in the same way). The **if** may contain either or both a **then** and **else** elements. Each contains a set of child activities that will be executed depending on the value of the test attribute.

3.5 Step

The **step** activity represents an invocation of a CEA application. Within the step element is a **tool** element, which specifies: the name of the CEA application to invoke; which interface of the application to used; and the input and output parameters to be passed.

Each **parameter** element contains a **name** and **value**. Parameters may either be *direct* – in which case **value** contains the data to be passed to the CEA application; or *indirect* – in which case **value** contains a URI reference to a resource (typically a MySpace file) that contains the actual parameter value. This is indicated by setting the **indirect** attribute to `true`. This is analogous to the distinction between 'pass by value' and 'pass by reference' in many programming languages.

For output parameters that are *indirect* the value of the parameter specifies the location to store the result. However, output parameters that are *direct* cause the CEA server to return the associated result directly to the JES. The **result-var** attribute can be used to access results that have been returned directly. It names a workflow variable in which to these results are to be stored (as a map). This allows results to be passed to subsequent steps; used in script expressions in loops and conditionals; or be processed further by script blocks

To dispatch the application, the JES passes the entire **tool** document fragment (coloured teal in the figure)to a CEA server[2]. Any script expressions in this fragment (for example in parameter values) are evaluated into strings by JES before being passed to the CEA server – hence the workflow variable environment does not need to be accessible from the CEA server. Indirect parameter values are passed to the CEA server by reference – the CEA is responsible for retrieving the resources pointed to by these parameters.

Details of the CEA server used, start and finish times, and the messages received from the CEA server are all recorded by JES in a **step-execution-record** element that it adds as a child of the **step**. This forms part of the transcript for this workflow. A **step** may be executed more than once in a workflow, for example, if it occurs in a loop: a new **step-execution-record** is added for each execution.

```
<step name="example-of-calling-swarp" result-var="results">
  <description>with different kinds of parameters</description>
  <tool name="ivo://org.astrogrid/swarp" interface="simple">
    <input>
      <parameter name="config_file">
        <value>
         WEIGHT_TYPE NONE
         RESAMPLE    Y
        </value>
      </parameter>
      <parameter name="Image" indirect=”true”>
        <value>${imgFileReference}</value>
      </parameter>
    </input>
    <output>
      <parameter name="IMAGEOUT_NAME" indirect="true">
        <value>
          ivo://uk.ac.le.star/nw#results/image${i++}.fits
        </value>
      </parameter>
    </output>
  </tool>
</step>
```

Example 2: a step that calls SWARP

2 Applications can also be executed outside a workflow by passing a **tool** document to the server.This allows invocations to be developed and executed in isolation and then reused by inserting them into workflows.

The example step illustrates many of the features mentioned. The step calls SWARP – the name attribute of the tool element contains the IVO identifier of the registry entry for the CEA application that publishes SWARP. Examining each of the parameters in turn:

- `config_file` – a direct input parameter whose value is given inline.
- `Image` – an indirect input parameter. Its value is a script expression that refers to the workflow variable `imgFileReference`. This variable should be declared earlier in the workflow, and assigned to the URI of the location to retrieve the image from.
- `IMAGEOUT_NAME` – an output parameter that specifies where to store results – in this case the value is a URI that points to a location in the MySpace for the user `nw`. The URI contains a script expression referring to a variable `i` – assuming this variable was initialized earlier, this will write the output to a different file each time the step is executed, with `i` being incremented as a side-effect.

Finally, the results of the execution will be assigned to the workflow variable `results`. As this step lacks any direct output parameters, `results` will not contain actual image data – instead, the `IMAGEOUT_NAME` key will contain the URI where the data has been stored. In this case, this is the same as the value of the `IMAGEOUT_NAME` parameter, but with the script expression evaluated.

3.6 Error Handling

When an error occurs during the execution of an activity, the normal flow of control is interrupted. The error is recorded in the transcript and then propagates upwards. If the exception reaches the root workflow element the entire workflow is judged to be in error, and is halted.

The workflow schema defines a **try** element that can be used to wrap activities and intercept errors. There is also a **catch** element, which can be used to define activities to execute only when an error occurs. However, these have not yet been implemented with JES, and are silently ignored.

3.7 Scripting

The **script** activity contains a block of script program code. Unlike a **step** activity, which executes on a CEA server, a **script** is executed in the JES. A **step-execution-record** element is added to the transcript to record any output generated when the code is executed.

The code may reference and modify the contents of workflow variables. It may also define local variables and functions. However, these are only available to the script itself - they are not visible to subsequent script blocks or expressions. Hence any result that is to be accessed later should be stored in a previously-defined workflow variable.

Scripting adds a great deal of versatility to the workflow system – and is necessary to overcome the lack of data models describing the inputs and outputs

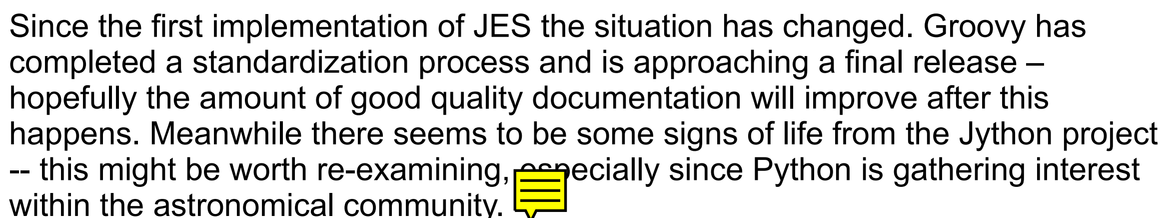
of each application[3]. Common uses for scripts include extracting columns from tables; converting representations of dates; converting between VOTable and other table formats; and simple computation, such as summing columns in a table.

Scripts can be used to manage MySpace files – for example a workflow could include a script block that tidies up some of the intermediate results produced. More advanced scripting – such as searching VO registries, or querying Cone or SIAP services – is also possible. From scripts it is possible to import and use any Java class available on the classpath. Within JES, scripts can access useful libraries for tasks such as networking, XML parsing, and VOTable manipulation[13]. In addition, the script environment contains some pre-initialized system objects that are always available[14]. Scripts can use these to interact with the current workflow, the JES Server, and other AstroGrid components.

The language used in script blocks and expressions is Groovy[15] - a dynamic soft-typed language that runs on the JVM. Java expressions and statements are valid in Groovy scripts – this subset of the language is sufficient for most purposes and will be familiar to anyone with experience of Java / C / Javascript. Groovy adds features native syntax for collections, closures, internal iterators, and regular expressions inspired by similar features in Python[16] and Ruby[17].

The choice of scripting language was constrained by the environment it had to execute in – the cross-platform Java virtual machine – and the ease of integration with the JES implementation. At the time that JES was being implemented, Groovy was the clear choice. The most suitable alternative, a JVM-based version of of an old version of Python named Jython[18] was not being actively supported or developed. Javascript was another alternative, although the language lacks a standard library of utility types and functions.

Since the first implementation of JES the situation has changed. Groovy has completed a standardization process and is approaching a final release – hopefully the amount of good quality documentation will improve after this happens. Meanwhile there seems to be some signs of life from the Jython project -- this might be worth re-examining, especially since Python is gathering interest within the astronomical community.



4 User Environment

The user interacts with the AstroGrid Workflow system using a graphical desktop application[19],[20]. This is not an integral part of the system - there is also a scripting interface[21], and previously a web-browser interface was used[4]. It is hoped that this decoupling will allow different user interfaces, that for example

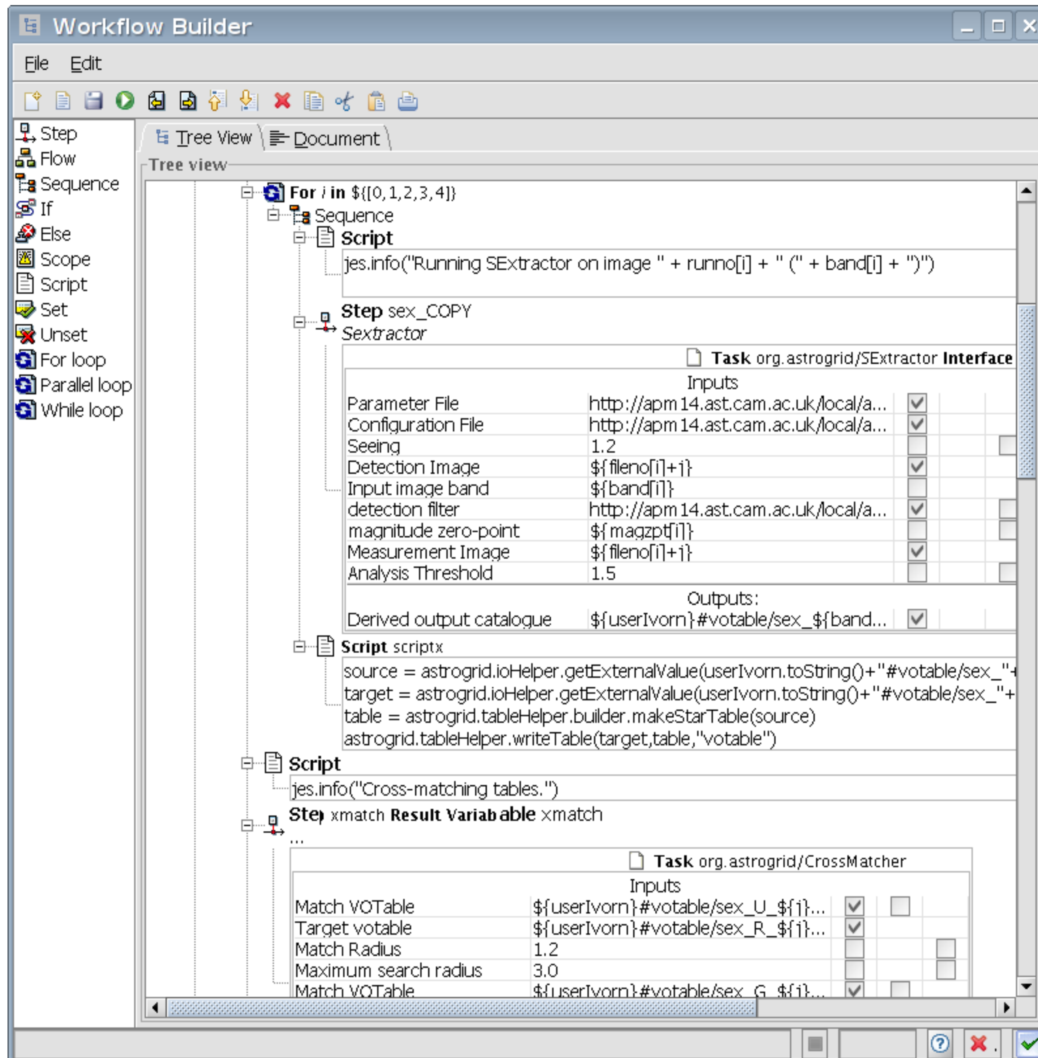
3 If data models were available, it would be possible for the workflow system to do most of the data conversion tasks that scripts are currently required for.

4 The browser solution suffered from poor usability due to the page-request model of web applications. Recent development of web technologies such as 'Single Page Applications' and AJAX may now make a browser interface feasible.

shield the user from more of the details of workflow, to be introduced to the system in the future.

Any workflow user interface will need to support the following operations: workflow construction; workflow submission to a JES; progress monitoring; and accessing results. However, there is plenty of space for innovative approaches to these tasks. The following sections describe the current user interface, which is quite explicit in presentation of the details.

4.1 Workflow Builder



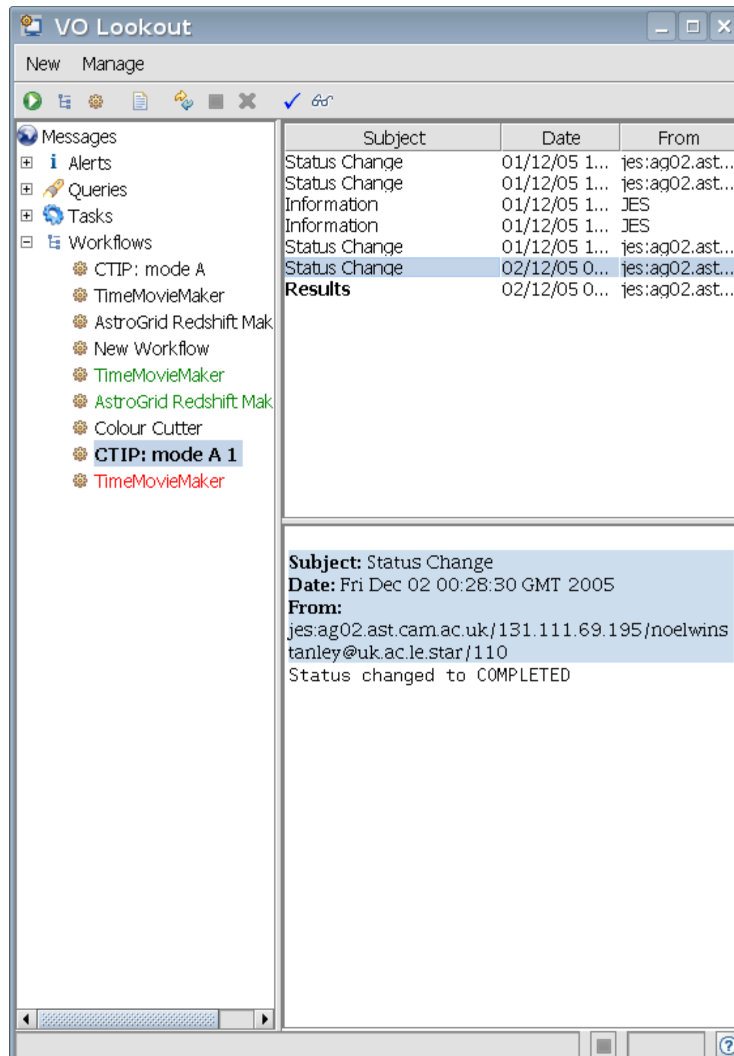
The *workflow builder* is used to construct a new workflow: either from scratch, or by loading a previous workflow as a template. The screenshot shows part of the redshift maker parameterized workflow.

The builder displays the activities in the workflow in a folding editor. New activities may be inserted by dragging and dropping them from the palette into the workflow. Pop-up dialogs prompt the user to input the required details for each activity – these are triggered by inserting a new activity, or double-clicking

on an existing activity. In the case of calls to CEA applications, the dialog's fields and documentation are dynamically generated from meta-data retrieved from the registry.

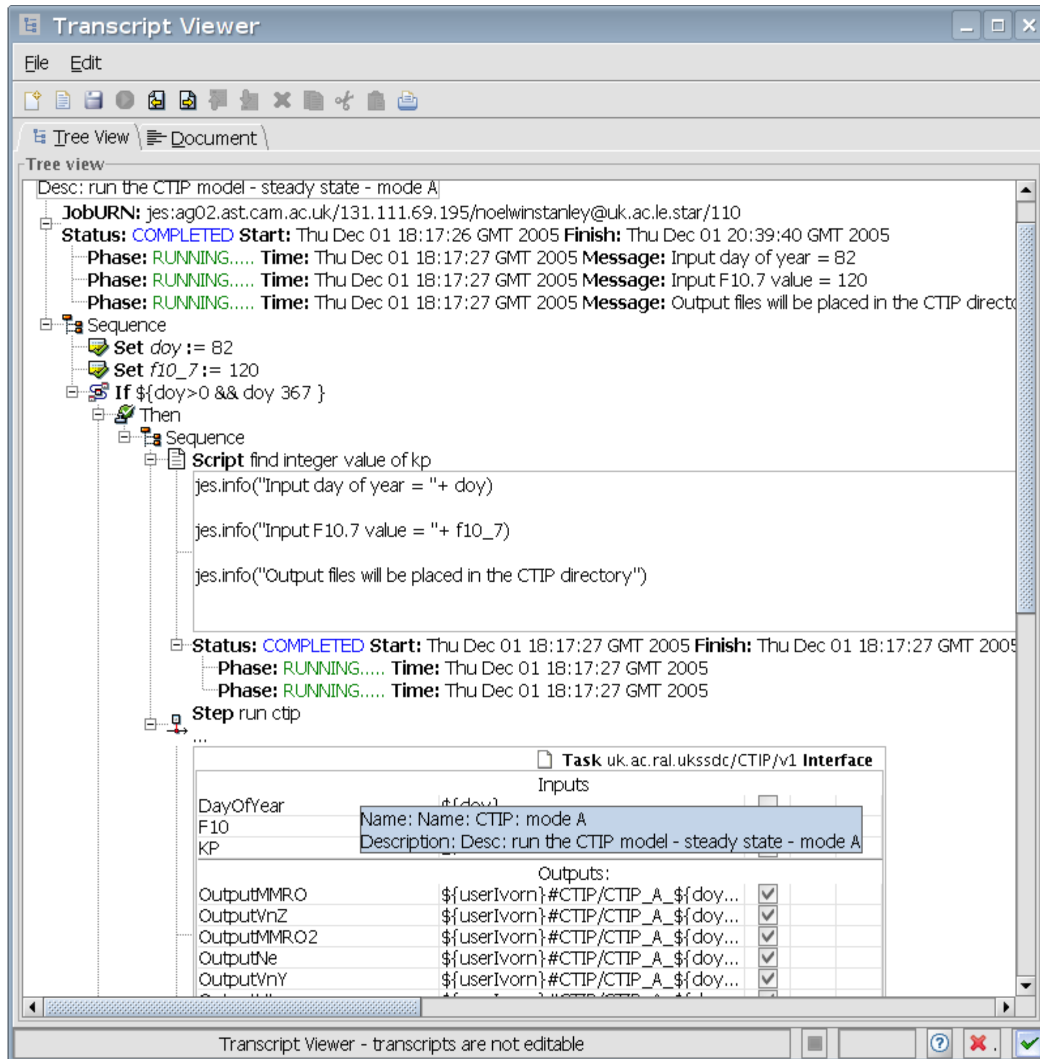
The editor ensures that an schema-invalid workflow is never constructed. It also performs static checking on any embedded scripts. The constructed workflow can be saved to storage or submitted straight to a JES.

4.2 Progress Monitor



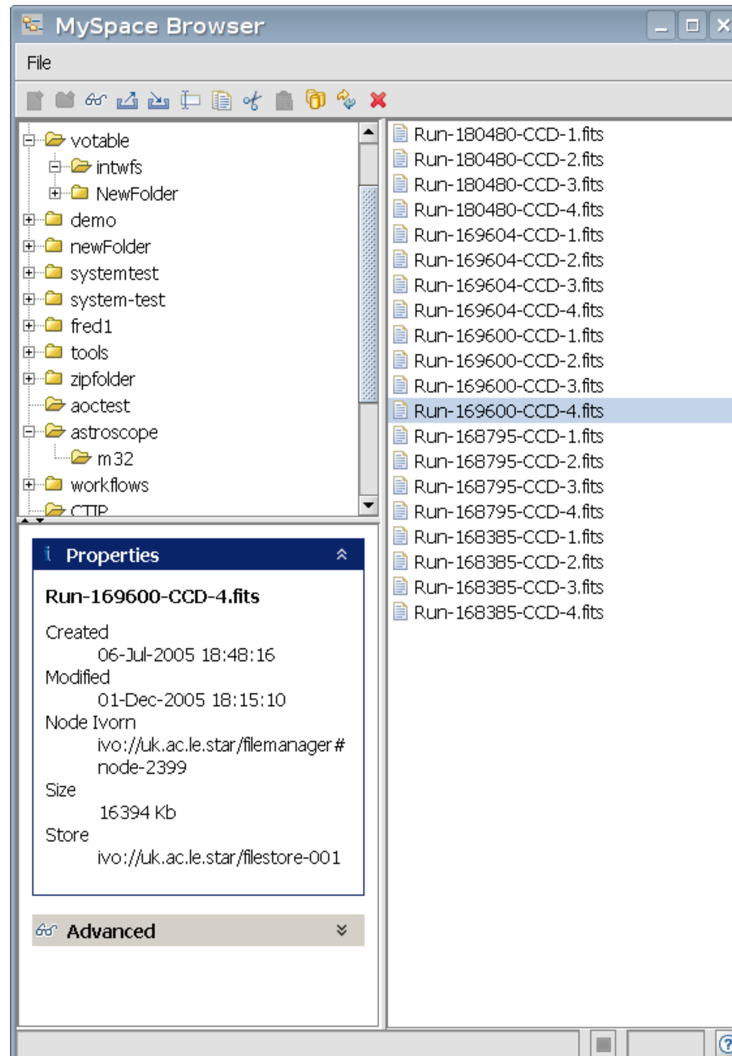
When the user submits a workflow to a JES for the *progress monitor* appears. This interface uses the metaphor of an email application. The user's jobs are represented as 'folders', which are colour coded according to their status - whether they are pending, running, completed, or in error. Selecting a job from the folder list displays a list of the message 'headers' contained in that workflow transcript. In turn, selecting a message header displays the content of the message.

4.3 Transcript Viewer



If the user wishes to examine the execution of the workflow in more detail than the progress monitor allows, the workflow transcript can be displayed in the *transcript viewer*. This is a a read-only variant of the workflow editor that displays the original workflow plus the progress messages that JES annotates it with during execution.

4.4 MySpace Browser



Once the workflow has completed the results can be accessed using the *MySpace browser*. This is a simple file-manager style interface that allows MySpace files to be managed in the usual way, relocated to a different server, and downloaded to local storage.

5 Implementation of JES

The current JES is implemented in Java and is packaged as a *web archive*. This makes it straightforward to deploy and configure in a servlet container, such as provided by Apache Tomcat[22]. JES can concurrently process multiple workflows, from multiple users. The state of each workflow is frequently persisted to disk as XML: this ensures that the server can resume workflow processing even after a system restart.

JES consists of an assemblage of the components shown in the figure below. The server exposes two web-services, **Controller**[23] and **Monitor**[10][11] through which external actors may interact with JES.

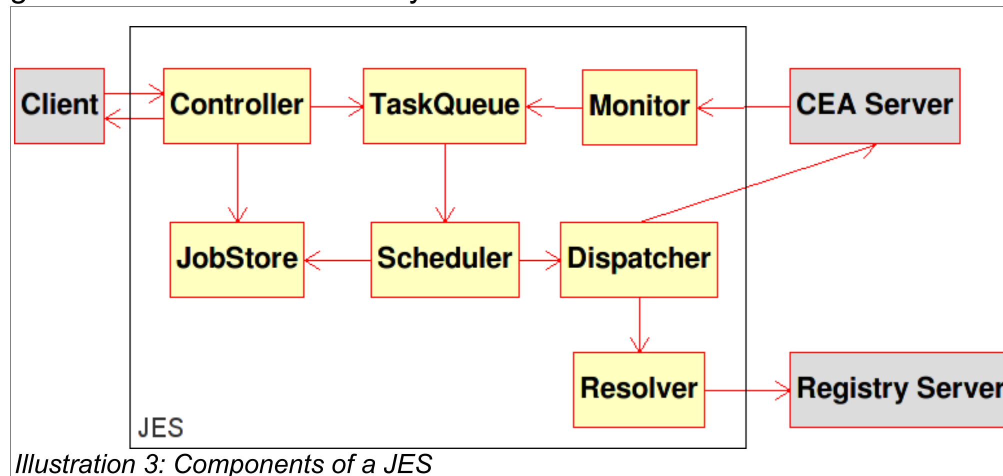


Illustration 3: Components of a JES

The **Controller** webservice handles interaction with the user. It supports operations to submit a workflow for execution; list and retrieve transcripts for a user; and cancel the execution of workflows. It processes requests from clients in one of two ways. Requests that cause no change to the server state -- such as retrieving a transcript from the **JobStore** -- are processed directly in this component. Requests that cause a change to the server state – such as submitting or canceling a workflow — are validated and then placed onto the **TaskQueue**.

The **Monitor** webservice operates in a similar way, but handles progress notifications from CEA servers. The notification messages are validated, and then placed onto the **TaskQueue**. Unlike a user client calling the **Controller**, the notifying CEA server expects no response – a *one-way* message is being sent.

JobStore is a persistent store for workflow documents annotated with transcript and state information. Documents are retrieved from the store by their unique Job ID. Interestingly, this component provides no file locking or transactional support, and there is no synchronization or locking logic in the other components – the design of the server avoids the need for these as no concurrent writes or race conditions can occur.

This property is due to the **TaskQueue**, which queues state-changing operations. This component is required because both the web services are by necessity *multithreaded*: each must be able to accept multiple requests simultaneously from different clients. The implementation of **TaskQueue** uses a standard channel-based concurrency library, and in effect merges concurrent streams of operations into a single sequence of tasks.

5.1 Scheduler

The sequence of tasks is consumed in a separate thread by the core of the system: a single instance of **Scheduler**. There are a range of different kinds of task placed on the queue – however, the most interesting are an *execute workflow* task from the **Controller** and a *notification* task from the **Monitor**.

To process the first kind of task, the workflow must be compiled from it's XML representation into a set of rules, where each rule has a *trigger* and a *consequence*. If the trigger is 'true' the rule becomes *valid* and may be fired: this causes the consequence to occur.

Workflow compilation is implemented using a XSLT transformation that converts the workflow document into a Groovy script that defines a datastructure containing a set of rule objects – the *rule-set*. The trigger of a rule object is a Groovy expression, and the consequence is a block of Groovy code. Some of the simpler activities, such as **set,** translate into a single rule. Composite activities, such as **for** generate more than one rule. **parfor** is especially complicated, as the workflow environment has to be cloned into a set of parallel environments, and then later merged.

The rule triggers typically refer to a tree of states that model the execution progress of each branch in the workflow document – another Groovy datastructure. Consequence blocks may: call internal routines on the JES server to perform operations such as dispatching steps to CEA servers; examine the *state-tree* to determine which course of action to take; and update values in the state-tree to record progress in the workflow.

The rule-set, state-tree and a freshly initialized workflow environment are attached to the original workflow document, and the whole ensemble persisted as XML in the **JobStore**. The rule-set, state-tree and environment are meanwhile parsed by an embedded Groovy interpreter – this creates a native Java datastructure from the text script that contains the rules, state and environment for the workflow.[5]

To process the second kind of task from the queue (a notification from CEA), the workflow ensemble that the notification refers to is retrieved from the **JobStore**. Again, it is parsed using Groovy into a Java datastructure. The notification message is then added to the workflow transcript. Furthermore, If the messagei was a notification of an application completing, or in error, the corresponding branch of the state-tree is updated to reflect this.

---

5 In principle it would be possible to compile the workflow directly to Java source – however, this would require a separate pass to then compile the sources into java classes before they could be loaded. Generating script text that is then interpreted has the same effect, but is simpler. Furthermore, the textual representation of datastructures in a scripting language makes it simple to persist the workflow state – this would be harder if going direct to Java code, where state cannot as easily be serialized to a string representation. This 'soft' approach also makes it easier to prototype new activities in the workflow language.

For both kinds of task, the scheduler now possesses a datastructure of rules and associated state for the workflow. The scheduler enters into a loop, selecting rules that are active and firing them until no active rule are left. The firing of rules causes activities in the workflow to be executed – which involves changes to the state-tree – which in turn may activate more rules. When no active rules remain, all control-paths through the workflow have either completed, or are blocked on notification from CEA servers. At this point the workflow ensemble is persisted back to disk, to be reloaded next time a notification is received.

Note that although the rules are processed sequentially, the workflow itself proceeds in parallel. This is because the rules may concurrently dispatch more than one CEA application.

The remaining components in JES are the **Resolver**: which queries the registry to convert an abstract application name into a URL endpoint of a CEA server that provides that application; and **Dispatcher** which performs the mechanics of invoking the CEA application.

6 Challenges

Considering JES is the first operational workflow system for the virtual observatory, it has been surprisingly successful. However, in the course of observing astronomers use the system, several shortcomings have become apparent. The major objections involve scripting - the amount of scripting required to produce an useful workflow; the difficulty of development; and the choice of scripting language.

At the moment the inputs and outputs of applications are described in little more detail than 'a VOTable'. Once datamodels become available that describe the contents of these tables, in many cases a workflow system should be able to automatically transform data between steps. However, some ad-hoc transformations requiring input from the user will always be necessary: and scripting is a flexible and familiar, although explicit way of expressing this.

Reducing the amount of scripting required would also simplify workflow development . JES should be extended so that common scripting idioms, such as iterating over the rows in a table or transforming tables, are captured as new workflow language constructs. Other programming language features such as subroutines, parameter passing, and assertions would also be useful additions to the workflow language. Finally, the workflow language must be able to express invocations of other standard service types – like SkyNode, SIAP, and UWS – in the same way it expresses CEA applications. These extensions to the workflow language would mean the user could accomplish more in the graphical workflow builder before dropping down to scripting.

Another cause of development difficulties is the batch architecture of the system; although this has benefits at run-time. This becomes apparent when using an unfamiliar set of libraries within a script; the common user behaviour is to

experiment to see what works. However the slow feedback cycle of a batch system is quite different from the interactive prompt of IDL or Python, and makes experimentation much more awkward.

To ameliorate this, we plan to integrate our client-side scripting environment (the Astro Client Runtime[21]) into JES. This means the same programming interface and libraries will be available on the interactive client-side and the batch server-side. This will allow the user to experiment and develop scripts on the client before entering them into a workflow document. It also provides a more integrated migration path for astronomers who start working client-side but then wish to share some of their process as a workflow. Finally, if feasible, we may embed a cut-down version of the JES engine within the ACR: this would give a framework for a more interactive workflow development environment, although it would lack many benefits of running workflows on a dedicated server.

If the quantity of required scripting is reduced, and the development simplified, the scripting language used becomes less important. At the simplest level, function calls, assignment and variables are pretty much the same in any language. The main weakness with Groovy at the moment is the poor documentation. It assumes a level of familiarity with Java first, which isn't common for astronomers. Although it is hoped the documentation will improve soon, other scripting languages, such as Python, should be investigated again. Ideally a JES should be able to accept workflows containing scripts in a variety of languages, just chosen by user preference.

Another weakness of the current JES is it's poor error handling and recovery. It assumes a reliable message transport for notification from CEA servers; this is not the case for web-services operating over the internet. Logic needs to be added to JES to enable it to recover from missed messages – for example by requesting message resend from CEA servers – and to detect when a CEA server has become unavailable using the proposed IVOA Support Interfaces[24]. A related aspect of this is implementing the exception handling part of the workflow language, so that the user can express what is to happen when services fail.

Finally, it has become clear that a workflow engine is just another remote application of the virtual observatory. It makes little sense to define a separate interface for it when it can be expressed in a standard way in the CEA or UWS frameworks. Fitting JES into the CEA framework has some other benefits: a uniform user interface, and it becomes trivial for workflows to submit further workflows as subroutines.

7 Suggestions for IVOA

To be useful a workflow system must have a wide selection of applications that can be used as building blocks. The development of JES has demonstrated the importance of standards for how remote applications are described and invoked. A virtual observatory workflow engine should be expected to support all the

service standards agreed by the community, but not arbitrary web-services. This is because the meta-data provided about a service in just a raw WSDL is not rich enough – it does not describe the semantics of the service functions, or the protocol of calling them in the correct order. The workflow engine cannot be expected to have this knowledge for each ad-hoc webservice, but it is feasible to encode this knowledge for each standard. The IVOA process for data access standards is proceeding well – but effort needs to be spent on finalizing and adopting the UWS proposal.

Devising a standard workflow language, however, is a futile task. Users always seem to resist using standardized programming languages[6], which tend to be more complex and harder to implement than languages designed by small groups. (see PL/1 [25]). We should recognize that different languages are suitable for different problems and classes of user. The AstroGrid workflow language is imperative and quite low-level: it would be useful to be able to compare it against prototype workflow systems using other computational models, such as dataflow (flow charts), logic (rule base), declarative query (maybe a distributed variant of ADQL), or functional. Developing workflow systems using higher-level models may also stimulate and guide the creation of appropriate data models.

The JES server could be used as a underlying platform for other workflow languages – it's an open-source product that can be extended by any interested group. A more agile approach would be to have a client-side user tool that builds a workflow in its own computational model, which is then transformed down to an imperative AstroGrid Workflow before being submitting to an unmodified JES. The AstroGrid Workflow language is well suited to be an underlying implementation language, being able to express distribution and parallelism, whilst being extensible via scripting. An advantage of this approach is the speed of implementation, and that many of the existing user interface applications could be reused.

References

1 Guy Rixon, 2005 *Software Architecture of AstroGrid v1* http://software.astrogrid.org/developerdocs/astrogrid-v1.1-architecture.htm
2 Noel Winstanley, 2005 *Parameterized Workflows* http://wiki.astrogrid.org/bin/view/Astrogrid/ParameterizedWorkflows
3 Anita Richards, 2005 *Colour Cutter User Documentation* http://wiki.astrogrid.org/bin/view/Astrogrid/ColourCutter
4 Eduardo Gonzalez, 2005 *Redshift Maker User Documentation* http://wiki.astrogrid.org/bin/view/Astrogrid/RedshiftMaker
5 Guy Rixon, *Universal worker service* http://www.ivoa.net/internal/IVOA/IvoaGridAndWebServices/uws.html
6 Dave Morris, 2005 *VOStore* http://www.ivoa.net/internal/IVOA/IvoaGridAndWebServices/vostore-0.18.pdf

6 although this doesn't apply to *query* languages, where standardization has been well received.

7 Masatoshi Ohishi and Alex Szalay, *IVOA Astronomical Data Query Language* http://www.ivoa.net/Documents/latest/ADQL.html
8 François Ochsenbein, *VOTable Format Definition* http://www.ivoa.net/Documents/latest/VOT.html
9 Paul Harrison, 2005 *A Proposal for a Common Execution Architecture* http://www.ivoa.net/Documents/latest/CEA.html
10 Noel Winstanley, 2004 *JobMonitor WSDL* http://software.astrogrid.org/schema/jes/JobMonitor/v1.0/JobMonitor.wsdl
11 Noel Winstanley, 2004 *Result Listener WSDL* http://software.astrogrid.org/schema/cea/CEAResultsListener/v1.0/CEAResultsListener.wsdl
12 Guy Rixon, *Index of AstroGrid Schemata* http://software.astrogrid.org/schemata/index.html
13 Mark Taylor, *Starlink Tables Infrastructure Library* http://www.star.bristol.ac.uk/~mbt/stil/
14 Noel Winstanley, 2004 *JEScript Object Model* http://software.astrogrid.org/components/jes/api-reference.html
15 Guillaume Laforge, *Groovy home page* http://groovy.codehaus.org
16 Guido van Rossum, *Python home page* http://www.python.org/
17 Yukihiro Matsumoto, *Ruby home page* http://www.ruby-lang.org/
18 Brian Zimmer, *Jython home page* http://www.jython.org
19 Noel Winstanley, 2005 *AstroGrid Workbench User Documentation* http://software.astrogrid.org/userdocs/workbench.html
20 AstroGrid, 2005 *Webstart the Workbench* http://software.astrogrid.org/jnlp/astrogrid-desktop/astrogrid-desktop.jnlp
21 Noel Winstanley, 2005 *Astronomy Client Runtime API* http://www.astrogrid.org/maven/docs/HEAD/desktop/multiproject/acr-interface/apidocs/index.html
22 Apache Software Foundation, *Apache Tomcat* http://tomcat.apache.org/
23 Noel Winstanley, 2004 *Job Controller WSDL* http://software.astrogrid.org/schema/jes/JobController/v1.0/JobController.wsdl
24 William O'Mullane, Guy Rixon, Ani Thakar, 2005 *IVOA Support Interfaces* http://www.ivoa.net/internal/IVOA/IvoaGridAndWebServices/VOSupportInterfaces-0.24.pdf
25, *Wikipedia: PL/1* http://en.wikipedia.org/wiki/PL1