ATLAS Job Transforms: A Data Driven Workflow Engine

G A Stewart[1,2], W B Breaden-Madden[2], H J Maddocks[3], T Harenberg[4], M Sandhoff[4] and B Sarrazin[5]

[1]CERN, CH-1211, Geneva 23, Switzerland
[2]University of Glasgow, Glasgow G12 8QQ, Scotland
[3]University of Lancaster, Bailrigg, Lancaster, Lancashire LA1 4YW, UK
[4]Bergische Universität Wuppertal, Gaußstraße 20, 42119 Wuppertal, Germany
[5]Universität Bonn, D-53012 Bonn, Germany

E-mail: `graeme.andrew.stewart@cern.ch`

Abstract.
The need to run complex workflows for a high energy physics experiment such as ATLAS has always been present. However, as computing resources have become even more constrained, compared to the wealth of data generated by the LHC, the need to use resources efficiently and manage complex workflows within a single grid job have increased.

In ATLAS, a new Job Transform framework has been developed that we describe in this paper. This framework manages the multiple execution steps needed to 'transform' one data type into another (e.g., RAW data to ESD to AOD to final ntuple) and also provides a consistent interface for the ATLAS production system.

The new framework uses a data driven workflow definition which is both easy to manage and powerful. After a transform is defined, jobs are expressed simply by specifying the input data and the desired output data. The transform infrastructure then executes only the necessary substeps to produce the final data products. The global execution cost of running the job is minimised and the transform can adapt to scenarios where data can be produced along different execution paths. Transforms for specific physics tasks which support up to 60 individual substeps have been successfully run.

As the new transforms infrastructure has been deployed in production many features have been added to the framework which improve reliability, quality of error reporting and also provide support for multi-process jobs.

1. Introduction
The evolution of high energy physics computing in the LHC era has been towards an increasingly distributed environment. ATLAS [1], for example, uses a Tier-0 centre [2] that is centrally operated at CERN and has a computing grid of 10 regional Tier-1 facilities and around 100 smaller Tier-2 sites. This has increased the complexity of computing, both in bookkeeping and in the overheads associated with job submission.

In order to manage ATLAS jobs on the grid and at the Tier-0, a set of scripts were developed to help configure the main ATLAS offline computing framework, Athena [3]. These scripts simplify the configuration of jobs at the Tier-0 and in the grid production system [4], ensure that errors from Athena are consistently reported and provide a validation of the outputs from the jobs. These scripts are known as *Job Transforms*.

Content from this work may be used under the terms of the Creative Commons Attribution 3.0 licence. Any further distribution of this work must maintain attribution to the author(s) and the title of the work, journal citation and DOI.
Published under licence by IOP Publishing Ltd

In this paper we describe the role that the transforms play in ATLAS and how the new job transform framework better fulfils the needs of the experiment.

2. ATLAS Job Transforms

2.1. Job Configuration

ATLAS offline software uses Gaudi [5], a well established software framework that has been developed by ATLAS and LHCb. Gaudi utilises a python based job configuration system, which is extremely flexible. However, there is no built-in facility for converting command line options to python configuration files and the actual job configuration may become very large and unwieldy. One of the first roles of the job transform, then, is to provide a mechanism by which jobs may be configured via command line options and that the overall configuration of the job may be simplified. e.g., a job transform should allow a job to be specified in terms of a simple input and output file such as:

```
Reco_tf.py --inputBSFile data12.1234.RAW --outputESDFile data12.1234.ESD
```

This is then converted to a python job options file such as

```
runArgs.inputBSFile = ["data12.1234.RAW"]
runArgs.outputESDFile = "data12.1234.ESD"
```

In addition to these simple options, the actual transform (in this case `Reco_tf.py`) has a job options template (or skeleton) associated with it. This template encapsulates all of the standard job options necessary to reconstruct a RAW detector file to an ESD (Event Summary Data) file – in this case the template used is `skeleton.RAWtoESD_tf.py`. These templates allow most of the complexity of standard jobs to be hidden and to be updated by experts as necessary, quite independently of the normal simple use cases.

2.2. File Validation

As part of managing the processing of a job, the transforms offer the ability to test and validate input and output files. This is because errors in file access can be difficult to diagnose inside the offline software and, once a file has been written, it is useful to do a check that actually reads the file back off the storage system.

The transforms should offer the ability to do both fast event count checks and full ROOT scans of files. As the value of such checks must be balanced against the cost of doing them, in practice ATLAS makes a fast check on input files and a deeper check of output files.

2.3. Error Handling

HEP software implements complex algorithms in a sophisticated framework with a considerable number of external dependencies (storage systems, database access). This provides a multiplicity of ways in which jobs can fail, both in terms of the component that can have produced the failure and the underlying cause of the failure itself. From an operational point of view it is essential when triaging failures that these error messages are clearly reported and categorised, so that errors that need expert attention can be dispatched appropriately.

The transform framework must provide a clear way to generically categorise failures (core dump, file corruption, etc.) and also offer a facility for passing on a detailed error message.

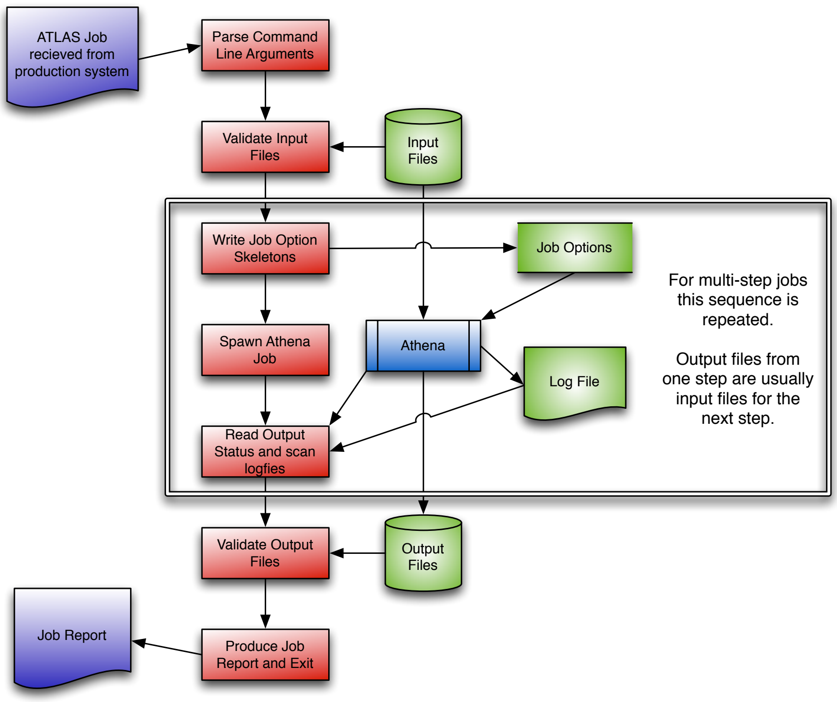
ATLAS Job recieved from production system
Parse Command Line Arguments
Validate Input Files
Input Files
Write Job Option Skeletons
Job Options
For multi-step jobs this sequence is repeated.
Output files from one step are usually input files for the next step.
Spawn Athena Job
Athena
Log File
Read Output Status and scan logfies
Validate Output Files
Output Files
Job Report
Produce Job Report and Exit

Figure 1. Overview of transform workflow

2.4. Metadata Handling
All jobs, successful or failed, produce metadata. For failed jobs the most important metadata is the failure messages (handled as per section 2.3). For successful jobs, it is important to propagate metadata upstream to the ATLAS Metadata Interface [6] (AMI). Such metadata can concern the job configuration, the execution of each substep and the output files that are produced. For the output files the number of output events is critical metadata, but other data, such as conditions and geometry setup, is very useful for final physics analysis.

2.5. Workflow Overview
Figure 1 gives an overview of the workflow for a job transform. An important feature of this workflow is the ability to run multi-substep jobs, where the outputs from one substep are fed as inputs to a subsequent step. At the end of the chain final data products are produced from the original inputs and the intermediate files may either be ephemeral or may also be final job outputs.

3. Old Transforms Framework
Originally job transforms were implemented as a simple set of shell script wrappers around jobs at the ATLAS Tier-0. As transforms proved useful there was a new version written in python.

This framework was run for a number of years in production for ATLAS and was successful in processing many millions of jobs. However, the maintenance of the framework became increasingly problematic as the original developers moved on to new projects. In particular, the implementation suffered from a number of specific problems:

- Extensive use of *implicit* methods in the framework, e.g., the execution of a transform was triggered implicitly by retrieving its exit code. This made tracing the way in which the

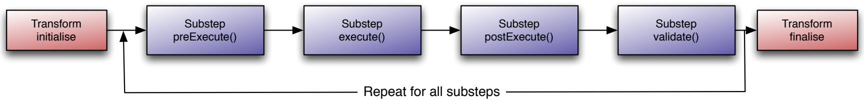

Figure 2. Internal workflow of the new transform framework

transforms worked more difficult.

- Accumulation of redundant code no longer relevant to ATLAS's running conditions, e.g., code to stage data from tape systems at CERN.
- Deep inheritance for classes that introduced unnecessary complexity into the framework.
- No builtin support for jobs that need to execute multiple substeps, with multi-step jobs implemented as a customised extension in a separate module.

4. New Framework

Increasing problems in maintaining the transforms framework and a significant backlog of open bugs led to a review of the future of the transforms in ATLAS. This review concluded that the old framework should be frozen (aside from essential bug fixes) and the development of a new framework embarked upon. The advantages of such an approach were:

- The successful transforms model, which is well integrated with other code, was maintained.
- The risks of extensively modifying the old framework were ameliorated by freezing it during data taking in 2012.
- The old framework lacked the capacity to cope with the important use case of multi-step jobs, with no clear way to introduce it.
- The new transforms could be developed in parallel and then introduced gradually, allowing for a phased deployment.

4.1. Fundamental Design

The guiding design for the new transforms was to make the fundamental workflow shown in figure 1 as explicit as possible. The development was also guided by agile programming principles, in particular meaning that development should be focused in solving actual problems for the collaboration instead of trying to cover all possible use cases from the outset.

Internally the transform adopted a workflow adapted from the classic HEP event processing model of `initialise()`, `eventLoop()`, `finalise()`. However, in this case, the event loop is replaced with the execution of an individual substep. These substeps themselves have a well defined workflow: `preExecute()`, `execute()`, `postExecute()` and `validate()`. The transform internal work flow is shown in figure 2.

In every case where functionality was required standard python modules were used if possible. Thus all logging in the new framework is done through the standard python `logging` module. Argument parsing is handled by the `argparse` module, with some customised extensions needed by the transforms (e.g., returning multiple value arguments as single class instance with an array as its 'value' instead of an array of single valued instances, which is the default behaviour).

4.2. Error Handling

As dealing clearly and consistently with failures during execution is extremely important, the decision was made to consistently use exceptions to handle errors internally. The transform exception classes have two internal data members: an error number, which is the exit code for

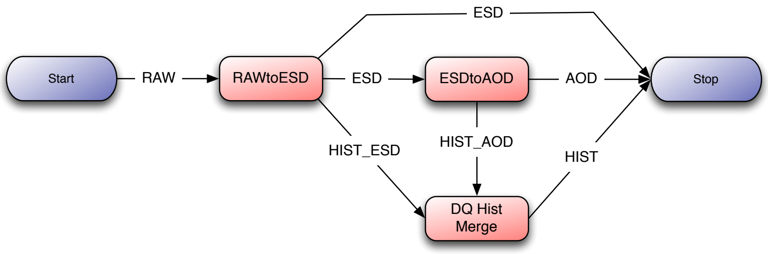

Figure 3. A graph representation of a three step reconstruction transform. The individual substep nodes are in red and the edges show data flow. Note that in this case ESD is a final data product, so it flows into the *Stop* node; however, HIST_ESD and HIST_AOD are ephemeral and are only used to make the final data quality histograms (HIST), then they are discarded.

the transform if this exception is unrecoverable; and an error message, which gives as many details as possible about why this exception was raised and will be used by shifters and experts to analyse the problem.

The allows the monitoring components that are used by shifters to provide good quality information regarding general error categories (exit codes) and specific information regarding failure reasons (exit messages).

Special handlers for Athena logfiles are also implemented that ease the extraction of key information about execution failures without the need for Athena itself to implement complex error message reporting.

4.3. Graphs and Multi-Step Jobs

As pointed out in section 2, support for multi-step jobs is a key use case for job transforms. Such use cases are characterised by the flow of data out of one substep and into the next. This gives rise to the natural representation of a multi-step job as a graph, where nodes are substeps and edges are data products [7]. A basic illustration of a multi-step workflow is shown in figure 3. Transform graphs are directed and acyclic.

A specific transform has a representation of the graph of all possible data flows and substeps as part of its definition. For a developer, input and output data types are simply lists of strings, making them very easy to use. (When multiple inputs are all necessary as a substep's input a tuple of strings is passed instead.) The transform framework then examines the *input* and *output* data for a specific job and determines which substeps need to be executed. An example of a reconstruction job that runs three substeps is shown in figure 3. In that example if histograms (HIST data type) had not been requested then the production of HIST_ESD and HIST_AOD would have been suppressed, as would the execution of the DQ Hist Merge substep.

4.3.1. Solution Algorithm The problem that the transform faces is not the classic problem of tracing the shortest single path through a graph (solutions to these problems are well known). Instead the transform must trace a path for *each* data type that is a final output and minimise the *overall* cost of execution.

To do this, the following algorithm is applied:

(i) The substep nodes are topologically ordered (listed in such a way that all edges point in one direction).
(ii) All data types are then also topologically ordered, by their earliest appearance in the graph.
(iii) Input data is flagged as *available*.

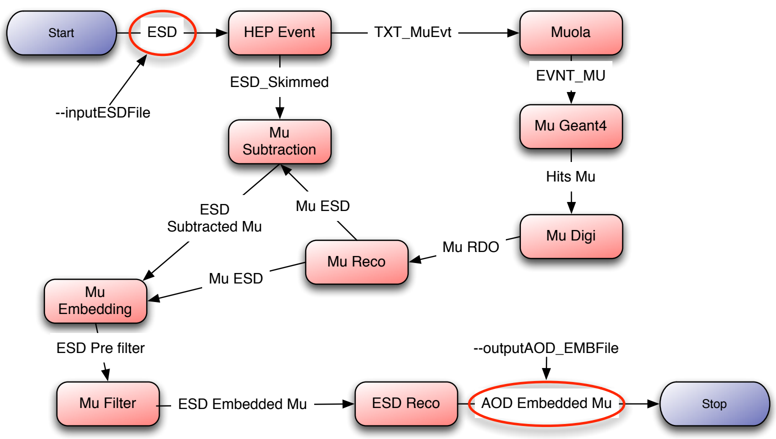

Figure 4. An extract from the $H \to \tau\tau$ embedding workflow.

(iv) Final output data is flagged as *required*.

(v) The *required* data products are considered in topological order (i.e., data produced earlier is considered first). For each needed data product:

(a) All paths that lead from a desired data product to an available data product are considered – as the graph is directed acyclic this is a linear time operation. (This allows transforms to be defined where data can be produced in different ways, the cheapest way will be chosen, which may vary depending on the set of output data required.)

(b) The cheapest path is set to *activated*, meaning that the cost of executing a substep on this path is now set to zero for any subsequent data production.

(c) If any additional data is required on this path it is added to the list of required data.

(d) The considered data type is now moved from the *required* list to the *available* list.

When all data is in state *available* and the *required* list is empty then the graph tracing algorithm can stop and provide a list of

(i) substeps that need to be executed;

(ii) data products that need to be produced (as substeps can produce more than one data product it is necessary to know this for the correct substep configuration).

If the graph algorithm finds a data type that it cannot produce it raises an exception.

4.3.2. Performance and Scaling This algorithm is not guaranteed to find the cheapest execution cost in all cases; however, it does find the best path for all workflows in ATLAS.

The most complex workflow considered so far is that to support the embedding of simulated $H \to \tau\tau$ decays into real data di-muon events. There are nine possible primary output data types, depending on the τ decay mode, and the transform contains 60 substeps (an extract of the workflow is shown in figure 4). Even in this complex case the graph algorithm runs in about 250ms, compared with a job execution time of around 20 hours.

4.4. Testing and Quality Control

Throughout the development of the new transform framework an extensive set of tests was developed. These included module unit tests and longer running end to end tests of important workflows. All of these tests, however, run within the python `unittest` framework, ensuring that they can be run easily by developers on the command line (before code is committed), by the nightly test framework [8], just after the transform packages are built, or by the more extensive ATLAS Run Time Tester framework [9].

4.5. Enhancements and New Features

The redesign of the transform framework has made it significantly easier to add new features to the workflow of particular transforms, by having specialist executor classes. This cleanly separates specialist requirements from the transform core. Features can be added to any of the `preExecute()`, `execute()`, `postExecute()` or `validate()` steps.

An example of such a feature is the ability to switch the version of Athena in which a particular substep executes. This allows ATLAS to, for example, re-run the trigger using an old release of Athena that may correspond to particular data taking period of interest.

Another example is special handling of the multiple outputs from AthenaMP [10] job (grid jobs where Athena runs in a multi-process mode). Here the transform can specially merge files at its discretion, improving job efficiency and output file size. As there is huge scope for flexibility within an executor, development is underway to ensure that merging occurs in parallel, which will further improve cpu to walltime efficiency for these jobs.

4.6. Metadata

Historically Tier-0 and the ATLAS grid production system used two different formats for transmitting metadata upstream. At Tier-0 a pickled python dictionary was employed, in contrast to the XML used in grid production. As part of the job transforms review and rewrite it has been agreed to unify these reports. The reports themselves are nested python dictionaries, structurally mirroring the transforms' internal workflow and JSON was chosen as a convenient serialisation technology. The quality of information transmitted has been improved and allows downstream components to gain knowledge of each substep's execution and resource utilisation.

In order to smooth the transition to the new framework a backward compatible version of both the pickle and XML job reports is available, allowing the deployment of the new transforms to be decoupled for downstream client dependencies.

5. Summary and Conclusions

A new job transforms framework for ATLAS has been developed. This framework has a considerably simplified design and clearer workflow that has helped ATLAS software developers write transforms for many new use cases. The new framework is already in production for many use cases and is simplifying computing operations and improving ATLAS's ability to flexibly exploit computing resources at Tier-0 and on the grid.

References

[1] The ATLAS Collaboration 2008 *Journal of Instrumentation* **3** S08003
[2] Elsing M, Goossens L, Nairz A and Negri G 2010 *Journal of Physics: Conference Series* **219** 072011
[3] Calafiura, P *et al* 2005 *Computing in High Energy Physics 2004,* Interlaken
[4] Golubkov D *et al* 2012 *Journal of Physics: Conference Series* **396** 032049 URL `http://stacks.iop.org/1742-6596/396/i=3/a=032049`
[5] Barrand G *et al* 2001 *Comput.Phys.Commun.* **140** 45–55
[6] Albrand S, Doherty T, Fulachier J and Lambert F 2008 *Journal of Physics: Conference Series* **119** 072003 URL `http://stacks.iop.org/1742-6596/119/i=7/a=072003`
[7] Wikipedia Graph theory (Retrieved 2013-12-10) URL `http://en.wikipedia.org/wiki/Graph_theory`
[8] Undrus A 2003 *Computing in High Energy Physics 2003,* San Diego URL `http://arxiv.org/pdf/hep-ex/0305087.pdf`
[9] Simmons B, Sherwood P, Ciba K and Richards A 2010 *Journal of Physics: Conference Series* **219** 042023 URL `http://stacks.iop.org/1742-6596/219/i=4/a=042023`

[10] S Binet *et al* 2012 *Journal of Physics: Conference Series* **368** 012018 URL `http://stacks.iop.org/1742-6596/368/i=1/a=012018`