Types of Editors and Specifications

James Blythe, Richard Cavanaugh, Ewa Deelman, Ian Foster, Seung-Hye Jang, Carl Kesselman, Keith Marzullo, Reagan Moore, Valerie Taylor, Xianan Zhang

December 2003

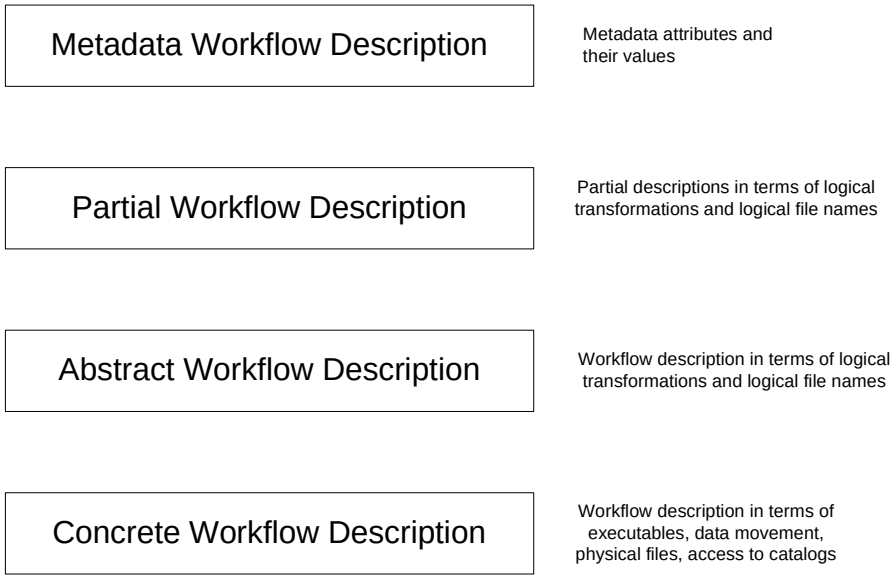

Figure 1: Various levels of workflow from the most abstract down to the most concrete.

1. Metadata Resolver

Input: Metadata Workflow Description

Output: Abstract Workflow Description

Functionality: The resolver takes a metadata workflow description and based on the knowledge of application components (transformations) and the required data, generates an abstract workflow identifying the input data, transformations and their dependencies necessary to produce the requested data. The experimental version of Pegasus that targets the LIGO application is able to map between various types of requests (pulsar search, short Fourier transform, and time interval) and produce an abstract workflow which includes all the steps necessary for the computation.

Use case scenario: A user provides a high-level data description, such as conduct a pulsar search over a particular time interval in a particular frequency range and particular location in the sky. The editor takes that description and produces an abstract workflow that contains the description of the Fourier transforms and pulsar search codes that need to be applied to the raw data.

Notes: The metadata workflow description could also be called a "concept-based workflow". The workflow at that level requires an understanding of the relationships between the physics names used to describe a process, the collections that contain the associated data, and the transformations that create the desired physical data quantities. These are expressed using concept spaces. The concept spaces define semantic terms, and either logical relationships between terms ("isa", "hasa"), or procedural relationships (generation of flux from velocity and density), or structural relationships (how to build a mosaic given images from different sky surveys).

2. Abstract Workflow Generator

Input: Partial Workflow Description, Data Product Name

Output: Abstract Workflow Description

Functionality: The generator takes a partial description and a logical name of a desired data product and connects the necessary partial workflows into a full workflow descriptions where all the transformations and necessary files are identified by their logical names. An example of such a generator is Chimera, which takes the partial workflows expressed in VDL and a logical file name and returns a full abstract workflows, which includes all the names of all necessary logical input files and logical transformations and their dependencies.

Use case scenario: A user provides the VDL, that contains all possible partial workflow description and the logical file name, Chimera finds the appropriate derivations and chains them together to form an abstract workflow.

Note: It could be desirable to be able to include access to collections as part of the workflow. The query to the collection generates a result set that is then processed as a dataflow through specified services. This means that the file names are not known a priori. An implication is that the dataflow can loop over the files in the result set, that conditional execution is supported (not all files may be processed by each step of the workflow), that state information is maintained about each element in the result set.

3. Graph Evaluator

Input: Abstract Workflow Description

Output: Abstract Workflow Description (annotated)

Functionality: Using the request submitter's distinguished name, policy information, resource descriptions, and grid weather, a scheduled completion time is estimated for each node and a resulting completion time for the graph estimated within some confidence level. We can use systems such as Prophesy to predict the expected completion time of each node in the workflow or of the entire graph on all the available resources. A QoS confidence level can be estimated by computing the probability that the graph will complete on or before the requested deadline.

Use case scenario: A "Production Group" submits a large workflow (graph) for execution. In order that the produced data is available to scientists in time for winter conferences, the Production Manager would like to know the current probability estimate that the workflow will be able to finish before 31 December.

Note: We should be able to execute the workflow using available grid services. This may make the performance prediction harder or impossible as the services may perform their own optimization. In case of grid services, the graph evaluator might focus on the identification of clusters within the dataflow related to operations on data from the same collection, and then call an aggregate web service on that cluster of data files or result sets. The invoked service would then plan the execution of the files in the cluster.

4. Graph Predictor

Input: Abstract Workflow Description

Output: Abstract Workflow Description (annotated)

Functionality: Using either the request submitter's stated resource requirements (if given), or domain knowledge (like performance as a function of input parameter, I/O types, etc), or statistical resource usage (such as benchmarking, or provenance information from previous executions, profiling, etc), the Graph Predictor estimates the resource usage (within some confidence level) for every node as well as the graph as a whole. Initial resource usage predictions may include the number of nodes and disk space required to run the abstract workflow. The confidence level of declared resource usage requirements might be considered very high (e.g 100 %), whereas historical information might be considered with high or low confidence, depending on the size and character of the statistics gathered. The confidence level may be established based on the accuracy of the analytical model used to estimate the resource usage.

Use case scenario: A user generates an abstract workflow (either indirectly or directly) and wishes to estimate how much resources are required to successfully complete the workflow execution.

5. Graph Aggregator

Input: Partial (or Abstract) Workflow Description

Output: Modified Partial (or Abstract) Workflow Description

Functionality: The Aggregation Editor examines the graph for sub-graph regions which are logically associated according to execution locality, data locality, or possibly a submitter supplied aggregation metric. All nodes within some local distance would be clustered into an aggregate node within the output "aggregated" graph. Each aggregate node would be treated as a single entity by subsequent editors, but internally consist of the sub-graph corresponding to the aggregated region in the original graph.

Use case scenario: Several nodes are connected in a pipeline-like sub-graph. By treating the pipeline-like part of the graph as an aggregated node, use of bandwidth resources can be minimised by taking advantage of data locality.

6. Graph Reducer

Input: abstract workflow specification

Output: abstract workflow specification

Functionality: It reduces the abstract workflow based on some algorithm. One algorithm could be based on the availability of intermediate data products. In this case

the editor would assume that it is more efficient to access available data than to recompute it (*). An example of such an editor is Pegasus that takes an abstract workflow and queries the RLS to find out which data products specified in the workflow can be already found in the Grid. The node whose output files have been found are removed from the workflow and their antecedents (if applicable) are removed. During the reduction process, the benefit of accessing the data versus regenerating it needs to be examined to evaluate whether the assumption * is accurate.

Use case scenario: An abstract workflow of the form "a -> b -> c" is given to the system. The graph reducer finds out that the data produced by b is already available and reduces the workflow to "c".

7. Graph Expander

Input: Partial (or Abstract) Workflow Description

Output: Modified Partial (or Abstract) Workflow Description

Functionality: The Expansion Editor examines the graph for sub-graph regions which may be parallelised according to execution or data parallelism, or possibly a submitter supplied expansion hint. Once identified, the expansion editor replicates the sub-graph (or job) one or more times.

Use case scenario: A huge-throughput graph for Monte Carlo Event Simulation is presented to the Expansion Editor. Using domain specific expansion hints, the editor determines that the graph may be replicated many times over, taking advantage of the current grid weather as well as the parallel properties of Monte Carlo Event Simulation.

8. Graph Rewriter

Input: Partial (or Abstract) Workflow Description

Output: Modified Partial (or Abstract) Workflow Description

Functionality: It is possible that an editor is neither a graph expander nor a graph reducer. For example, a wide-area mater-worker may

9. Graph Partitioner

Input: Abstract Workflow Specification

Output: A set of Abstract Workflow Specification and dependencies between them.

Functionality: The idea behind the partitioner is to divide the graph into sections which can be scheduled. The paritioner allows for setting the planning horizon for the following editors such as the Concrete Workflow Generator. The Partitioner takes an abstract workflow graph and partitions it into subgraphs, maintaining data and computation dependencies between the subgraphs as shown in the Figure below. The subgraphs can then be edited by other editors that take an abstract workflow as an input.

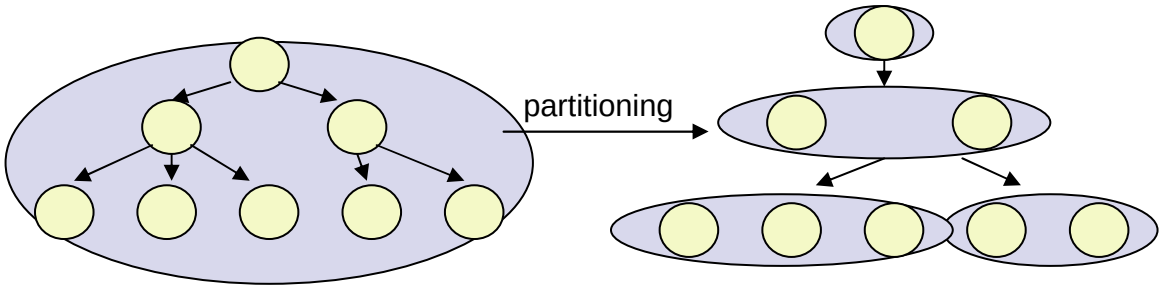

Figure 2: An example of partitioning the abstract workflow.

Use case scenario: Because the state of the Grid can change frequently, it is often beneficial to schedule only portions of the workflow at a time. Given a large workflow, one would partition it into smaller workflows and schedules the top most nodes then as these get executed, schedules the graphs dependent on the executed graphs.

10. Concrete Workflow Generator

Input: Abstract Workflow Specification

Output: Concrete Workflow Specification

The Concrete Workflow Generator takes an abstract workflow description (possibly already reduced by another editor and generates a workflow which can be executed on the Grid. An example of such a generator is Pegasus, which locates the resources that can execute the transformations, selects the replicas of data to access. Additionally, Pegasus schedules the necessary data movement for staging data in and out of the computations. Additionally, it augments the workflow with the registration on new data products into the Grid.

Use case scenario: Let's assume we have a workflow containing a single transformation (A) and it takes a file f_a as input and produces f_b as output. The editor will produce a concrete workflow that moves f_a to resource X schedules (A) to be executed at X, stages f_b from X to the user-defined location and possibly registers f_b in RLS.

Note: We may also want to specify control operations on the workflow beyond the simple mapping of output files to input files. Examples are error handling, partial completion, looping, gather/scatter, etc.

11. Graph Execution

Input: A concrete workflow

Output: A workflow whose nodes are marked "done" or "failed"

Functionality: This editor takes a concrete workflow description and executes the nodes in the workflow on the resources specified in the nodes in the order specified in the workflow. If failures occur the executor can retry the computation (or data movement) or it can mark the node as failed. An example of such an editor is DAGMan, which takes a concrete workflow in the form of a DAG and executes the operations specified in it. The DAGMan can also do last-minute node scheduling. If the DAG fails, DAGMan returns a rescue DAG containing the nodes still to be executed.

Use case scenario: Continuing from the previous example, DAGMan will move f_a to X, schedule the execution of A on X and move f_b to the user-defined location.

Note: We may need to add control structures to the execution.

12. Graph Recorder

Input: Concrete Workflow Description

Output: Completed Concrete Workflow Description annotated with Provenance Information

Functionality: This editor takes a concrete workflow description and records provenance information about the environment and conditions in which each node executes.

Use case scenario: In order to debug a problem, or validate a workflow, a user who submitted perhaps a workflow description would like to know the environment and conditions in which the workflow actually executed.

Note: We need a more general mechanism that will allow state information generated about the execution of each process within the dataflow to be mapped into an appropriate context catalog.

An example is the use of a dataflow environment to support curation of documents for a digital library. We would like to map the state information onto the attributes that are used by the digital library to assert provenance. A similar mapping is needed for the application of archival processes. We need to map state information that is generated by the processing onto the authenticity attributes managed by the persistent archive.