# DØ Job Definition

**3/25/02**

**D. Meyer, K. De, M. Sosebee, T. Wlodek, J. Yu,**
**University of Texas at Arlington**

## 1. Motivation

In the Work Plan for DØ Grid of 12/7/01 (distributed Jan 2002), section 2 calls for a job description language for DØ jobs. As a starting point for such a language, in terms of what it has to "say", consider the following definition of a DØ job. This definition is presented in the form of the database schema that could be used to store the job in a database. It describes a logical job and its physical components. If this schema, or some version of it, can be accepted as a basic component in the Work Plan, then efforts can be commissioned to create such a database while ongoing parallel efforts are used to schedule and complete the jobs in such a database. UTA would be interested in working on the job creation tasks.

The schema was derived from an analysis of current DØ tasks that included the Monte Carlo binaries and root, and was further modified to handle the anticipated tasks of pre-fetching data files, pre-fetching software, and purging software. To this point, all tasks appear to fit into a simplified DAG that contains only linear chains, merges, and splits. For example, the following logical-job DAG (or any segment of it) can be incorporated into the proposed database schema:

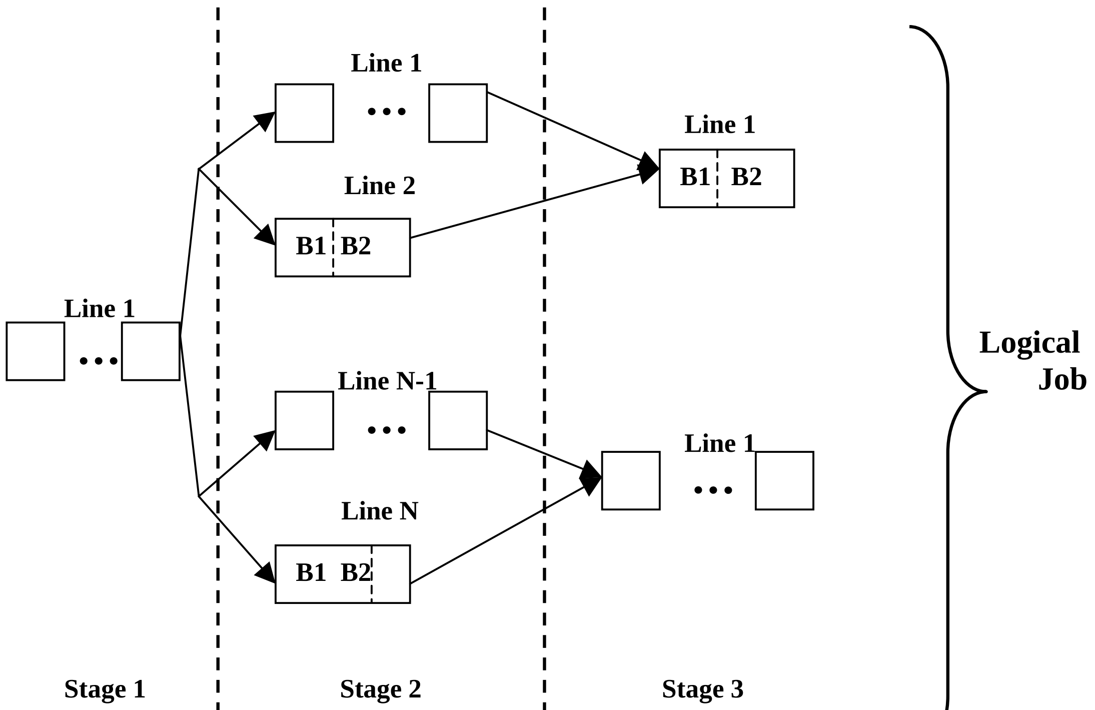

**Figure 1 A logical job DAG**

The vertical dashed lines denote a change of *stage* of the job. The arrows that cross stage boundaries represent datafile forwarding. Within a stage there are one or more independent and possibly dissimilar *lines* of tasks. Within a line, each *package* (a rectangle in Figure 1) is performed in sequence, and can be individually scheduled and possibly restarted. If restart is to be traded-off for efficiency, then within a package there can be multiple *chained binaries* that are scripted.

In this discussion, the following glossary will be used:

- **Job**: a logical DØ job, one task from the viewpoint of the researcher
- **Stage**:a segment of the job that has different parallelization than the prior segment (e.g., it denotes a *split* or a *merge* of parallel tasks).
- **Line**: a set tasks to be performed in sequence, possibly on different execution nodes, where the tasks are implicitly connected by the dataset(s) they produce/consume
- **Package**: a physical DØ job (a rectangle in the diagram), the atomic unit of scheduling. Always contains a command and its command-line arguments.
- **Binary**: a single executable (binary code or script) with its command-line arguments. Could be constructed one per package (in which case its command and arguments are the same as in its package), or could be chained under the control of a script (as on the rightmost package above, and the command and arguments are assumed to be contained in the package script).

The job schema is described in the next section in a skeleton form (the elements thought to be essential to scheduling). A brief discussion of how scheduling would use this database is presented after that, because this structure must ultimately be adequate for scheduling. The next section presents a set of use cases to show how to populate this database structure for typical DØ jobs. Additional use cases are solicited for this treatment. The last section addresses error recovery and restart.

## 2. Job Schema

This section contains a description of the records and data elements that would be stored in a database for a job. Many additional data elements will eventually be added to support scheduling, monitoring, accounting, and other issues.

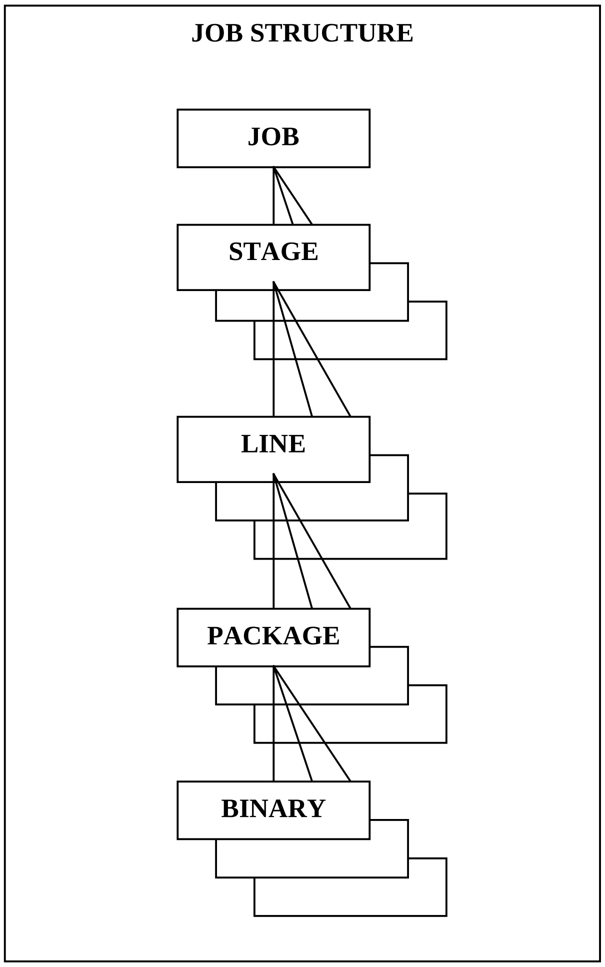

**Figure 2 A Job Structure DAG**

**Job Record** (One per logical job)
Job ID – unique, assigned by job construction, probably includes part of researcher's name
Originator ID (researcher)
Origination Date & Time
Priority – policies regarding priority probably need to apply at the logical job level
Hold (yes/no) – in case jobs need to be defined but not yet scheduled

**Stage Record** (At least one per job, more if parallelization differs)
Job Parent – the logical job parent
Stage ID – sequentially assigned within a job, 1 to n.

**Line Record** (At least one per stage. Only one if the binaries are not parallelizable)
Stage Parent – the stage parent

Line ID – which line within this stage, 1 to n.
Early-start (yes/no) – if yes, then the first package of this line may be started without waiting for its input, if any, to be completely available.

**Package Record** (At least one per stage. This is the *indivisible unit* for scheduling and running)
Line Parent – the line parent
Package Sequence – processing order of this package within this line, 1 to n.
Status – unscheduled, scheduled, launched, successful, failed, canceled
Failure History – a list of failures, each containing:
Reason code/message (see section 5)
Count of number of failures of this type
Startup Script ID – if there is a wrapper script for this package, then this ID specifies its ID (as it was registered in a code server database). Any such wrapper script would have to be platform-independent, and its name would change if it incurs a version change. A wrapper script is required if there are multiple binaries in this package, optional if there is only one. The physical job will either be this script or the (single) binary's invoking script itself.
Startup Script Parameter Set – if there is a wrapper script, this element contains its input Parameters (typically the names of the binary's scripts it is to run)
Node Restriction List – if the originator requires the package to run on one of a specific set of nodes, then this list identifies them.
Scheduled Node – the node where this packages is actually scheduled to run or did run.
Platform ID – the type of platform of the scheduled node. Each binary in this package must have a version from the code server for this platform.

**Binary Record** (At least one per package. More if binaries are chained.)
Package Parent – the package parent
Binary Sequence – processing order of this binary within this package.
Software ID – references a software database in which this ID identifies this binary (e.g. "reco").
Software Version – specifies which DØ version of the binary. This plus the software ID specifies The bulk of the parameter data for the binary (extracted from the code server).
SAM Required (yes/no) – does the binary require SAM station. This, together with the Software ID and Version, restricts this binary to running on specific types of platforms.
CPU Requirement – a percentage from 0 to 100 that estimates how much of a CPU this binary Will consume during normal operation. DØ binaries are typically 99%, while tasks such as software purging would be typically 10-20%.
Input Parameter Set – contains all job-specific input parameters for this binary – everything that was not specified by the software version itself.
Input File Names – list of the any and all input files required for this binary. Each list element consists of:
Base Name – the file name excluding any locators such as URL names, directories
Locator – an object that specifies where the input file will exist at run time (e.g., a URL and directory, or a SAM locator)
Size (estimated if necessary)
Acquisition (none/self/pre-fetch) – "one" means the file is supposed to be present. "self" means the binary or the wrapper script will acquire this input (using SAM or other grid tool) st startup. "pre-fetch" means grid middleware must fetch the file into the cluster's cache before launching this binary on any node of that cluster.
Disposition – action to be taken on input file. If file already exists, a "stickier" disposition here will override its present retention setting.
RETAIN INDEF – keep in cache, researcher will have to explicitly purge
RETAIN TEMP – keep in cache but allow delete if space is needed
DELETABLE – after binary is done, and not needed by other jobs
Output File Names – list of any and all output files from this binary. Each list element consists of:
Base Name – the file name excluding any locators such as URL names, directories

    - Locator – an object that specifies where the output file will exist at run time (e.g., a URL and directory)
    - Estimated size
    - Dispositions – list of actions to be taken on output file, any logical combination of:
        - FORWARD – make available to downstream binary as needed.
        - SAM STORE – invoke SAM station to store
        - RETAIN INDEF – keep in cache, researcher will have to explicitly purge
        - RETAIN TEMP – keep in cache but allow delete if space is needed
        - DELETABLE – after forwarding or storing, and after no longer needed by downstream binary (or, a copy has been forwarded), delete it (unless needed by some other job)
- Estimated size of work files
- Estimated run time if dedicated on a “standard” platform
- Estimated or Actual Start Time – using the node’s time zone
- Estimate or Actual End Time – using the node’s time zone
- Invoking Script – attaches the actual script to invoke this binary, supply run-line arguments, etc. This was obtained from the code server using software ID, version, and target platform after the target node was determined.

## Case IV. Reconstruction / Recoanalysis on Reco Farm with Explicit Store

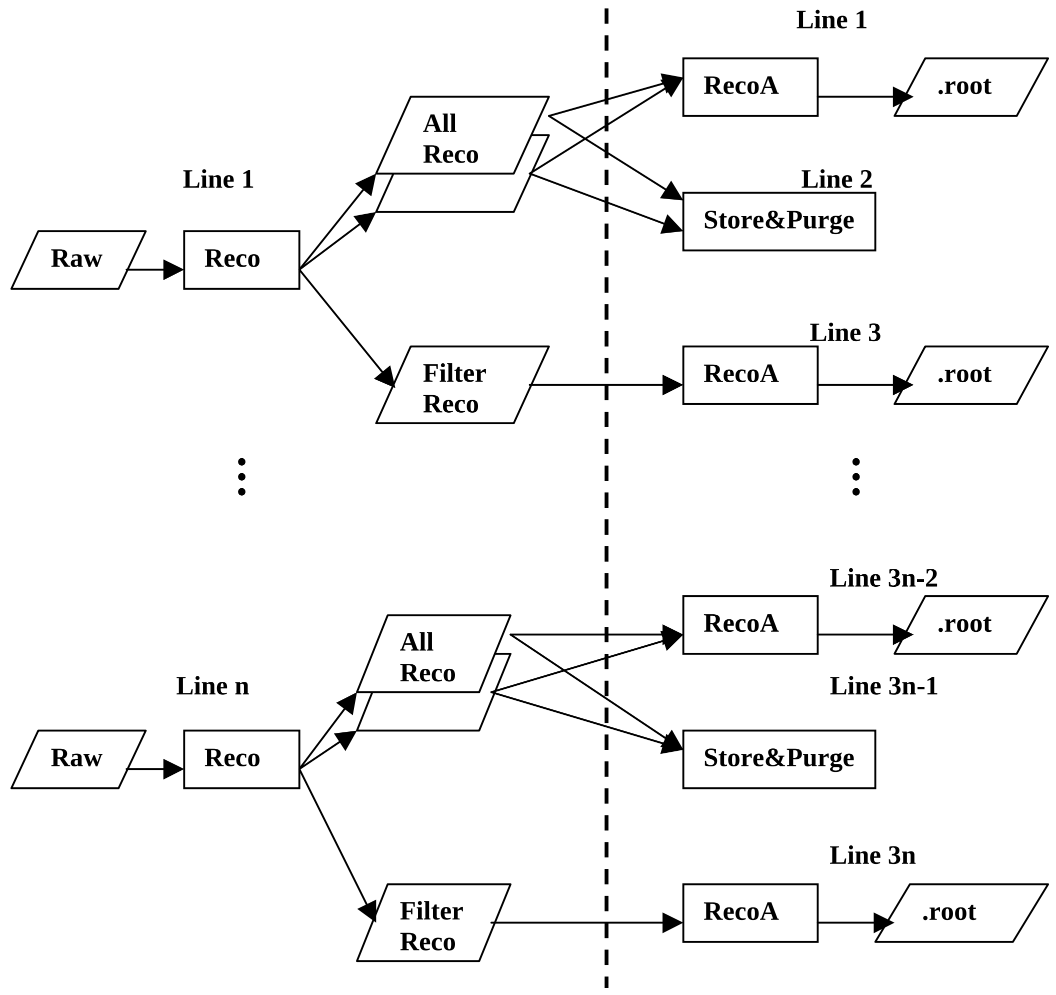

**Figure 3 Reconstruction with Recoanalysis use case**

Reco is performed on a “reco farm” for two reasons. One reason is the need to access to large databases that would be expensive to propagate to the general grid. A second reason is to facilitate the merging of many small root and filtered-reco files (produced in this use case but merged and stored by a later job). A farm allows the researcher to retain the small files indefinitely, until they can be merged, stored, and deleted.

Assumption: # 1 The many small filtered-reco and filtered-root files produced by this use case are left in the farm’s cache, and a later job is constructed to merge, store SAM, and flag these small files as deletable. The job construction task to create such a merge job would have access to the farm’s cache for filenames and sizes. This would be a simple job to make, and could easily be automated on the reco farm.

Assumption # 2: At the time of job construction for this use case, we can predict the number and names of the all-reco files to be created in stage 1. Thus, we can include these names in the filename lists at job construction time. If this is not possible, then we would need to be able to make jobs with file lists that are “not yet known”, and make the original reco package supply the filenames to the job database when it completes. This would inhibit downstream scheduling until the lists were populated, but at least in this case the number of downstream tasks is already known so stage 2 can be constructed early.

**Job Record**

Job ID – reco001
Originator ID – reco@fnal.gov
Origination Date & Time – 03/05/2001/1400
Priority – mid level
Hold - no

**Stage Record 1 (Original Reco)**

Job Parent – reco001
Stage ID – 001

**Line Record 1 of Stage 1**

Stage Parent – reco001/001
Line ID – 001
Early-start – not applicable to lines in the first stage

**Package Record 1 of Line 1 Stage 1**

Line Parent – reco001/001/001
Package Sequence – 001
Status – unscheduled
Failure history - empty
Startup Script ID – omitted
Startup Script Parameter Set – omitted
Node Restriction List – the list of nodes in the farm cluster
Scheduled Node – not yet assigned
Platform ID – not yet assigned

**Binary Record 1 of Package 1 Line 1 Stage 1**

Package Parent – reco001/001/001/001
Binary Sequence - 001
Software ID – xxx (references “reco” in the code server database)
Software Version – xx-xx-xx
SAM Required - yes
CPU Requirement – 99%
Input Parameter Set – from researcher, arguments and parameter filenames for this execution
Input File Names – list with one element consisting of:

- Base Name – raw input file 1
- Locator – not yet assigned
- Size – not yet assigned
- Acquisition – pre-fetch (assuming may now be on tape)
- Disposition – deletable (assuming it was obtained from tape)

Output File Names – list with multiple elements, one for each of the all-reco files, and one for the single filtered-reco output file, each element consisting of

- Base Name – reco file n or filtered-reco file
- Locator – not yet assigned
- Estimated Size – from history
- Estimated run time –from history
- Disposition – retain indef (will be stored and deleted by future steps)

Estimated size of work files – 10KB
Estimated run time - from history
Estimated or Actual Start Time – not yet assigned
Estimate or Actual End Time – not yet assigned
Invoking Script – attaches the actual script to invoke reco

**Line Records 2 thru n for Stage 1**

If there are n raw input files, then there will be n lines each with one package and one binary. The rest are identical to the line-1 records above except that the input and output file base names are different.

**Stage Record 2 (Recoanalyze and Store)**

Job Parent – reco001
Stage ID – 002

**Line Record 1 of Stage 2 (Recoanalyze on All Reco)**

Stage Parent – reco001/002
Line ID – 001
Early-start - no

**Package Record 1 of Line 1 Stage 2**

Line Parent – reco001/002/001
Package Sequence – 001
Status – unscheduled
Failure history - empty
Startup Script ID – omitted
Startup Script Parameter Set – omitted
Node Restriction List – the list of nodes in the farm cluster
Scheduled Node – not yet assigned
Platform ID – not yet assigned

**Binary Record 1 of Package 1 Line 1 Stage 2**

Package Parent – reco001/002/001/001
Binary Sequence - 001
Software ID – xxx (references “recoanalyze” in the code server database)
Software Version – xx-xx-xx
SAM Required - no
CPU Requirement – 99%
Input Parameter Set – from researcher, arguments and parameter filenames for this execution
Input File Names – list with one element for each all-reco file from the predecessor, each element Consisting of
- Base Name – reco input file
- Locator – not yet assigned
- Size – not yet assigned
- Acquisition - none (we assume NFS makes it available locally)
- Disposition – deletable (won’t delete until line 2 has finished with it)

Output File Names – list with one element consisting of
- Base Name – .root file
- Locator – not yet assigned
- Estimated Size – from history
- Estimated run time –from history
- Disposition – retain indef (will be merged, stored, and deleted by future steps)

Estimated size of work files – 10KB
Estimated run time - from history
Estimated or Actual Start Time – not yet assigned
Estimate or Actual End Time – not yet assigned
Invoking Script – attaches the actual script to invoke recoanalyze

**Line Record 2 of Stage 2 (Store and Purge)**

Stage Parent – reco001/002
Line ID – 002

Early-start - no

**Package Record 1 of Line 2 Stage 2**

Line Parent – reco001/002/002
Package Sequence – 001
Status – unscheduled
Failure history - empty
Startup Script ID – omitted
Startup Script Parameter Set – omitted
Node Restriction List – the list of nodes in the farm cluster
Scheduled Node – not yet assigned
Platform ID – not yet assigned

**Binary Record 1 of Package 1 Line 2 Stage 2**

Package Parent – reco001/002/002/001
Binary Sequence - 001
Software ID – xxx (references a script to store to SAM )
Software Version – xx-xx-xx
SAM Required - yes
CPU Requirement – 99%
Input Parameter Set – from researcher and job construction, arguments and parameter filenames for this execution
Input File Names – list with one element for each all-reco file from the predecessor, each element Consisting of
    Base Name – reco input file
    Locator – not yet assigned
    Size – not yet assigned
    Acquisition - none (we assume NFS makes it available locally)
    Disposition – deletable
Output File Names – empty
Estimated size of work files – 10KB
Estimated run time - from history
Estimated or Actual Start Time – not yet assigned
Estimate or Actual End Time – not yet assigned
Invoking Script – attaches the actual script to invoke SAM

**Line Record 3 of Stage 2 (Recoanalyze on Filtered Reco)**

Stage Parent – reco001/002
Line ID – 001
Early-start - no

**Package Record 1 of Line 3 Stage 2**

Line Parent – reco001/002/003
Package Sequence – 001
Status – unscheduled
Failure history - empty
Startup Script ID – omitted
Startup Script Parameter Set – omitted
Node Restriction List – the list of nodes in the farm cluster
Scheduled Node – not yet assigned
Platform ID – not yet assigned

**Binary Record 1 of Package 1 Line 3 Stage 2**

Package Parent – reco001/002/003/001
Binary Sequence - 001
Software ID – xxx (references "recoanalyze" in the code server database)

Software Version – xx-xx-xx
SAM Required - no
CPU Requirement – 99%
Input Parameter Set – from researcher, arguments and parameter filenames for this execution
Input File Names – list with one element consisting of
- Base Name – filtered reco input file
- Locator – not yet assigned
- Size – not yet assigned
- Acquisition - none (we assume NFS makes it available locally)
- Disposition – store indef (will be merged, stored, and deleted by future steps)

Output File Names – list with one element consisting of
- Base Name – .root file
- Locator – not yet assigned
- Estimated Size – from history
- Estimated run time –from history
- Disposition – retain indef (will be merged, stored, and deleted by future steps)

Estimated size of work files – 10KB
Estimated run time - from history
Estimated or Actual Start Time – not yet assigned
Estimate or Actual End Time – not yet assigned
Invoking Script – attaches the actual script to invoke recoanalyze

**Line Records 4 thru m for Stage 2**

If there are n raw input files, then there will be m=3n lines in stage 2. Each set of three lines performs the recoanalyze on the all-reco files, the store on the all-reco files, and the recoanalyze on the filtered reco file. The remaining lines are identical to the three presented above except that the input and output file base names are different.

## DAGman Instructions

In order to submit this use case to DAGman, I believe the following lines would appear in a submission file:

```
Job   Stage1Line1Package1Reco    (with processing information
Job   Stage2Line1Package1RecoA (  “  )
Job   Stage2Line2Package1Store    ( “ )
Job   Stage2Line3Package1RecoA ( “ )
PARENT   Stage1Line1Package1Reco   CHILD  Stage2Line1Package1RecoA
                                          Stage2Line2Package1Store
                                          Stage2Line3Package1RecoA
```

And then similarly for every other set of tasks for a given raw input file.

DAGman does not know about input file dependencies. The logic that constructs this explicit parent-child relationship is based on examination of the input files needed for the child tasks and the output files specified for the parent reco task. I have purposely avoided putting any explicit parent-child relationships into the job definition because it would be redundant with the relationships that are implied by the datasets. (See section 5 for why this helps restart and error recovery).

## 5. Restart and Error Recovery Considerations

### Detecting Errors

The following scenarios could occur.

1. A processing node goes down.

   Grid monitoring processes must watch for this, probably by occasional queries to the nodes. If the node does not respond for some period of time, all the packages on that node must have their status changed to “failed”, with a reason “platform failure”. An email could be sent to cluster-management.

2. A binary goes into a processing loop, or ceases to produce new output.

   Detecting this condition will require additional coding to monitor the binary, be aware of the nature of its log files and output files, and thus know when it is doing something unexpected. The binary itself may have to be changed to assist in this status monitoring. If this condition can be detected, the package should be killed and its status changed to “failed”, with a reason “looping”. The reasearcher would be notified by email.

3. A binary ends with a processing exception (seg-fault or similar), or ends without producing a specified output file.

   The OS may be able to detect a seg fault without additional coding. The scheduler knows what output files are expected and can check for them to be present. The binary has ended in this scenario, but not successfully. As in case 2, the failure reason should be stored in the binary’s record in the job definition database, and the package should be marked as “failed” with the reason “seg fault” or “missing output”. The researcher would be notified via email (as would both grid-control and cluster-management).

4. A binary outputs a particular failure message to its log file.

   An examination of the log file requires additional coding, such as a parent task wrapping the binary. Such monitoring may also be required to detect a seg-fault (since standard-error output will probably be redirected to the log file). The binary has ended in this scenario, but not successfully. Treatment would be the same as for case 3 (with the failure reason taken from the log file). There is probably no need to tell grid management or cluster management about the failure. The researcher would be notified via email.

It appears that to be able to automate error recovery to any reasonable degree, some monitor coding should be launched along with the binary. (This would make the monitoring a local, distributed task, as opposed to asking that some master monitor program look at the log files for hundreds of jobs).

## Restart

Once the scheduler has determined that a package failed, it should "clean" the job from the node and its cache before deciding whether or not to restart it.

Any downstream packages or stages that were awaiting the production of output file(s) from this failed package will naturally continue to wait, since at this point their input file lists still require the missing files.

The researcher should be notified of all failures, regardless of whether or not an automatic restart will be attempted.  At any time, the research should be able to "kill" a line of packages.  When doing so, the option should be given to also remove the output files (that are no longer going to be produced) from any downstream stage's input-file list.  In other words, drop that line and output file(s) from the job entirely.  If a downstream package or stage had been waiting for this file to be launched, it could now be launched without it (because the implicit dependence has been removed).

If the schedule does decide to attempt a restart (based on the failure type and possibly on some configuration parameter for number of retries for specific failure reasons), then it changes the package status from "failed" to  "unscheduled".    The scheduler will then accept it for restart (perhaps on the same node, which is still known from the package record).  One could make a case for giving a restart a higher priority, because it is holding up the completion of a job.  Not only is the researcher waiting for the job, but also cache may otherwise be holding files from the job overly-long.

If the scheduler decides not to restart the package, it will leave the package status as "failed".  Eventually, someone must either drop that line or manually change the status back to "unscheduled" to force a restart.

## Binary Chaining vs Restart

When constructing a line of chained tasks, one can either make them individual packages or one can chain them tightly into one package using a script.  The main reason for chaining tightly is processing efficiency: the scheduler does not have to be involved in the launching of a chained binary (with the attendant gap time).  However, this impacts the ability of the grid to restart the failed binaries.  The scheduler has to treat the script as a single unit, so a failure of binary # 3 would normally require a restart of binary # 1.  (It would be possible to code the wrapper script to do intelligent restart itself, at binary # 3 for example, and thus one could avoid the restart problem with more coding).  The job definition allows either method to be invoked.