Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, 37203

Meta-Tools For Designing Scientific Workflow Management Systems: Part-I, Survey

Tripti Saxena, Abhishek Dubey

TECHNICAL REPORT

ISIS-Report Number

April, 2011

Scientific Workflow Management Systems: Part I, Survey

Tripti Saxena and Abhishek Dubey
Institute for Software-Integrated Systems
Vanderbilt University
Nashville, TN 37203, USA

Abstract

Scientific workflows require the coordination of data processing activities, resulting in executions driven by data dependencies. Due to the scales involved and the repetition of analysis, typically workflows are analyzed in coordinated campaigns, each execution managed and controlled by the workflow management system. In this respect, a workflow management system is required to (1) provide facilities for specifying workflows: intermediate steps, inputs/outputs, and parameters, (2) manage the execution of the workflow based on specified parameters, (3) provide facilities for managing data provenance, and (4) provide facilities to monitor the progress of the workflow, include facilities to detect anomalies, isolate faults and provide recovery actions. In this paper, part-I of a two part series, we provide a comparison of some state of the art workflow management systems with respect to these four primary requirements.

1 Introduction

Workflow management systems are used for orchestrating systematic analysis of scientific problem on large computing platforms. There are several state of the art workflow management systems. However, most workflow management systems are tailored for a particular domain, with specialized requirements for process management. Furthermore, different workflow system uses different scheduling tools. Clearly, while there are common principles in all these tools it is hard to reconcile them. Furthermore, any extensions are difficult to implement.

Our core idea is to propose a **meta-framework** that provides tools necessary to create customized workflow management systems or extend existing workflow systems. However, this paper provides the review of state of the art with respect to some salient features that they are expected to support.

2 Preliminaries

Scientific workflows require the coordination of data processing activities, resulting in executions driven by data dependencies; whereas in business workflows, the control dependencies describe the processing steps, which usually are coordinating activities between individuals and systems [13]. Each processing step involves the execution of a single participant. A participant is a unit distributed computation tasks. It can be a legacy application invoked from within a scripting language program. Typically, they are parallelized over multiple computation nodes.

All participants belonging to a workflow are related to each other by control and data dependencies. The workflow is defined as tuple $W{=}(P_{ts}, I_{ps}, D_{pt})$, where P_{ts} is the set of participants, I_{ps} is the set of user input parameters and $D_{pt} \subseteq P_{ts} \times P_{ts}$ defines a set of directed dependencies between the participants. We call a workflow directed acyclic and executable if the graph induced by the set D_{pt} is a directed acyclic graph. Notice that we allow the definition of cyclic workflows. However, typically this implies the existence of a translator that can generate directed acyclic and executable workflows from a cyclic workflow definition.

Typically, there are two types of workflows: abstract and concrete. An abstract workflow is a specification that defines a directed relationship between several participants. They also describe the parameter type and the number of parameters (tokens) that are consumed by each participant. A pictorial example of a workflow is shown in figure 1, where P_t is represented by a circle of different shades for distinct participants, arrows define the data dependencies D_{ps}. The parameter set is used to generate a concrete workflow from an abstract workflow.

Concrete workflows are executable workflows generated from a given abstract workflow. These workflows are directed acyclic graphs. Given a parameter set and an abstract workflow a concrete workflow can be generated. Different workflow management systems handle these notions of abstract and concrete workflows differently.

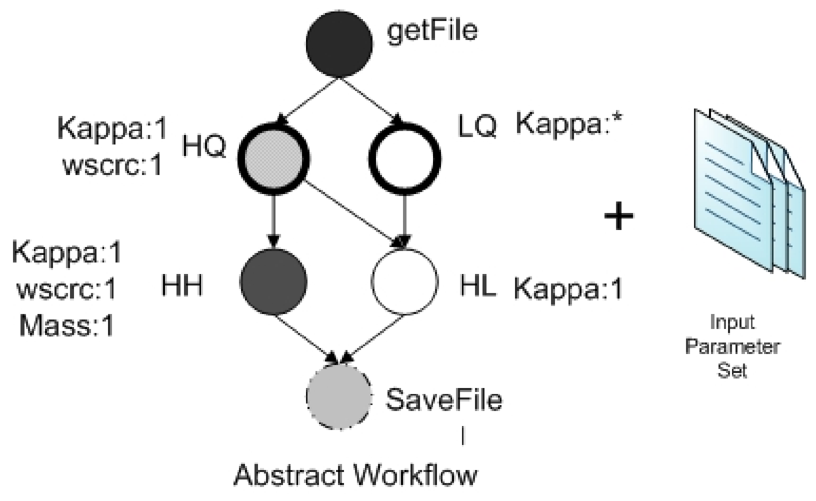

Figure 1: Sample workflow definition

3 Features of Workflow Management Systems

3.1 Workflow Lifecycle

At an abstract level a workflow can be viewed as a dataflow graph with a set of processing nodes with connections between the nodes to signify dependencies. At the execution level, however, a workflow is a computer program that is used to orchestrate the deployment and execution of a set of distributed tasks on a network. Workflow management systems provide an environment to ease the programming effort involved in the workflow execution. In order to fully understand the mechanics of a workflow system, we first talk about the life cycle of a workflow. In the following, we extend the workflow lifecycle stages considered by Ewa et. al. in [2] to include fault tolerance and use these workflow stages as a guide to compare a set of workflow management systems:

3.1.1 Composition, Representation and Data Model

In this stage the abstract workflow is composed using a graphical or textual editor. This abstract workflow might be created from scratch or by modification of a previously developed workflow. The composition can be done by using the workflow components found in the component library. The workflow description can be iteratively modified at this stage. There are different mechanisms of composing workflows as well as different formats for saving the composition.

Furthermore, there are different models for representing the workflow. These can be Directed acyclic graphs, petri-nets, UML sequence diagrams, or Business Process Models (BPMN).

3.1.2 Mapping and Execution

Mapping involves transforming an abstract workflow to a concrete workflow. This requires expanding the abstract workflow tasks into the set of concrete tasks based on the input parameter set. Also, it involves mapping of input parameters to concrete URIs. Finally, the scheduling decisions are made.

Scheduling implies task allocation, which maps each stage of the workflow to a computing node. This task allocation map is computed with respect to a scheduling policy. For example, PBS/Torque [1, 12] with Maui Scheduler [11] can be used for imposing an approximate First in First Out (FIFO) scheduling policy with some extra features such as reservations for queued jobs, the throttling rules, and the flexible priority system. See Maui documentation [11] for further details on available scheduling policies. A valid task allocation map θ has to satisfy the constraint on resources. Specifically, nodes allocated to a participant P_t must have the minimum amount of resources required

This is followed by workflow execution. Execution models can be control flow or data flow models, where control flow models represent transfer of control from the preceding task to the next and data flow model represents transfer of data from one to the next. Hybrid models represent a combination of control and data flow. Scientific workflows are typically data flows. There are two different execution semantics that can be employed to execute scientific workflows:

Batch Execution : All participants of a workflow are instantiated once and fired once. The complete workflow will finish when all participants have fired once and output product of the end stage has been recorded. Results in one set of output data product

Pipeline Execution : Similar to a Dataflow network. All participants are instantiated once together on the cluster. The participants are fired multiple times. The workflow finishes when all set of input data has been consumed. This results in multiple sets of output data product.

The choice of execution model impacts the scheduling decision.

3.1.3 Fault tolerance

Clusters built out of commodity computers, used for scientific computing, exhibit intermittent faults, which can result in systemic failures when operated over a long continuous period for executing workflows. While executing, a typical workflow can spawn hundreds of jobs. Many of these jobs are computation intensive and use dozens to hundreds of processing nodes. Typically, several users analyze different workflows on the clusters concurrently. Given the scale of numbers, identifying job anomalies, fault isolation and fault mitigation becomes critical, specifically when the success of whole workflow might be affected by even one job failure.

The workflow management needs a manager to ensure that the workflow execution can continue execution even in the presence of faults. Fault tolerance schemes are a part of the execution and mapping, but we deal with them separately to highlight its importance. Most workflow management systems at present use an ad-hoc technique for fault tolerance that is developed by the individual designers rather than a standard scheme that is a part of the system.

3.1.4 Metadata and Provenance

Like fault tolerance, provenance and data collection is also a part of execution. This stage requires recording of metadata and provenance information during the execution stage of workflow lifecycle. Provenance data can be used to record the changes at design time as the workflow evolves from one version to the other. But more importantly, provenance data is used to record the data during the execution of the workflow.

4 Workflow Systems

In this section we use the different workflow lifecycle stages presented in Section 3 to compare the capabilities of existing workflow. We extend this comparison criteria to include non-functional properties of the frameworks.

4.1 Running Example: Black Diamond

We use the black diamond workflow, shown in figure 2 as an example. This worklow has 4 nodes, laid out in a diamond shape and exchange files between them. We have written small C programs corresponding to each node of the blackdiamond. For example, the preprocess C program reads the input files designated by arguments, writes them back onto output files.

4.2 Kepler

Kepler is a scientific workflow system developed at the San Diego Supercomputer Center (SDSC) at the University of California, San Diego [5]. Kepler extends the Ptolemy II environment (`http://ptolemy.berkeley.edu/ptolemyII/`), which is based on the actor-oriented architecture. Ptolemy II allows modeling, simulation, and design of concurrent real-time embedded systems using different models of computations.

4.2.1 Composition and representation

- **Workflow Template**: Kepler supports a GUI based on Ptolemy/Vergil to create workflow templates. Kepler enables the scientists to graphically model an executable workflow by dragging and dropping components from the library. Kepler supports components called *Actors* that represent atomic operation. Kepler also

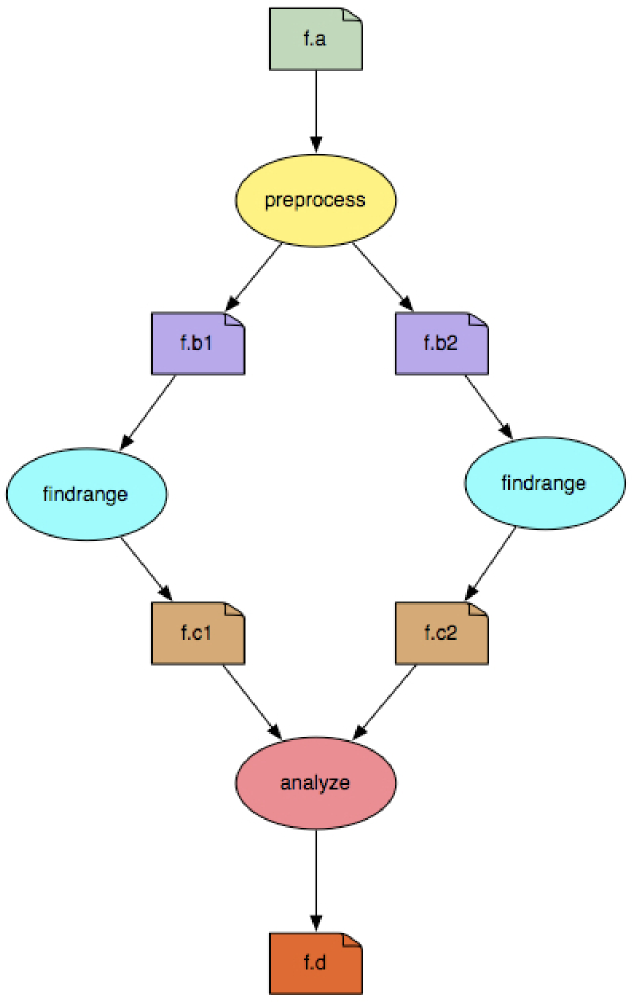

Figure 2: Sample workflow definition [7]

provides computational controllers called *Directors* that dictate how the control dependencies amongst the various *Actors* are resolved depending on the kind of director. Different *Directors* conceptually allow the same workflow to be scheduled/executed differently. Scientists can also write code to customize or create new actors more appropriate for their domain. Higher level actors can be created using *Composite Actor*. Parameterized inputs are associated with actors and directors and are usually kept fixed during a workflow execution. For example, a director can have parameters for the number of workflow iterations, criterion for each iteration, start time of the workflow. These parameters are supplied at the start of the execution of the workflow. Kepler workflows can be exchanged in XML using MoML (Modeling Markup Language) supported by Ptolemy. Kepler workflows and customized components can also be shared using the Kepler Archive Format (KAR). The current version of Kepler supports 350 actors, which includes R and Matlab actors, Web Service actor, and so on. More details on the actor library can be found in Kepler actor reference [4]

- **Workflow Instance**: A workflow instance in Kepler can be created by supplying parameter values (from the command prompt) while executing an existing workflow specification. There is a one-to-one mapping between the computational elements in the template and the workflow instance.

4.2.2 Workflow Mapping and execution

Kepler is used for workflow prototyping and has two set of actors that can submit jobs to two typical distributed resources: Cluster and Grid. Each set has some actors which can be used for different job operations: create, submit, status check.

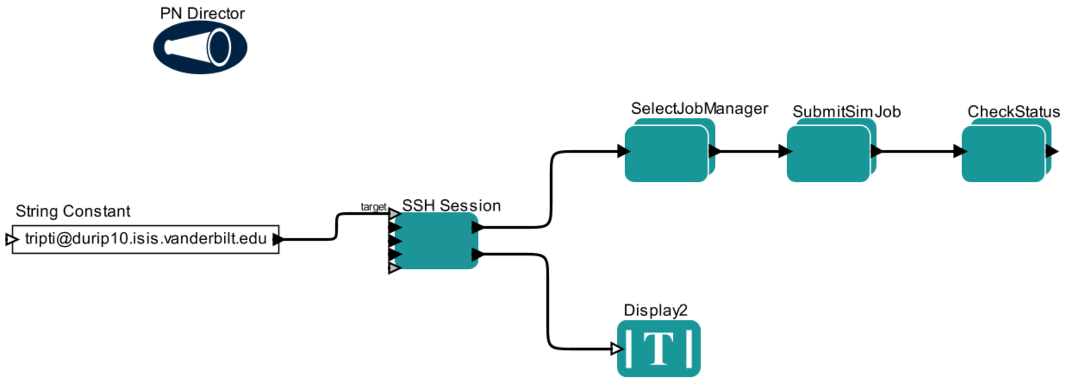

Figure 3: Kepler Job

- **Execution on a Cluster**: The exact steps involved in mapping and scheduling of a job on a cluster are specified using a workflow. There are three logical steps involved in executing a job on a cluster: (1) Select a job manager; (2) Submit a job to the Job Manager; (3) Check the status of the job. Before these steps can be performed, an SSH session has to be created using the *SSHSession* actor. Figure 3 shows the logical steps modeled in Kepler. We briefly discuss each of these steps:

  1. Create an SSH session using an *SSHSession* actor which creates an SSH session to the remote host. The connection can be deferred till the first actor which uses the SSH session (*SelectJobManager* actor) is invoked. This actor is used to provide the private key for public-key authentication and outputs a reference to the SSH session that can be used by other actors.
  2. The *SelectJobManager* composite actor just consists of a *JobManager* actor that is used to define a job manager on a local or remote machine depending on the value of the `jobmanager` parameter. If the value is null or "local" then a job manager is defined on the local machine, otherwise a reference to the remote machine is given as input to the actor for remote definition. At present only Condor, PBS, LoadLeveler, SGE or Fork job managers are supported by the actor. It outputs a reference to the job manager which can be used by the actors that are used to create and submit the jobs.
  3. The *SubmitSimJob* composite actor is used to create and submit a job to the job manager. Figure 4 shows the details of the composite actor. First, the *JobCreator* actor is used to create a job that can be submitted. It requires the path to the job submission file. In our case this is the location of the condor

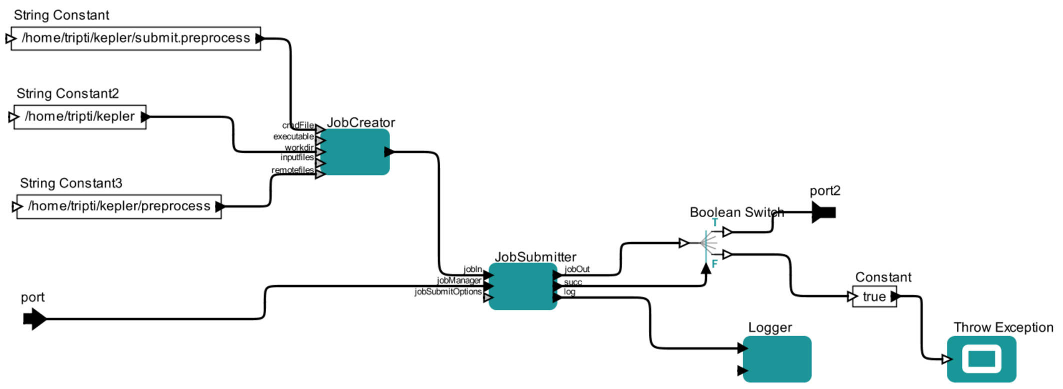

Figure 4: Create and Submit a job

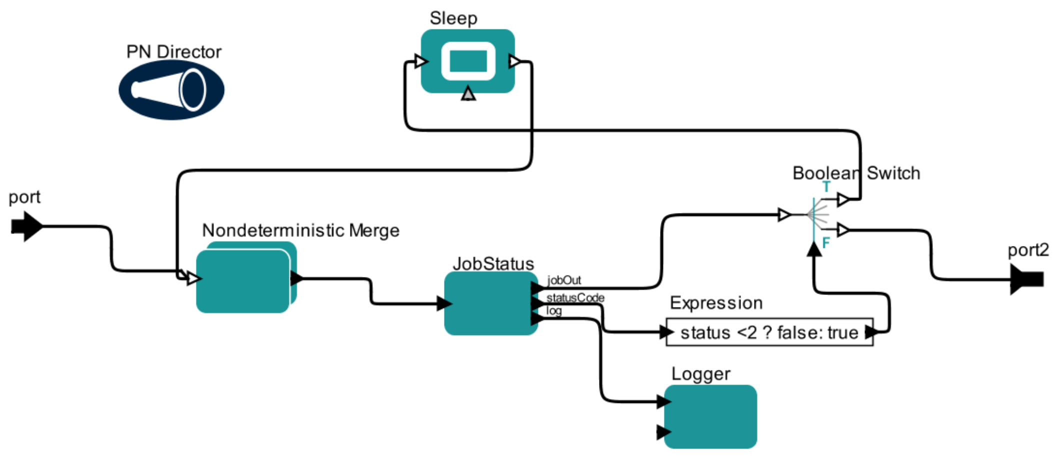

Figure 5: Check Job Status

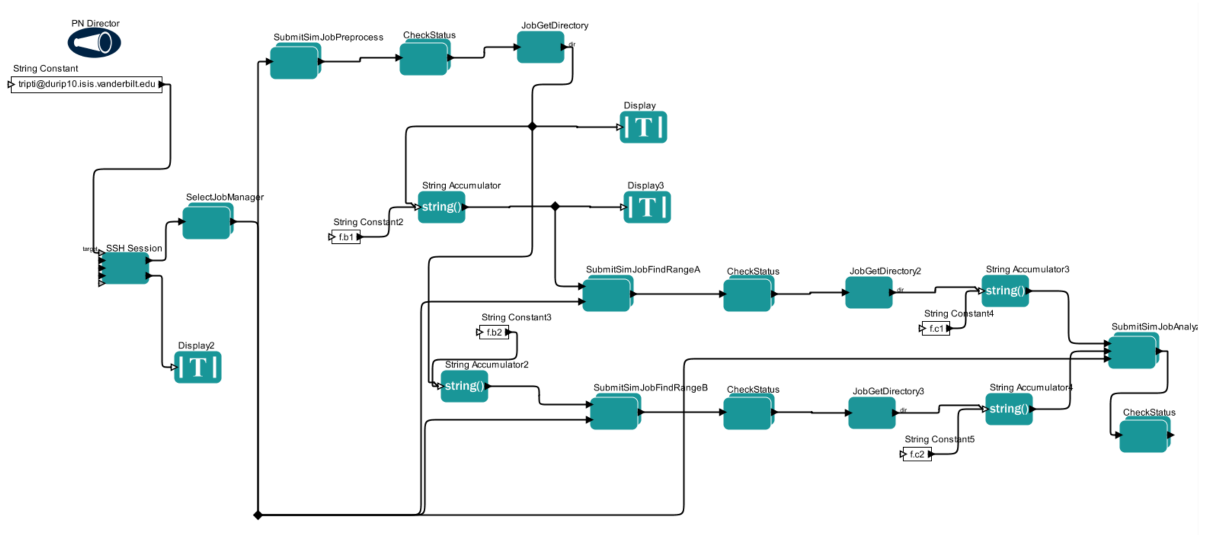

Figure 6: BlackDiamond Workflow in Kepler

job submit file. The job executable can be located in the local machine or remote machine. The actor can takes a list of local files that need to be staged to the remote site before job submission. It also takes a list of remote files that are to be placed into jobs working directory before the submission of the job. It outputs a reference to the created job. This reference is used by the *JobSubmitter* to perform the actual submission using the previously defined job manager using the *JobManager* actor. The actor gives out the reference of the submitted job that can be later used to check the status of the job.

4. The *CheckStatus* composite actor is used to check the status of a job. The *JobStatus* actor is put in a loop where it can check the status of the job. Once it receives the desired status code(for example, status code 0 = Execution error, 1 = Job NotInQueue, 2 = Job Waiting, 3 = Job Running) then the loop is terminated.

Besides the above mentioned actors, Kepler also supports a *GenericJobLauncher* actor that can create, submit and manage a job on a remote machine accessible through SSH. Kepler also supports a *JobGetDirectory* actor that is used to retrieve the name of the working directory created for job execution. Some other job related actors like *JobFileFetcher*, *JobGetDirectory*, *JobGetRealJobID*. The documentation of these actors can be found in the Kepler tool. An example of job mapping and scheduling can be found in the Kepler User Manual [5].

Figure 6 shows the blackdiamond example as modeled in Kepler. There is*no separation between the abstract and concrete workflows in Kepler.* The modeled workflow contains location details of the executables and submission files. Moreover, the sequencing of the workflow is done using status checkes for each job submitted. For example the sequencing between the `preprocess` node and `findrange` node in the blackdiamond example is modeled using a status to check the completion of the preprocess job before the creation and submission of the findrange job. In case of indepedent jobs `findrangeA`, `findrangeB` the Condor scheduler order these jobs.

- **Execution on a Grid**: Kepler mainly supports job submission to Grid resources built by Globus Toolkit. To initiate, monitor, manage, schedule, and/or coordinate remote computations, Globus toolkits, supports the Grid Resource Allocation and Management (GRAM) interface. Usually two different GRAM implementations, namely Pre-WS GRAM and WS GRAM, are provided by the different versions Globus Toolkit. Kepler provides two sets of actors to support these two implementations respectively. More details of the actors can be found in Kepler User Manual [5].

- **Monitoring:** Kepler supports passive monitoring of the workflow systems. Kepler throws an exception in case of error while the workflow is being executed. The error message also consists of additional information of the problem and the affected area. For workflows executing on a cluser, *Kepler provides a* JobStatus *actor which is iteratively invoked. (description not found). For workflows executing on a grid, the string description of the status, which is `UnSubmitted`, `Active`, `Done`, `Failed` or `Expired`, can be gotten from the* Job Status *output of the* GlobusWSJobStatus *actor. History Stack Trace* stores the execution history. Size of the history is limited to the available RAM

4.2.3 Fault-Tolerance

A composite actor called `Checkpoint` provides fault tolerance capability. When a subworkflow within a `Checkpoint` produces an error event, all execution within the Checkpoint is stopped. Checkpoint handles the error itself, or passes it up the workflow hierarchy. Another method for fault mitigation is to retry primary workflow, where number of retries can be configured. Kepler Provenance Framework (KPF) [9] also proposes use of alternative workflow in case the primary workflow fails a certain number of times.

4.2.4 Provenance

The current Kepler framework does not support it, but Kepler Provenance Framework (KPF)[9] is a proposal for addition of data provenance with the current Kepler framework. *Provenance Store*, a database to store provenance data. The KPF has a Provenance Recorder actor that has plugin interfaces for new data models, metadata formats and storage destinations was designed to serve the multi-disciplinary requirements of the broad user community. Querying the provenance store is supported using an API. The API has three components: (1) Kepler, its actors, and external scripts use a Recording API to collect and save provenance information; (2) a Query API provides different query capabilities for dashboards, and query actors in Kepler; and (3) a Management API.

4.2.5 Non-functional Properties

Kepler CORE is a proposed workflow infrastructure, which is a comprehensive, open, reliable, and extensible management system meant to serve a wide variety of scientific communities. The goal of Kepler CORE development is to (i) enable multiple groups in a number of distinct disciplines to easily create, support, and make available domain-specific Kepler extensions; (ii) better support those crucial features that are needed by all disciplines; and (iii) provide for the wide range of deployment scenarios required by different disciplines and distinct research settings. More information about Kepler/Core can be found at [6].

4.3 Pegasus

Pegasus (Planning for Execution in Grids) is a workflow mapping engine that maps high-level workflow descriptions onto distributed resources. It takes a description of computational tasks to be performed and adds the necessary data transfers and data registration tasks (optionally). Pegasus uses the Condor DAGMan execution engine to execute on Condor pools, and clusters managed by PBS and LSF. Pegasus can optimize the workflow performance and reliability.

The Pegasus Workflow Management System (WMS) takes the abstract workflow and maps it onto distributed resources. The Pegasus-WMS consists of three subsystems, namely (1) Pegasus Mapper, which maps the abstract workflow to concrete workflow; (2) Execution Engine which takes the concrete workflow and deploys onto the distributed resources; and (3) Task Manager, which supervises the execution of tasks on local and remote resources.

4.3.1 Composition and representation

- **Workflow Template:** A template in Pegasus is a logical description of a workflow which is location independent. In Pegasus this is the abstract workflow and is written in Directed Acyclic Graph XML (DAX). Users can use different workflow generation utilities (e.g. WINGS), Chimera's Virtual Data Language (VDL), a scripting language, or use APIs to generate the abstract workflow in DAX format. Pegasus supports Java, Python and Perl APIs to generate DAX. Appendix B.1 shows the Java program using API for generating a DAX for the blackdiamond example and Appendix B.2 shows the DAX file generated for a part of the blackdiamond example. The DAX file consists of three main parts:

    1. a list of jobs, where each job is associated with a logical id.
    2. a list of control dependencies between the jobs where each dependency is denoted by a parent-child relationship, thus enforcing a strict ordering on the execution of the tasks in the workflow. No cyclic dependencies are allowed in the pegasus workflow description.
    3. optional catalogs which consist of different kinds of location information. Mainly there are three kinds of catalogs: (1) Replica Catalog, which consists of a list of input and output files that are used by the jobs in the workflow; (2) Transformation Catalog consists of a list executables that are used in the workflow; and (3) Complex transformations that require additional transformations and executables. The physical location of all these files can be included in separate Replication and Transformation Catalogs to keep the DAX file resource and data location independent and portable across different platforms.

- **Workflow Instance**: The abstract workflow is converted to a workflow instance, by using the physical location of the files and data. In Pegasus, the workflow instance is an executable workflow which is in the form of Condor DAG. The `pegasus-plan` is a utility function that is used to create a workflow instance that is mapped on to the target resources. Before we can plan the mapping of the workflow onto the resources we need populate the catalogs and configure workflow execution properties. Besides Replica and Transformation Catalogs, we need a Site Catalog which describes the information of the submit host and various sites (and layout in case of grids) where the workflow can be run. For each site in the grid or cluser the following information is specified:

    1. The job managers for different kinds of schedulers.
    2. Local Replica Catalogs where data residing in that site has to be catalogued
    3. Site Wide Profiles like environment variables
    4. Work and storage directories

    Appendix B.3 shows a sample of the site catalog. *The Site Catlog can be autogenerated using* `pegasus-sc-client`*.*

    We also need to set the properties reflecting the location of the catalogs that will be used for mapping the workflow onto the execution resources. In case separate Replica and Transformation catalogs are used to define the physical location of the files, the properties file should contain the location of all three catalogs.

4.3.2 Workflow Mapping and Execution

- **Mapping** involves several decisions starting with a check on the resource access, site selection, clustering individual jobs. `pegasus-plan` utility performs the task of transforming an abstract workflow into an executable workflow consisting of CONDOR DAG and submit files. The refinement steps in the mapping are as follows:

    1. Data Reuse Module: The Data Reuse Algorithm is used to prune out the nodes from the abstract workflow for which the output files exist in the Replica Catalog. The parents of such nodes are also deleted because the execution of the parent node is not required. The abstract workflow after parsing is optionally handed over to the Data Reuse Module.

2. Site selection Module: The pruned abstract workflow is mapped to the sites passed by the user as an arguement with `pegasus-plan`. If no sites are mentioned then pegasus considers all sites in the Site Catalog and computes a mapping depending on whether it can find an executable on a given site or if the executable can be staged to the particular site. There are number of strategies that are used to perform the mapping if more than one site fulfils the criteria. s
3. Job Clustering Module: clusters the jobs mapped onto the same site.
4. Additional nodes: After clustering, the workflow is handed to the Data Transfer Module which adds data stage-in, inter- site and data stage-out nodes to the refined workflow. The data stage-in nodes move the input files from the location specified by the Replica Catalog to the site where the job will execute. The data stage-out nodes are used to register the output with the Replica Manager. Besides this additional nodes are added to create one shared directory per site. This directory is visible to all worker nodes and will contain the staged in files. Finally, the Clean Up module adds clean up nodes that remove extraneous data when no longer needed.
5. Code Generation module: This module generates the executable workflow which is in the form of Condor DAG and associated job submit files.

- **Execution** in Pegasus is performed using DAGMan [8] execution engine and Condor-G [3] that is used to remotely submit jobs to a variety of jobs using Globus Toolkit services. The concrete workflow produced as a result of the mapping step is in a form of submit files that are given to DAGMan for execution. DAGMan follows the dependencies in the workflow and submits the jobs to Condor-G for execution. Pegasus has a utility called `pegasus-run` that is used for local job submissions.

- **Monitoring** Pegasus comes with a series of utilities that can be used to monitor and debug workflows both in real-time as well as after execution is already completed.
  `pegasus-status`: It is a perl wrapper written around `condor-q`. It allows to check the status of a set of jobs of a particular workflow.

  `Pegasus-run` starts the monitoring utility `pegasus-monitord` in the directory containing the condor submit files.

  `pegasus-monitord` uses the `tailstatd` monitoring daemon to parse the output of DAGMan's `dagman.out` files. It also updates the status of the workflow to an SQLite database and updates the job status to a text file `jobstate.log`, which contains the various states that a job goes through during the workflow execution. `pegasus-monitored` can also be used to mine information from jobs' submit and output files, and either populate a database, or write a file with NetLogger events containing this information. When a workflow fails, and an alternative DAG is resubmitted, `pegasus-monitord` will automatically pick up from where it left previously and continue to write the jobstate.log file and populate the database. Besides this, `pegasus-monitord` will automatically detect when sub-workflows are initiated, and will automatically track those sub-workflows using separate jobstate.log files in each workflow directory. The database at the top-level workflow will contain all the information from the main and sub-workflows.

  `pegasus-analyzer`: pegasus-analyzer is a command-line utility for parsing several files in the workflow directory and summarizing useful information to the user. It should be used after the workflow has already finished execution. pegasus-analyzer quickly goes through the jobstate.log file, and isolates jobs that did not complete successfully. It then parses their submit, and kickstart output files, printing to the user detailed information for helping the user debug what happened to his/her workflow.

4.3.3 Fault tolerance

Pegasus removes the DAGMan and all the jobs related to the DAGMan from the condor queue upon failure. A rescue DAG is generated in case one wants to resubmit the same workflow and continue execution from where it last stopped. A rescue DAG only skips jobs that have completely finished. It does not continue a partially running job unless the executable supports checkpointing.

In case of job failure, DAGMan can also be configured to retry a job a given number of times or if that fails, DAGMan generates a rescue DAG that can be potentially modified and resubmitted at a later time. The rescue

DAG is useful in cases where the failure was due to lack of disk space that can be reclaimed or in cases where totally new resources need to be assigned for execution.

Pegasus supports the following fault-mitigation mechanisms:

- Redundancy
- Light-weight checkpointing at the workflow level
- Resubmission of tasks to the same or another resource
- Restart mechanism at the task level.

4.3.4 Provenance

Provenance capabilities of Pegasus produce (1) application-level provenance through the semantic representations used in Wings, and (2) execution provenance through the Pegasus workflow mapping process. Currently, the details of the job executions are recorded in the Virtual Data System (VDS) provenance tracking catalog (PTC). Each job in the planned workflow is followed by a postscript that stores information in the PTC. Information such as the job executable name and arguments, start time for execution and duration of each job, machine information on which the job ran, etc. are stored in PTC. For a failed job, the error message and the exit status are stored in PTC.

4.3.5 Pegasus

In contrast to Kepler, Pegasus is a workflow management system that is based on the programming language principles. It has a clear separation between the workflow specification and execution by use of abstract workflow. It also provides a compiler to map the high-level abstract workflow to a low level executable description. This compiler is capable of producing correct mapping of the executable workflow onto the underlying resources but is also able to optimize the workflow for a performance enhanced execution. Pegasus relies on a runtime execution engine to carry out the instructions in a scalable and reliable manner.

Support for monitoring and fault-tolerance in Pegasus in bettern thatn kepler.

Pegasus does not support camapigns properly, which are useful to executed multiple

4.4 Observations

We briefly summarize the shortcomings of the frameworks that have been studied as a part of this report.

4.4.1 Kepler

- **Composition and Representation**: Kepler Workflow Management is built upon concepts borrowed from Ptolemy II, and is based on an actor oriented design. The Ptolemy based GUI provides a pre-built library of actors and support for a number of execution models (stream-based pipeline execution, process network etc.).

  One of the disdvantages of Kepler is that any addition of problem specific aspects requires development of a number of specialized actors. For example, if we want to add provenance tracking and workflow optimization aspects to the workflow, then specialized actors have to be created to perform these actions. Thus, local improvements are added to Kepler to customize it to fulfill the requirements of the project at hand. Moreover, the default actors suported by Kepler are fine grained and for example, modeling a workflow to be executed on Condor (as in the case study) involves graphically representing all steps that are required to launch and monitor the jobs manually.

- **Mapping and Execution**: The generic library of actors can be used in any domain and the different execution models enable the developer to create more than simple DAG-based workflows. Kepler system does not differenciate between abstract and concrete workflow. This means that if 2 instances of the same job have to be run on different data sets, 2 instances have to be included in the graphical representation created in Kepler. This is not a problem when small workflows are created but due to lack of scalability, becomes a time consuming task when large workflows are created.

During modelin our case study we observed that auxillary jobs like data movement from one node to the other have to be explicitly modeled as a part of the workflow. We used condor to execute our workflow and needed a separate Condor Submit file for each job in the workflow. Moreover, the sequence of execution of the actors(jobs) have to be explictly modeled using status check actors which checked the status of completion of the preceedin actor in order to dispatch and execute the following actor. The causality of the jobs was not derived automatically by Kepler. Kepler being independent of a scheduler does not use condor efficiently . Should be able to produce a DAG out of the logical dependencies and use condor DAGMAN.

Although Kepler provides a number of execution models, in our case study we found that on the PN Director was used. Therefor, the separation of the director from the actors, which is one of the strengths of Kepler was not completely utilized. Also in our case study it was not possible to exchange the Director from the initial model with a different Director.

The strength of Kepler lies in the graphical interface that is used to capture the workflow. It provides tools to visualize workflow execution, monitoring and debugging.

- **Fault Tolerance**: Kepler does not have support for fault tolerance at present. The simplest type of monitoring that can be done is checking the status of a job. This is also done by explicitly modeling modeling the status check as a part of the workflow. One method of introducing fault tolerance aspects in Kepler is to create special actors and use them in conjunction with third party fault tolerance and recovery components. Mouallem et al [10] proposes a fault tolerance and recovery framework for Kepler-based workflows. The framework proposes creation of a Kepler actor that provides a recovery block functionality at the workflow level and an external monitoring module and checkpointing mechanism. An addition of the fault-tolerance aspect to the existing framework will require an intensive programming effort.

- **Provenance**: Currently, Kepler does not provide support for collection of provenance data. The Kepler Provenance Framework is being developed to provide data provenance support but the implementation of the framework are in testing phase.

4.4.2 Pegasus

In contrast to Kepler, Pegasus is a workflow management system that has a clear separation between the workflow specification and execution by use of abstract workflow.

- Composition and Representation: Pegasus maintains a clear separation between abstract and low level executable workflow The abstract workflow provides a generic view of the causal ordering of the jobs in a workflow. A non-intuitive textual representation is used to capture the abstract workflow.

- Mapping and Execution: Pegasus provides a compiler to map the high-level abstract workflow to a low level executable description. This compiler is capable of producing correct mapping of the executable workflow onto the underlying resources. It also optimizes the workflow for a performance enhanced execution.

  Most of the power of Pegasus comes from the utilization of Condor and related services. Therefore Pegasus cannot be recofigured to use any other distributed scheduler. Pegasus relies on a runtime execution engine to carry out the instructions in a scalable and reliable manner.

- Fault tolerance and Provenance: Fault tolerance and provenance support in Pegasus is much better in Pegasus as compared to Kepler. One of the reasons is better support is because Pegasus is based on the programming language principles. However, similar to workflow execution, Pegasus relies heavily on Condor and DAGMan for fault tolerance support. Pegasus uses DAGMan's rescue DAG for recovery from the failures. However, it is still rudimentary. There is no integration with any existing monitoring framework being used in the facility. Most scheduling decisions in Pegasus are taken by DAGMan and Condor.

References

[1] Bode, B., Halstead, D., Kendall, R., and Lei, Z. The portable batch scheduler and the maui scheduler on linux clusters. *Proceedings of the 4th Annual Linux Showcase and Conference* (October 2000).

[2] Deelman, E., Gannon, D., Shields, M., and Taylor, I. Workflows and e-Science: An overview of workflow system features and capabilities. *Future Gener. Comput. Syst. 25*, 5 (May 2009), 528–540.

[3] Frey, J., Tannenbaum, T., Livny, M., Foster, I., and Tuecke, S. Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing 5*, 3 (2002), 237–246.

[4] Ludaescher, B. The kepler 2.0 actor reference. `https://code.kepler-project.org/code/kepler-docs/trunk/outreach/documentation/shipping/2.0/ActorReference.pdf`.

[5] Ludaescher, B. The kepler 2.0 user manual. `https://code.kepler-project.org/code/kepler-docs/trunk/outreach/documentation/shipping/2.1/UserManual.pdf`.

[6] Ludaescher, B. The kepler core. `https://kepler-project.org/users/projects-using-kepler-1/kepler-core`.

[7] Ludaescher, B. The pegasus workflow management system. `http://pegasus.isi.edu/wms/docs/3.0/quickstart.php#id2816973`.

[8] Malewicz, G., Foster, I., Rosenberg, A. L., and Wilde, M. A tool for prioritizing dagman jobs and its evaluation. *Journal of Grid Computing 5*, 2 (2007), 197–212.

[9] Mouallem, P., Barreto, R., Klasky, S., Podhorszki, N., and Vouk, M. Tracking files in the kepler provenance framework. In *Proceedings of the 21st International Conference on Scientific and Statistical Database Management* (Berlin, Heidelberg, 2009), SSDBM 2009, Springer-Verlag, pp. 273–282.

[10] Mouallem, P., Crawl, D., Altintas, I., Vouk, M., and Yildiz, U. A fault-tolerance architecture for kepler-based distributed scientific workflows. In *Proceedings of the 22nd international conference on Scientific and statistical database management* (Berlin, Heidelberg, 2010), SSDBM'10, Springer-Verlag, pp. 452–460.

[11] Resources, C. *Maui Scheduler Administrator's Guide.*

[12] Resources, C. *TORQUE Administrator Manual.*

[13] Taylor, I. J., Deelman, E., Gannon, D. B., and Shields, M. *Workflows for e-Science: Scientific Workflows for Grids.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

A Glossary

- **actor** - a software component that performs a task, typically by reading input and producing output.
- **job** - a submission to a batch queue. A workflow may contain participants that perform job submissions and manage interactions with the batch system. The submission itself may consist of a workflow (or subworkflow) together with the engine required to execute it, provided it is compatible with the batch system.
- **Writer** - a software entity that makes instances of a user-defined data structure available over a network.
- **Reader** - a software entity that reconstitutes an object from the data that has been received from a writer.
- **Participant** - an actor whose action is triggered by a workflow engine. From the point of view of the workflow engine, the task carried out by a participant is atomic in that the task completes successfully or fails to complete. Moreover, the workflow engine does not manipulate the internal state of a participant. A participant might, for example, be a shell script that runs a data processing application to perform a specific task.
- **Participant Product** - The output generated by a participant.
- **Pipeline** - a workflow in which the participants are arranged according to a pipe and filter architecture. In such an architecture, the participants process units of work and can execute concurrently, each reading input from its predecessor and providing output to its successor. See, for example, the wikipedia entry on software pipelines, Avgeriou Zdun, and Clements et al.. (Often, when astronomers speak of pipelines, what is meant is either a psuedopipeline or some other workflow, or sometimes even just an isolated participant.) publisher - a software entity that prepares data for transmission based on one or more writers.
- **Workflow** - a collection of participants and a defined set of rules specifying when (under what conditions) and how (with what parameters, configurations, and input data) the actions performed by the participants should be triggered.
- **Workflow Template-** - A workflow in which a subset of the data and/or participants are specified abstractly. At a minimum, a workflow template contains only the workflow rules, which refer to data and actors using undefined symbols.
- **Workflow Instance-** is the result of the application of relevant input files and parameters to a workflow template by a user. Essentially, Workflow Instance = workflow template + participants + input parameters and data + output data . It could include multiple versions of the participants. A workflow in which all data sources and participants, as well as their configurations, are specified.
- **Workflow Execution Model-** An executable workflow is produced by applying cluster resource usage policies and constraints to an instantiated workflow by the workflow management system. In other words, the workflow graph is transformed from ideal to something that can be run on an existing cluster.
- **Workflow Management System-** The system that manages and executes workflows on the computing resources. It is responsible for resolving dependencies among the participants, scheduling the participants, keeping track of participant products and fault tolerance.
- **Fault Tolerance-** A fault tolerant system provides its services reliably even when faults occur.
- **Quality of Service-** The level of performance guaranteed by the workflow system. For example, the ability of the workflow system to finish a job within a given time limits.
- **Provenance -**
- **Campaign** - a workflow initiated by a human.
- **Quality of service (QoS)** - the ability to provide different priorities to data flow in a network. Quality of service refers to control mechanisms that are used to reserve resources for different applications, users or data flows, or to guarantee a certain level of performance.

- **roving laptop environment** - an environment in which the user can disconnect from the network and later reconnect (possibly with a different IP address), and is able to continue the work being done before the interruption in the connection.
- **stream of data** - a sequence of units of work of the same type.
- **submit node** - the node that manages a grid job, and that can do monitoring of other nodes involved with running the job. The nodes that run the job can not easily contact each other because they dont know what is the id of the node in which another part of the job is running.
- **subscriber** - a software entity that receives data using one or more readers. unit of work - the smallest data element that is processed in its entirety by an actor. Usually actors operate on a sequence of data elements.
- **Workflow Engine** - a software application that manages and executes workflows. workflow management system - a software application that triggers the actions performed by participants according to the rules that define that workflow. A workflow management system may additionally record provenance and other metadata about the execution of the workflow, and may provide tools for users to specify workflows.

B Pegasus Example

B.1 Java Program to Generate DAX

```
{

import java.io.FileWriter;

public class PegasusCaseStudyDAX {

     /**
     * Create an example DIAMOND DAX
     * @param args
     */
        public static String NAMESPACE = "pegasus";
        public static String VERSION = "2.0";
        public static String PREPROCESS = "preprocess";
        public static String FINDRANGE = "findrange";
        public static String FA =  "f.a";
        public static String FB1 = "f.b1";
        public static String FB2 = "f.b2";
        public static String FC1 = "f.c1";


        public static void main(String[] args) {

                PegasusCaseStudyDAX daxgen = new PegasusCaseStudyDAX();
                if (args.length != 2) {
                        System.out.println("Usage: java PegasusCaseStudyDAX
                         <pegasuslocation> <outputfile>");
                        System.exit(1);
                }
                try {
                        daxgen.constructDAX(args[0], args[1]);
                }
                catch (Exception e) {
                        e.printStackTrace();
                }
        }

        public void  constructDAX(String pegasuslocation, String daxfile){

        try{

                ADAG dax = new ADAG("blackdiamond");

                //files
                File fa  = new File(FA);
                fa.addPhysicalFile("file://"+pegasuslocation+"/f.a","local");
                dax.addFile(fa);
                //fa.addPhysicalFile();
                File fb1 = new File(FB1);
                File fb2 = new File(FB2);
                File fc1 = new File(FC1);

                //location of the executables of the task
                Executable preprocess = new Executable("pegasus", "preprocess", "4.0");
                preprocess.setInstalled(true);
                preprocess.addPhysicalFile("file://" + pegasuslocation
                  + "/preprocess", "local");

                //
                Executable findrange = new Executable("pegasus", "findrange", "4.0");
                findrange.setInstalled(true);
                findrange.addPhysicalFile("file://"+ pegasuslocation + "/findrange", "local" );

                //
                dax.addExecutable(preprocess).addExecutable(findrange);

                // Add a preprocess job
                Job j1 = new Job("j1", "pegasus", "preprocess", "4.0");
                j1.addArgument("tripti");


                // add the files used by the job explicitly
                j1.uses(fa, File.LINK.INPUT);
                j1.uses(fb1, File.LINK.OUTPUT);
                j1.uses(fb2, File.LINK.OUTPUT);
                dax.addJob(j1);

                // Add left Findrange job
                Job j2 = new Job("j2", "pegasus", "findrange", "4.0");
                j2.addArgument("f.c1");
                j2.uses(fb1, File.LINK.INPUT);
                j2.uses(fc1, File.LINK.OUTPUT);
                dax.addJob(j2);

                //add the relationships between the jobs
                dax.addDependency("j1","j2");

                dax.writeToFile(daxfile);

        }catch (Exception e){
                e.printStackTrace();
        }
 }
```

B.2 DAX File

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- generated on: 2011-02-17T12:30:35-06:00 -->
<!-- generated by: tripti [ ?? ] -->
<adag xmlns="http://pegasus.isi.edu/schema/DAX" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://pegasus.isi.edu/schema/DAX
http://pegasus.isi.edu/schema/dax-3.2.xsd"
version="3.2" name="blackdiamond" index="0" count="1">

<!-- Section 1: Files - Acts as a Replica Catalog (can be empty) -->

   <file name="f.a">
      <pfn url="file:///home/tripti/pegasus/pegasus-3.0.1/examples/case-study/f.a"
      site="local"/>
   </file>

<!-- Section 2: Executables - Acts as a Transformaton Catalog (can be empty) -->

   <executable namespace="pegasus" name="preprocess" version="4.0" installed="true">
      <pfn url="file:///home/tripti/pegasus/pegasus-3.0.1/examples/case-study/preprocess"
      site="local"/>
   </executable>
   <executable namespace="pegasus" name="findrange" version="4.0" installed="true">
      <pfn url="file:///home/tripti/pegasus/pegasus-3.0.1/examples/case-study/findrange"
      site="local"/>
   </executable>

<!-- Section 3: Transformations - Aggregates executables and Files (can be empty) -->


<!-- Section 4: Job's, DAX's or Dag's - Defines a JOB or DAX or DAG (Atleast 1 required) -->

   <job id="j1" namespace="pegasus" name="preprocess" version="4.0">
      <argument>tripti</argument>
      <uses name="f.a" link="input" transfer="true" register="true"/>
      <uses name="f.b1" link="output" transfer="true" register="true"/>
      <uses name="f.b2" link="output" transfer="true" register="true"/>
   </job>
   <job id="j2" namespace="pegasus" name="findrange" version="4.0">
      <argument>f.c1</argument>
      <uses name="f.b1" link="input" transfer="true" register="true"/>
      <uses name="f.c1" link="output" transfer="true" register="true"/>
   </job>

<!-- Section 5: Dependencies - Parent Child relationships (can be empty) -->

   <child ref="j2">
      <parent ref="j1"/>
   </child>
</adag>
```

B.3 Site Catalog

```
<?xml version="1.0" encoding="UTF-8"?>
<sitecatalog xmlns="http://pegasus.isi.edu/schema/sitecatalog" xmlns:xsi="http://www.w3.org/2001/XMLSchema
    <site  handle="local" arch="x86" os="LINUX">
        <grid  type="gt2" contact="localhost/jobmanager-fork" scheduler="Fork" jobtype="auxillary"/>
        <grid  type="gt2" contact="localhost/jobmanager-fork" scheduler="unknown" jobtype="compute"/>
        <head-fs>
            <scratch>
                <shared>
                    <file-server protocol="file" url="file://" mount-point="/home/tripti/pegasus/pegasus-3
                    <internal-mount-point mount-point="/home/tripti/pegasus/pegasus-3.0.1/examples/case-st
                </shared>
            </scratch>
            <storage>
                <shared>
                    <file-server protocol="file" url="file://" mount-point="/home/tripti/pegasus/pegasus-3
                    <internal-mount-point mount-point="/home/tripti/pegasus/pegasus-3.0.1/examples/case-st
                </shared>
            </storage>
        </head-fs>
        <replica-catalog  type="LRC" url="rlsn://dummyValue.url.edu" />
        <profile namespace="env" key="PEGASUS_HOME" >/home/tripti/pegasus/pegasus-3.0.1</profile>
    </site>
    <site  handle="condorpool" arch="x86" os="LINUX">
        <grid  type="gt2" contact="localhost/jobmanager-fork" scheduler="Fork" jobtype="auxillary"/>
        <grid  type="gt2" contact="localhost/jobmanager-fork" scheduler="unknown" jobtype="compute"/>
        <head-fs>
            <scratch>
                <shared>
                    <file-server protocol="file" url="file://" mount-point="/home/tripti/pegasus/pegasus-3
                    <internal-mount-point mount-point="/home/tripti/pegasus/pegasus-3.0.1/examples/case-st
                </shared>
            </scratch>
            <storage>
                <shared>
                    <file-server protocol="file" url="file://" mount-point="/home/tripti/pegasus/pegasus-3
                    <internal-mount-point mount-point="/home/tripti/pegasus/pegasus-3.0.1/examples/case-st
                </shared>
            </storage>
        </head-fs>
        <replica-catalog  type="LRC" url="rlsn://dummyValue.url.edu" />
        <profile namespace="pegasus" key="style" >condor</profile>
        <profile namespace="condor" key="universe" >vanilla</profile>
        <profile namespace="env" key="PEGASUS_HOME" >/home/tripti/pegasus/pegasus-3.0.1</profile>
    </site>
</sitecatalog>
```