# Design of a Fault-Tolerant Job-Flow Manager for Grid Environments Using Standard Technologies, Job-Flow Patterns, and a Transparent Proxy

Gargi Dasgupta[1], Onyeka Ezenwoye[2], Liana Fong[3], Selim Kalayci[4],
S. Masoud Sadjadi[4], and Balaji Viswanathan[1]
[1] IBM India Research Lab, New Delhi, India, *{gdasgupt, bviswana}@in.ibm.com*
[2] South Dakota State University, Brookings, SD, USA, *onyeka.ezenwoye@sdstate.edu*
[3] IBM Watson Research Center, Hawthorne, NY, USA, *llfong@us.ibm.com*
[4] Florida International University (FIU), Miami, FL, USA, *{skala001, sadjadi}@cs.fiu.edu*

## Abstract

*The execution of job flow applications is a reality today in academic and industrial domains. Current approaches to execution of job flows often follow proprietary solutions on expressing the job flows and do not leverage recurrent job-flow patterns to address faults in Grid computing environments. In this paper, we provide a design solution to development of job-flow managers that uses standard technologies such as BPEL and JSDL to express job flows and employs a two-layer peer-to-peer architecture with interoperable protocols for cross-domain interactions among job-flow mangers. In addition, we identify a number of recurring job-flow patterns and introduce their corresponding fault-tolerant patterns to address runtime faults and exceptions. Finally, to keep the business logic of job flows separate from their fault-tolerant behavior, we use a* transparent *proxy that intercepts job-flow execution at runtime to handle potential faults using a growing knowledge base that contains the most recently identified job-flow patterns and their corresponding fault-tolerant patterns.*

**Keywords:** Software Design, Job-Flow Patterns, Fault Tolerant, BPEL, JSDL, Grid Computing, Peer-to-Peer.

## 1. Introduction

In Grid and Cluster computing environments, unlike traditional batch environments, individual jobs are typically part of higher-level functional units, generally known as job flows, which are represented by directed graphs. Today, numerous complex academic and commercial high-performance computing applications are being developed as job flows that are composed of several lower-function jobs. Due to the typical long running nature of these jobs, the support for fault tolerance and recovery strategy is especially important.

Very often failure of a job within a flow cannot be treated in isolation and recovery actions may need to be applied to preceding and dependent jobs as well. Thus specifying flow-level recovery mechanisms become important in such scenarios. A prevalent way to handle flow-level compensation is to include failure management logic at modeling time. Wei et al. [1] investigate how to incorporate fault handling and recovery strategy for long running jobs at development time. However, their work requires modification of the original flow to incorporate additional fault-handling logic. The approach also assumes pre-knowledge of all different failure scenarios that can arise. An alternate approach is to handle these job failures at runtime, without explicit changes to job flow process logic. The TRAP/BPEL [2] framework employs this approach for stateless Web service orchestration. In TRAP/BPEL, an intermediate proxy traps calls from the flow engine, and on behalf of it, deploys runtime failure handling. The advantage of this technique is that no direct (or manual) changes need to be made to the flow at development time.

We leverage this approach to enable runtime job failure handling in Grid environments, with dynamic selection of recovery policies. A big challenge in defining recovery policies for Grid jobs is that different jobs may fail at different stages of execution and may require different type of recovery actions. In addition, these long-running jobs often have non-transactional behavior and may require elaborate cleanup phases on account of failure. This is different from the stateless Web service model, where service invocations are of request/response types and recovery plans can mostly be limited to retries of the Web service invocation. Currently, recovery mechanism for long-running jobs requires a high degree of domain expertise. In our work, we explore identification of common, recurrent job flow patterns, and some common fault-tolerance patterns that could be applied to them.

In this paper, we provide a design solution to development of job-flow managers that uses standard technologies to express job flows and employs interoperable protocols for cross-domain interactions among job-flow mangers (Section 2). We enumerate a number of recurring job-flow patterns (Section 3) and introduce their corresponding fault-tolerant patterns (Section 4) to address runtime faults and exceptions. To promote separation of concerns, we use a transparent proxy that intercepts job-flow execution at runtime to handle potential faults using a growing knowledge base that contains the most recently identified job-flow patterns and their corresponding fault-tolerant patterns (Section 5). Finally, we compare our work to a number of related works (Section 6), provide a short summary and a list of future work (Section 7).

## 2. Design Using Standard Technologies

It is of primary importance that the design of job-flow managers follows standard and interoperable technologies, such that both the academic and industrial Grid communities benefit from its flexible and open distributed architecture. As part of the Latin American Grid [3], we have developed a two-level distributed architecture that is comprised of two main middleware components: the *job flow manager*, responsible for maintaining concurrency and sequencing among jobs in the flow, and the *meta-scheduler*, responsible for resource selection and job execution control. In the rest of this paper, we will focus only on the design of the job flow manager. Details about the meta-scheduler can be found in [4].

Figure 1 illustrates two resource domains, namely, FIU and IBM and each are managed by their representative job-flow manager along with a meta-scheduler. The assumption is that within each domain, an application or a Web-based portal sends a job flow to the job-flow manager of its respective domain to be executed. The job-flow manager on its turn submits individual jobs to the meta-scheduler on its respective domain; or it sends partial workflows (sub-flows) to a peer job-flow manager in another domain. .

Peering relationships between job-flow managers and between meta-schedulers is established through a set of protocols that exchange dynamic resource capacity and capability information. This enables them to route sub-flows for remote execution at partner domains. The current protocol includes three phases: connection establishment, job-flow submission, and disconnection.

To express the job flows themselves, we chose the Business Process Execution Language (BPEL or WS-BPEL) [5], which has emerged as the standard workflow language for orchestrating service-based applications. Several production-level software from Oracle, Sun and IBM provide core WS-BPEL engines. These engines are virtual machines that interpret and execute WS-BPEL grammar. The grammar models the business logic of the workflow as a directed-graph, where the nodes represent tasks and the edges represent inter-task dependencies, data flow or flow control.

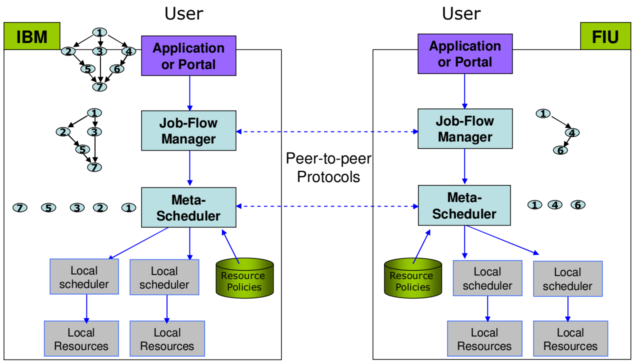

**Figure 1:** A distributed architecture for flow manager and meta-scheduler spanning multiple domains.

Currently, the BPEL specification does not contain the necessary semantics or support for defining long-running jobs. Grid jobs require the richness and flexibility for specifying varied resource requirements and system environments. The Open Grid Forum job scheduling working group recommends the use of Job Submission Definition Language (JSDL) [6], for capturing a job's resource and environment requirements as well as data dependencies. Ideally, we would like to use uniform modeling and processing semantics at the flow manager and at the meta-scheduler. However, in absence of such unified modeling support, we explore using WS-BPEL and JSDL to provide the combined modeling semantics for job flow. This way individual flow tasks are represented as JSDL jobs, woven together using a WS-BPEL workflow. This provides us with the necessary environment based on standardized technologies to explore the coordination of the flow managers and meta-scheduler for fault-handling purposes.

## 3. Job Flow Patterns

Figure 2 illustrates a basic set of workflow patterns [7] that are supported by BPEL. In the sequence pattern (Figure 2(i)), an activity in a process is enabled after the completion of another activity in the same process. Parallelism (Figure 2(ii)) allows activities to be executed simultaneously. Loops (Figure 2(iii)) allow for one or more activities to be executed repeatedly. In the choice pattern (Figure 2(iv)), a number of branches are chosen and executed as parallel threads. Based on these basic patterns, more sophisticated constructs can be built [6].

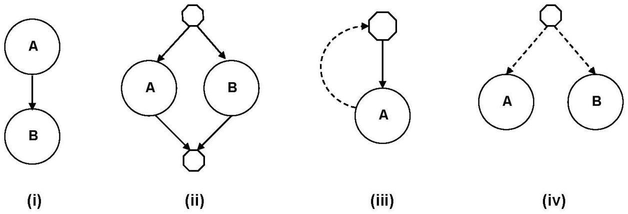

**Figure 2:** Basic workflow patterns supported by BPEL: (i) Sequence, (ii) Parallelism, (iii) Loop, and (iv) Choice.

In the rest of this section, we present some prevalent patterns arising in job flows. These patterns are stored in the flow patterns repository at the proxy and matched at runtime. Based on the underlying functions, they are categorized into the following:

### A. *Job Submission and Monitoring*

A job submission by the flow manager involves invoking the corresponding meta-scheduler interfaces to perform the functions of submission of the job to the resource management layer, and monitoring for any state changes. The different submission patterns observed in job flows include

1. Synchronous job submission: A job flow submits job and waits for completion. In this case the submission call does not return until job completes.
2. Asynchronous submission with polling: A job submission call returns immediately with a job ID. Using the job ID., the job flow polls for the job status.
3. Asynchronous submission with notification: A job flow submits job and gets a job ID. The job flow registers for notification by providing a callback (notification) EPR (End-Point-Reference). The job flow waits for a notification message, before proceeding to the next activity.
4. Asynchronous Fire and Forget: A job flow submits jobs and *does not* wait for job ID or job status (e.g., a batch submission or a cleanup activity). In case of failures, job IDs and job status are sent to the admin or logged instead of invoking job flow. The difference between this and above is:
   - Job flow does not wait for completion of job and thus might complete before such job(s) completes
   - Job failure or success does not affect the job flow business logic

### B. *Data Staging*

Many Grid jobs require input data, and in the absence of a shared file system, these datasets need to be staged in at the site of execution. Usually the data staging needs to be completed before the job can begin execution. In case of job-flows, the data requirement could be an input to the system or produced by the execution of a preceding job. In the latter case, a data-dependency is created in the flow between the producer and the consumer jobs of the data. Thus a typical data staging pattern in job flows comprises of staging in data from either producer jobs or from defined inputs, followed by a job submission pattern. There maybe several such data-staging activities, which could occur sequentially or in parallel. Once the data staging of all dependencies are satisfied, a job can be submitted for execution.

### C. *Job Execution*

Job execution completion status is captured in the job state and in the job state transitions. Some job execution failures are best handled by looking inside the job definition. For example, if a job failed at 'Data Stage In' state and status message gives which file and its reason it failed to be staged-in (e.g., source not available *or* no space on target), a possible failure handling might involve locating a redundant copy of the file or reserving/freeing space on target resource/filesystem before retrying the job. We generalize this as a job flow pattern where the job execution state helps identify failures and the JSDL job description is used for handling such failures.

## 4. Fault Tolerant Patterns

In this section, we introduce a classification for exception handling in the job-flows based on patterns introduced in the previous section. The patterns constitute abstract reusable concepts that can be configured for a range of situations. By identifying these patterns, a domain expert can develop a program generator that captures such reusable patterns and can specify which reusable patterns are to be used [8]. The use of a generator in this case would facilitate separation of concerns, that is, the separate addition of fault tolerant concerns to the job-flow. Selected fault-tolerance patterns are then associated with behavioral policies which define the actions to be taken for a failed monitored task. Below, we briefly describe each of these patterns.

Figure 3 shows a state transition diagram that models the patterns identified in Section 3. Explicit data staging activities may precede a job submission. Failure in any one or more of these staging activities entails a transition to the *Failed* state. A successful job submission assumes that the job is ready to be executed. Thus, this state is followed by either polling for job status or waits on job status notifications. On arrival of a job completion notification or change in job state information from the polled job status information, transition is made to the completed stage. At any of the submission or execution stages, a failure would cause a transition to the failed state. In the next few paragraphs, we describe how fault-tolerance patterns can be applied to offer recovery from failures at any of these stages.

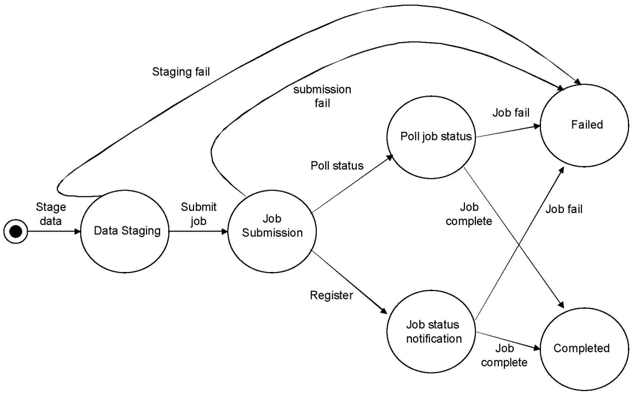

**Figure 3:** Normal job-flow patterns.

**Re-stage data**: Data is re-staged upon the occurrence of an exception either at the *data staging* state or during job execution. A job execution failure may explicitly require the data to be re-staged at the target. Data restaging can be done (a) between the same source and target endpoints of the original staging operation using the same parameters; or (b) by changing the parameters (e.g., data transport protocol, buffer size, timers, etc.) of the transfer; or (c) by specifying a different source in case of multiple copies of the data is present; or (d) by specifying a different target resource when the dependent job is being executed at a new site. Figure 4 illustrates the Re-stage data pattern.

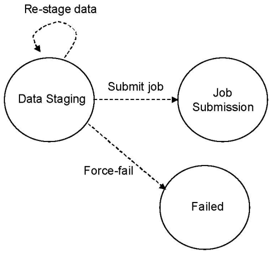

**Figure 4:** Re-stage data pattern**.**

**Re-submit job**: A job is re-submitted for execution upon the occurrence of an exception during job submission or execution. Jobs may be submitted to the same or a different domain and may require modifications in job specifications and resource requirements. Submission failures that arise from unavailability of the meta-scheduler can be recovered by submitting to a new domain meta-scheduler. Execution errors require more detailed analysis of job state, status and exit codes and a fair amount of domain expertise for their fault-handling. For example, a domain expert can realize from experience that a job fails due to lack of disk space, and can update the job definition to reflect to request additional disk space. The failed job can be re-submitted with this new requirement. .Figure 5 illustrates this re-submit job pattern. The possible states to transit from here are the *Data Staging*, *Poll Job Status, Job Status Notification* and *Failed* states. .

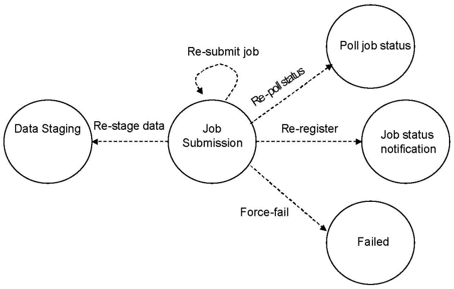

**Figure 5:** Re-submit job pattern.

**Re-poll status**: Polling for job status is resumed upon job re-submission. The proxy in this case, uses its co-relation capability to transparently re-poll for the new job re-submission. This involves translating and modifying the original polling messages from the flow to map to the re-polling of the newly re-submitted job. Figure 6 illustrates the re-poll status pattern. The possible states to transit from here are the *Data Staging, Job Submission, Completed,* and *Failed* states.

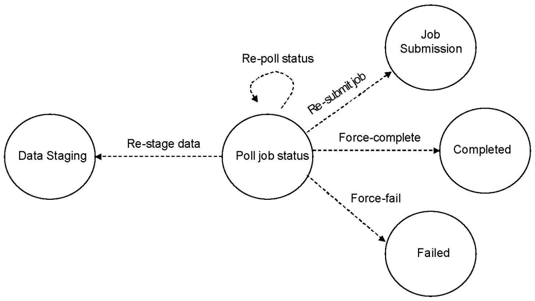

**Figure 6:** Re-poll status pattern.

**Re-register pattern**: Proxy registers for callback job status notification after job re-submission. As in case of the Re-Poll status pattern, this re-registration is transparent to the job-flow. Figure 7 illustrates the re-register for notifications pattern. The possible states to transit from here are the *Data Staging, Job Submission, Completed,* and *Failed* states.

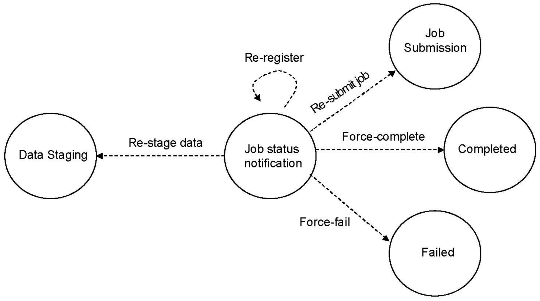

**Figure 7:** Re-register for notifications pattern.

**Force-fail pattern**: Upon job failure, no further progress is possible and its state is changed to failed [9].

**Force-complete pattern**: Upon successful job completion, its state is changed to *Completed.* All subsequent activities may now be triggered [9].

## 5. Fault-Handling Using a Transparent Proxy

As illustrated in the left side of Figure 8, first, the workflow is passed through a Flow Adapter that adapts the BPEL workflow by adding fault-tolerance concerns for specific tasks. The adaptation incorporates some generic interceptors at sensitive join-points in the original BPEL workflow. These join-points are certain points in the execution path of the program at which adaptive code can be introduced at run time. The most appropriate place to insert interception hooks in a BPEL workflow is at the interaction join-points (i.e., the *invoke* instructions). The inserted code is in the form of standard BPEL constructs to ensure the portability of the modified process. This adaptation permits for the BPEL workflow behavior to be modified at runtime [2].

Next, the BPEL based flow manager (FM) executes the adapted workflow. Its main responsibility includes submission of jobs to the meta-scheduler (MS) and monitoring their progress. Additionally, the notification interface can be used for sending back job state change notifications to the flow manager. Based on resource information at the meta-scheduler, it can decide to execute a job or sub-flow locally or dispatch it to the remote domain for execution. When a sub-flow is dispatched, its execution is handled by the flow manager of the target domain. Jobs dispatched from the flow manager to the meta-scheduler can fail due to several reasons. We broadly classify job failures at the meta-scheduler into the following categories:

1. Job submission failure: In this case, job submission from the flow manager to the meta-scheduler fails for one of several reasons. For instance, the network connection is down, the meta-scheduler is not operational, the meta-scheduler is operational but not accepting new submissions, etc.

2. Job execution failure: In this case the meta-scheduler queues the job for submission, but the job fails during execution for reasons that may include resource unavailability, data unavailability, incorrect input specification, internal job exceptions, output data staging, and exceptions during cleanup.

As illustrated in the right side of Figure 8, for runtime failure management at the level of individual jobs, we use a transparent proxy, introduced in TRAP/BPEL [2]. In this case, the proxy sits between the flow manager and the meta-scheduler, and intercepts calls in both directions. For all monitored invocations, the meta-scheduler interface calls are replaced with calls to the proxy interface. However, the proxy is transparent to the flow manager and to the meta-scheduler; therefore, it imposes no changes in either component. The proxy exposes a generic interface to the flow manager which accepts messages containing original invocation parameters, marshaled by the adapter.

A transparent proxy comprises three distinct components: (1) A monitoring component that monitors each adapted invocation; (2) A message correlator component, which correlates individual messages flowing through the proxy to construct conversational state; and (3) A recovery component that kicks in when failure is detected for any adapted component.

An extensible repository of job-flow as well as fault-tolerant patterns is maintained at the proxy. Job flow patterns comprise of common artifacts that are prevalent in job flows represented using the combination of a flow language and a job definition language (e.g., a job submission activity is typically followed by a monitor job state activity). The proxy by virtue of maintaining conversational state for each job is well equipped to detect and handle failures. Fault-tolerant patterns comprise common reusable recovery actions that can be specified for job flow failures. The mapping between job-flow patterns and fault-tolerant patterns can be manually defined at modeling time by the application developer or using pre-defined rule trees. Depending on the rules specified in the tree, a choice can be made on which fault-tolerance pattern to use depending on the job flow pattern. Rules could also be based upon runtime information and domain knowledge.

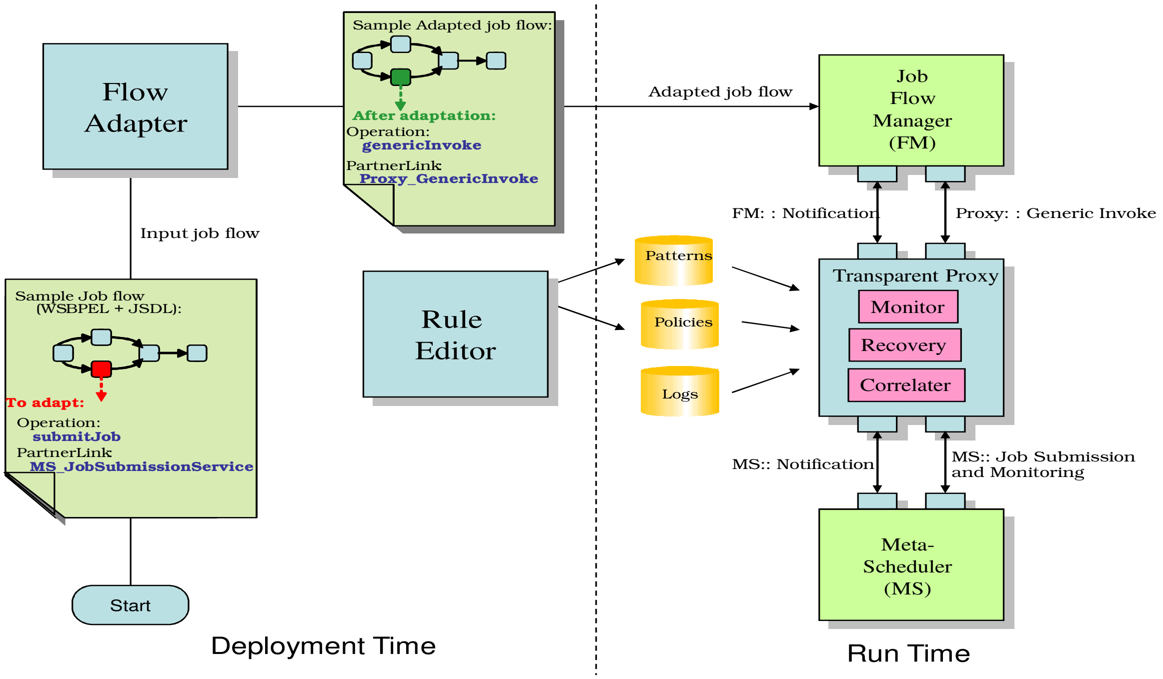

**Figure 8:** The fault-tolerant architecture using a transparent proxy

## 6. Related Work

BPELJ [10] is an extended version of BPEL. Java snippets can be included in BPEL processes for business logic or fault-tolerance concerns. This approach has portability problem, since it needs a specific BPEL engine. AdaptiveBPEL [11] follows an aspect-oriented approach for dynamically adapting a Web service to provide both functional and QoS customization. Adaptation process is policy-driven similar to ours, but this approach also needs a specially built BPEL engine. Pegasus project [12] provides a framework for constructing workflows and mapping these workflows onto Grid resources. Even though Pegasus has advanced capabilities for a better performance of workflow execution, less is provided in fault-tolerance aspect. It provides only remapping of an entire sub-flow in case of a failure whatever the reason may be. The prototype BPEL4JOB [1] also investigate how to incorporate fault handling and recovery strategy in WS-BPEL for long running jobs at modeling time. An alternate approach is to handle these failures at runtime. Authors in [13] study the impact of runtime optimizations made at the scheduler for handling workload surges, while minimizing the reconfiguration overhead.

## 7. Conclusion and Future Work

In this paper, we presented a design for a fault-tolerant job-flow manager that can handle failures at runtime using standard protocols, job-flow patterns and a transparent proxy. We identified common job-flow patterns and some reusable fault-tolerant patterns that can be used for their recovery. We discussed the processes required at development time for a successful runtime fault-tolerant behavior in job flows. In future work, we plan to evaluate our work using a comprehensive set of failure scenarios, explore automatic generation of mapping between job-flow patterns and fault-tolerant patterns, and study the performance impacts of some of these fault-tolerant patterns.

## Acknowledgements

This work was supported in part by IBM, the National Science Foundation (grants OISE-0730065, OCI-0636031, REU-0552555, and HRD-0317692).

## References

[1] W. Tan, L. Fong, and N. Bobroff. Bpel4job: a fault-handling design for job flow management. In Proceedings of Fifth International Conference on Service Oriented Computing (ICSOC), 2007

[2] Onyeka Ezenwoye and S. Masoud Sadjadi. TRAP/BPEL: A framework for dynamic adaptation of composite services. In Proceedings of the International Conference on Web Information Systems and Technologies (WEBIST 2007), Barcelona, Spain, March 2007.

[3] Rosa Badia, Gargi Dasgupta, Onyeka Ezenwoye, Liana Fong, Howard Ho, Sawsan Khuri, Yanbin Liu, Steve Luis, Anthony Praino, Jean-Pierre Prost, Ahmed Radwan, Seyed Masoud Sadjadi, Shivkumar Shivaji, Balaji Viswanathan, Patrick Welsh, and Akmal Younis. *High Performance Computing and Grids in Action*, chapter Innovative Grid Technologies Applied to Bioinformatics and Hurricane Mitigation. IOS Press, Amsterdam, 2007.

[4] Norman Bobroff, Liana Fong, Selim Kalayci, Yanbin Liu, Juan Carlos Martinez, Ivan Rodero, S. Masoud Sadjadi, and David Villegas. Enabling interoperability among meta-schedulers. In *Proceedings of 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid-2008)*, Lyon, France, 2008.

[5] Ezenwoye, O., Sadjadi, S.M.: Composing aggregate Web services in BPEL. In Proceedings of The 44th ACM Southeast Conference, Melbourne, Florida (2006).

[6] A. Anjomshoaa, M. Drescher, D. Fellows, A. Ly, S. McGough, D. Pulsipher, and A. Savva. *Job Submission Description Language (JSDL) Specification, Version 1.0*. Global Grid Forum, 2005.

[7] Dieter Cybok. A Grid workflow infrastructure: Research articles. Concurrency and Computation: Practice and Experience, 18(10):1243–1254, 2006.

[8] Ian Sommerville. Software Engineering, 8th Edition; Chapter 18: Software Reuse. Addison Wesley, May 2006.

[9] N. Russell, W.M.P. van der Aalst, and A.H.M. ter Hofstede. Exception Handling Patterns in Process-Aware Information Systems. BPM Center Report BPM-06-04 , BPMcenter.org, 2006.

[10] Michael Blow et al, BPELJ: BPEL for Java, A Joint White Paper by BEA and IBM, March 2004.

[11] Erradi, A.; Maheshwari, P.; Padmanabhuni, S. Towards a policy-driven framework for adaptive Web services composition, Next Generation Web Services Practices, 2005.

[12] Ewa Deelman et al. Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems, Scientific Programming Journal, Vol 13(3), 2005, Pages 219-237.

[13] G. Dasgupta, K. Dasgupta and B. Viswanathan. Data-WISE: Efficient management of data-intensive workloads in scheduled Grid environments. To appaer in Proceedings of *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2008.