

# New Execution Paradigm for Data-Intensive Scientific Workflows

Mahmoud El-Gayyar, Yan Leng, Serge Shumilov, Armin Cremers  
Department of Computer Science III,  
University of Bonn, Germany  
{elgayyar,leng,shumilov,abc}@cs.uni-bonn.de

## Abstract

*With the advent of Grid and service-oriented technologies, scientific workflows have been introduced in response to the increasing demand of researchers for assembling diverse, highly-specialized applications, allowing them to exchange large heterogeneous datasets in order to accomplish a complex scientific task. Much research has already been done to provide efficient scientific workflow management systems (WfMS). However, most of such WfMS are coordinating and executing workflows in a centralized fashion. This creates a single point of failure, forms a scalability bottleneck, and often leads to excessive traffic routed back to the coordinator. Additionally, none of the available WfMS provides means for dynamic data transformation between services in order to overcome the data heterogeneity problem. This work presents a new approach for scientific workflow management targeted to provide ways for an efficient distributed execution of data-intensive workflows. The proposed approach reduces the communication traffic between services and overcomes the data heterogeneity problem. Moreover, it allows full control over long-running applications, as well as provides support for smart re-run, distributed fault handling and distributed load balancing.*

## 1 Introduction

Many existing systems have already addressed several fundamental issues regarding scientific workflow specification and management. Nevertheless, the design and implementation of an efficient WfMS is still a challenge. The main reasons behind that are the characteristics of scientific workflows themselves. They have generally long lasting execution tasks (e.g. simulation systems) with large data flows and utilizing heterogeneous and dynamic resources.

The GLOWA Volta (GV)<sup>1</sup> project's use cases are con-

<sup>1</sup>[www.glowa-volta.de](http://www.glowa-volta.de) [GLOWA Volta is financed by the German Federal Ministry of Education and Research (BMBF) as part of the GLOWA research initiative: Global Change in Hydrological Cycle.]

sidered in this work as examples of scientific workflows. The project supports sustainable water resource management in the riparian countries in West Africa. In order to help the project's researchers in the decision-making process, scientific workflows are involved so as to orchestrate several involved simulation models and heterogeneous data sources. A practical evaluation of the most potentially suitable WfMSs showed that each of them still lacks some key features necessary to fulfill the project's needs. Therefore, a new approach for scientific workflow management was previously introduced by the authors in [10]. Figure 1 shows the presented approach. It has introduced a four-layered architecture for a WfMS supporting integration of heterogeneous data-intensive applications.

In the **Workflow (Wf) Composition Layer**, researchers can create semantically annotated abstract workflows. For every service, users can specify a set of *resource requirements* (e.g. OS, CPU speed...etc). Annotations in the abstract workflow provide references to a set of ontologies required for data transformations. The **Mapping Layer** contains three main components: First, the *Semantic Matching component* which aims to find corresponding pairs between ontologies. Second, the *Mapping component* which intends to create a set of mapping expressions according to the correspondences obtained from the matching layer. Finally, the *Mediator Generator* component which exploits mapping expressions to create data transformation mediators. Semi-concrete workflows, which combine descriptions of abstract workflows, resource requirements, and mediators information, are created in the **Semi-Concrete Workflow Generation Layer**. The term "semi-concrete" here means that the workflow still lacks information about which concrete computational resources will be used. Due to the dynamic nature of grid environments, this information will be obtained later at run-time. The semi-concrete workflows are then passed to the **Workflow Execution Layer** where they can be executed and monitored over a set of distributed resources.

This paper focuses on the **Workflow Execution Layer** which employs *Web Services* technology to originate a

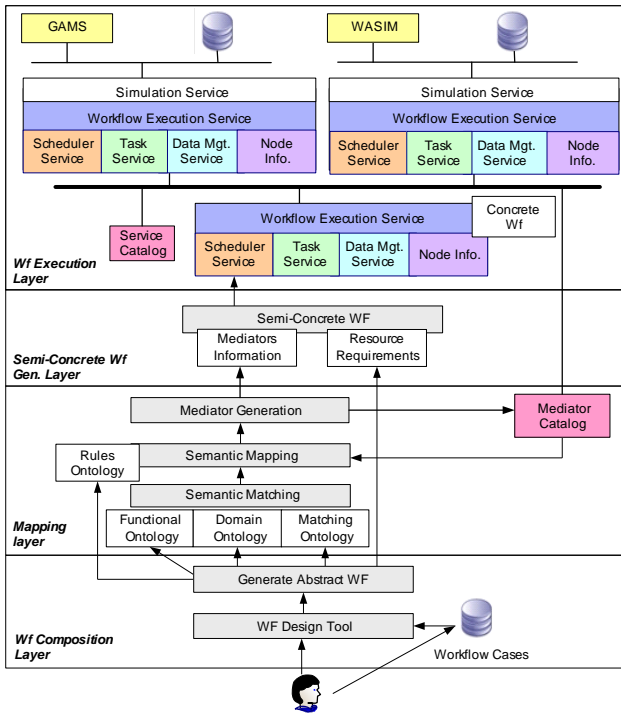


Figure 1. Architecture for WfMS

new distributed execution paradigm for data-intensive scientific workflows. The salient features of such execution mechanism include: i) support for distributed execution of workflows, ii) diminution of communication traffic through reference-based data movement, iii) full control over long-running applications, iv) dynamic data transformation via generated mediators, v) support for smart re-run through data caching, and vi) distributed fault handling and load balancing. The rest of the paper is structured as follows: Section 2 introduces our execution paradigm, whereas in section 3, we discuss a small experiment that we have conducted in order to ensure the functionality of the system and to highlight its main features. Related work is discussed in section 4, and finally, the summary and future work are presented in section 5.

## 2 New Workflow Execution Paradigm

The *Workflow Execution Layer* in Figure 1 provides a new approach for distributed execution of *semi-concrete workflows*. The basic idea of this approach is to separate workflow control and execution flows. It can be realized as a deployment of a bundle of Web Services Distributed Management<sup>2</sup> (WSDM) services for workflow management in every Grid node. This bundle consists of four manageable

<sup>2</sup><http://docs.oasis-open.org/wsdm/>

resources which need be deployed in every Grid node. The functionality of each of these resources is described in the following subsections.

### 2.1 Node Information Resource

The *Node Information WS-Resource* provides the capability to retrieve information about Grid nodes involving both relatively static information (such as system configuration) and more dynamic information (such as instantaneous load). Such service can be used by the scheduler resource in order to find the best Grid node to execute a given task according to the task-specific requirements. In case the current state of all available resources is not satisfying the task's requirements, the scheduler subscribes itself to the *Node Information* service in all available Grid nodes. Whenever a Grid node updated state satisfies a task's particular requirements, its *Node Information* service notifies the subscribed scheduler. The *Node Information* service is also responsible for updating the *Service Catalog*. Once a service is deployed/undeployed on the underlying node, the service sends a notification to the *Service Catalog* in order to update its services' table.

### 2.2 Data Management Resource

The *Data Management WS-Resource* is dedicated to reference-based data movement between nodes and automatic data transformation between heterogeneous services. The *Data Management* service is based on the OGSA-DAI<sup>3</sup>, an open source middleware which connects data resources to the Grid environment. The main reason behind selecting the OGSA-DAI framework is that data transformation mediators, which are built in our workflow management system, are just OGSA-DAI workflows that combine several atomic data transformation activities in order to transform the data formats from one service to another. In our execution paradigm, the output of a workflow's task is stored in an eXist-db<sup>4</sup>, an open source database management system entirely built on XML technology. The OGSA-DAI wraps the XML database and provides external access to it as a web service. Data movements are done through OGSA-DAI references in the format "ogsadai\_service\_url@collection\_name:document\_id" which consists of three parts: the first part is the URL of the OGSA-DAI service; the second part is the collection name in the XML database while the third part is the identifier of the required document.

<sup>3</sup>[www.ogsadai.org.uk](http://www.ogsadai.org.uk)

<sup>4</sup><http://exist.sourceforge.net>

## 2.3 Scheduler Resource

A new *Scheduler WS-Resource* (workflow's main scheduler) will be created every time a user submits a workflow for execution. The scheduler coordinates and monitors the overall execution of a workflow instance. To achieve this, it provides the following capabilities:

**Workflow partitioning:** The scheduler tries to break a submitted *semi-concrete* workflow into subworkflows. The scientific data model of our system is just an extension of the *XScufl* - the workflow description language used in Taverna<sup>5</sup> [6]. Such extension has been made in order to support loops, allow users to specify minimal resource (Grid node) specifications required for the underlying service execution, and to allow data handling through OGSA-DAI references. The main partitioning criteria here is that every subworkflow has only one remote task/processor (e.g. web services or Grid services). Each subworkflow will be submitted to an execution service located on the same Grid node where the remote service is located. Accordingly, the execution service will have a full control over the service execution. Additionally, the scheduler constructs a *dependency table* which determines the data and control dependencies between subworkflows. An example of a partitioned workflow is shown in Figure 5.

**Just-in-time planning:** Grids are very dynamic environments where the availability of resources and their load state can vary from one moment to another. Therefore, the scheduler utilizes a just-in-time planning to schedule the execution of subworkflows. First, it determines which subworkflows are ready for execution according to its *dependency table*. Then, it contacts the *Service Catalog* to retrieve a list of currently available Grid nodes for each remote task (Web/Grid service). In case that there are more than one resource satisfying the task's requirements, the scheduler gives a higher priority to the Grid resource where the task's input data is located, if such node exists in the retrieved list. Otherwise, it selects the best satisfying resource.

**Monitoring the execution of subworkflows:** For every subworkflow, the scheduler creates a new *Task WS-Resource* on the Grid node selected during the planning phase and submits the subworkflow for execution. The scheduler also subscribes itself to the execution state events produced by the *Task* service. Once the scheduler receives an "*execution completed*" event, it extracts the subworkflow's output OGSA-DAI references from the event, updates the dependent subworkflows' inputs with these references, updates the *dependency table* by removing all dependencies forced by this subworkflow, and starts a new planning phase for subworkflows not yet executed.

**Fault Handling:** In the event that the scheduler does not have any execution event from a monitored *Task* ser-

```
<s:source name="base_value" >
  <s:ogsadiRef>
    drogo.iai.uni-bonn.de@33947b92-116d:base_value
  </ogsadiRef>
  <s:mediatorID>
    med-337a5612-..
  </mediatorID>
</s:source>
```

Figure 2. Sample Input Segment

vice after a fixed time-out, the scheduler first tries to request a progress report from this service. The lack of response means that there is a problem either with the Grid node itself or with its deployed *Task* service. Such problem can be solved by rescheduling this subworkflow on a different Grid node. Moreover, the scheduler notifies the *Service Catalog* about the new node status. Hence, the *Service Catalog* updates its services' table in order to prevent other partners from selecting the broken node in the future.

**Checkpointing:** After the execution of each subworkflow, the scheduler stores a snapshot of the current execution state. These snapshots can be used later on to resume a computation in case of failure. Such a feature is very important, especially in our case where workflows contain several long running tasks (e.g. simulation systems).

## 2.4 Task Resource

A new *Task WS-Resource* will be created by the main scheduler for every subworkflow that needs to be executed. The *Task* service is responsible for the actual execution of the submitted subworkflow. Our workflow specification language is based on the *XScufl*, whereas the core of the *Task* service is based on the *freefluo* engine<sup>6</sup> which is the Taverna's workflow enactor. In order to execute a submitted workflow, the *Task* service follows the following sequence:

**Input preparation:** For every input found in the workflow description (see Figure 2), the *Task* service creates a new thread which performs the following steps in order to retrieve the required input:

1. If the workflow description indicates that a mediator is needed to transform the input data to the format of the underlying service, the thread contacts the *Mediator Catalog* and retrieves the indicated mediator which is represented as an OGSA-DAI workflow.
2. Constructs an OGSA-DAI workflow which should be submitted to the *Data Management* service located on

<sup>5</sup>Starting from version 2.0, Taverna uses a different language

<sup>6</sup><http://freefluo.sourceforge.net>

the source node in order to apply the required transformation and to retrieve the transformed data. The data transformation framework remains under development; it should later provide a *Transformation Analysis Tool* which can be used to analyze the transformations in order to determine where it should be performed.

**Execution of the workflow:** The *Task* service waits until all threads are completed, prepares an input list and starts the workflow execution. Before the execution of the underlying service (e.g. a simulation service) is started, the *Task* service first checks whether a cached output for its given input is available. This feature helps to achieve *smart re-run*, since scientists generally tend to change few parameters of their model and re-execute their workflows. In this case, only those services with modified parameters will be re-executed. The user is able to force the system to re-execute a service even if exists a cached output for its given input.

**Output caching:** The *Task* service asks the local *Data Management* service to store the workflow's output. The *Data Management* service saves the output of the underlying service and maps it to the MD5 fingerprint [8] of the given input. Then, it stores the final workflow outputs and generates their OGSA-DAI references.

**Notifying the scheduler:** Finally, the *Task* service notifies the *main scheduler* by sending an "execution completed" event, containing the output references.

During the workflow execution, the *Task* service gathers *provenance information*. In the future, we will record this information in a data catalog. Moreover, the *Task* service provides *distributed fault handling and load balancing* mechanisms. For instance, if the underlying service is broken or the Grid node is heavily loaded, the *Task* service can create a *local scheduler* instance in order to find a new Grid node whereto the subworkflow could be transferred. Then, it notifies the *main scheduler* about the newly selected Grid node so that it can subscribe itself to the subworkflow's execution events. In case of a broken service, the *Task* service also notifies the *Service Catalog* about the new service status. Hence, the *Service Catalog* updates its services' table and sends a notification to the Grid administrator in order to report a failure. In order to illustrate how services on different nodes collaborate together, figure 3 shows a sequence diagram for an execution scenario of a single subworkflow. In this scenario, the input required by the workflow is located in Node1.

### 3 Experiment

The validity of the concepts presented in this paper have been proven on a small use case from the GLOWA Volta project. Figure 5 shows a workflow for the use case "Best

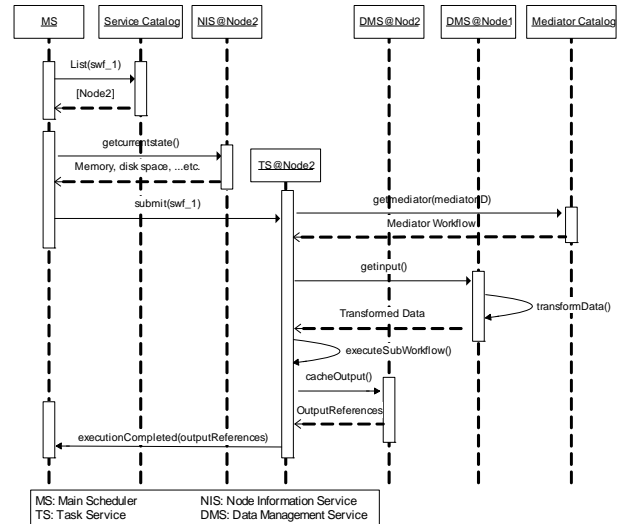


Figure 3. Wf. Execution Sequence Diagram

source of irrigation water" [9]. In this use case, decision makers try to choose the most efficient source of irrigation water in order to attain the optimal profit within a given catchment. In order to achieve this, the use case integrates three simulation systems. The main system is an economic optimization model coded in *GAMS* [1] which seeks to maximize the economic value of available water resources. To be able to deal with complex hydrological systems the model is coupled with a physical hydrology model, which was provided in *WaSiM-ETH* [4]. The input of the hydrological model is based on a climate model coded in *MM5*, as well as land-use data collected, refined, and stored in one of the project's available databases.

The use case has been executed over a small Grid environment with nine Grid nodes (Figure 4). Two Grid nodes represent our global services for the *Service Catalog*, where services' availability information can be retrieved, and the *Mediator Catalog*, where data transformation mediators can be obtained. Six nodes with a set of *GRIA*<sup>7</sup> job services were used as execution nodes. The last node was used to submit our workflow to the main scheduler (*MS*) which is responsible for the workflow management. The workflow was annotated with a set of services' requirements (e.g. OS, CPU speed ...etc) in order to force the *MS* to select a specific node for every service merely for testing purposes.

First of all, the *MS* breaks the workflow into subworkflows and builds its dependency table (Figure 5). According to the created table, subworkflows *swf\_1* and *swf\_2* are ready for execution. The *MS* then contacts the *Service Catalog* and retrieves the list of available nodes for *MM5* and *Land Use Data* services. Consequently, the *MS* submits *swf\_2* to *Node\_6* as it is the only one available for the *Land*

<sup>7</sup>www.gria.org

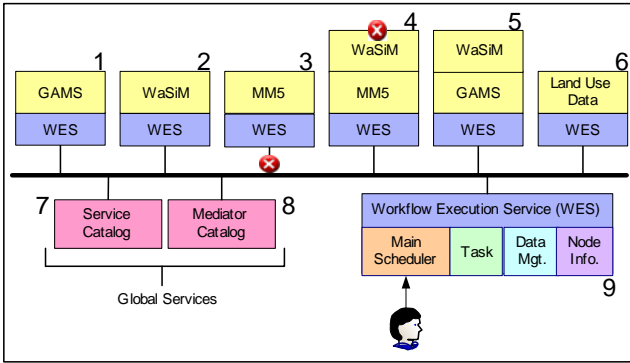


Figure 4. Experimental Grid Infrastructure

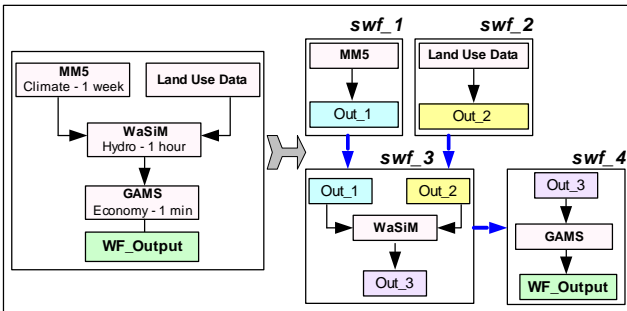


Figure 5. The Wf. for the GV Use Case

Use Data service, and *swf\_1* to *Node\_3* according to the *MM5* service’s requirements annotated in the workflow and the node’s current state retrieved through the *Node Information* service. Since *Node\_3* is already broken, the *MS* should not be able to submit the *swf\_1* subworkflow. In this case, the *MS* uses the cached list of available nodes for the *MM5* service and re-submit the *swf\_1* to *Node\_4*. A complete run of the *MM5* service requires a minimum of one week. During this long period, the *Task* service on *Node\_4* is responsible for monitoring the service’s execution, and sending progress reports to the *MS*. Depending on these reports, the user can steer or terminate the service’s execution.

After receiving “*execution completed*” events from the two submitted subworkflows, the *MS* updates its dependency table and performs the same procedure in order to submit the subworkflow *swf\_3* to *Node\_4*. The *MS* assigns a higher priority to *Node\_4* since the input needed from the *MM5* service is already available there. Then, the *Task* service in *Node\_4* prepares the input for the submitted subworkflow by retrieving the input provided by the *Land Use Data* service, contacting the *Mediator Catalog* in order to retrieve mediators required for data transformations, and applying the obtained mediators over the data. Afterwards, The *Task* service invokes the underlying service and discovers that the service is not working. In this case, the *Task* ser-

vice creates a *local scheduler* in order to find another node to which the subworkflow should be re-submitted. Meanwhile, the *Task* service notifies the *Service Catalog* of the new service’s status. Based on the task’s requirements, the *local scheduler* selects *Node\_5* to submit the subworkflow to it. Finally, the *Task* service on *Node\_4* notifies the *MS* of the new selected node. Consequently, the *MS* subscribes itself to the subworkflow’s events produced by the newly assigned node.

Similarly, the *MS* submits the last subworkflow *swf\_4* to *Node\_5*. After receiving the “*execution completed*” event for the last subworkflow, the *MS* downloads and prepares the workflow’s final output for the user. An *execution progress* report is additionally provided. This report provides information regarding the location and the duration of the execution of subworkflows. In addition, it reports about eventual failures, as well as rescheduling of tasks, that took place during execution. According to the results obtained during the first workflow run, the workflow’s *GAMS* task was updated and re-submitted. The second workflow run completed within few minutes, since the output from the *MM5* and the *WaSiM* services was obtained from the cache without the need to re-execute the simulation services. This small experiment ensures the functionality and the core features of the discussed execution paradigm. Nevertheless, a large-scale test bed involving more complex workflows is underway to conduct a comprehensive testing for our system and to compare its performance to that of other existing workflow management systems.

## 4 Related Work

We have previously evaluated a set of workflow management systems [10], including Taverna [6], Triana[11] and Kepler [5]. The final outcome of this evaluation indicated that these systems are still in need of extra features in order to be able to handle the complexity of scientific applications. For instance, Taverna and Kepler are based on centralized workflow execution engines and they lack abstract dataset types. Computational resources need to be explicitly specified and organized. Distributed execution is only supported by Triana; however it still lacks load balancing and strong fault handling mechanisms. Features such as automatic data transformations or data caching are hardly supported by any of the compared systems. Alongside the systems evaluated in [10], we discuss here two other systems which are similar to our work in their attempts to schedule large scale computations in distributed environments.

Pegasus [2] is designed to map abstract workflows onto the Grid environment. Pegasus proposes a just-in-time planning method which takes into account the dynamic nature of the Grid by splitting the workflow and scheduling the different parts immediately prior to their execution. Concrete

subworkflows are executed over the Grid through the centralized Condor's DAGMan<sup>8</sup> meta-scheduler which submits jobs to Condor-G [3] for execution.

Swift [13] is another system for the management of large scale scientific workflows. It offers the SwiftScript scripting language for specification of complex workflows. Swift programs are converted into *abstract* computations plans, which can be scheduled for execution by Cog Karjan [12], a centralized execution engine which offers libraries for job scheduling, data transfer, and Grid job submission. Swift also integrates Falkon (Fast and Lightweight Task Execution) framework [7] which supports an efficient execution of large numbers of small tasks in Grid environments.

The main difference between our approach and these two systems is that they use one main scheduler and integrate specialized execution environments (e.g Condor-G) for distributed execution. On the other hand, our WfMS deploys several schedulers and executors on available Grid nodes. As previously shown, this paves the way for full control over the remote service execution. In addition, distributed load balancing and fault handling are also possible. Furthermore, our approach integrates the OGSA-DAI middleware for reference-based data movement and dynamic data transformation. Last but not least, smart re-run is provided through data caching and checkpointing mechanisms.

## 5 Conclusion and Future Work

In this paper we have presented a new approach for an efficient execution of data-intensive scientific workflows. Although we have not yet performed a formal comprehensive study, it is evident from our preliminary testing that our new approach provides several concrete benefits. Using principles of decentralized execution of workflows and reference-based data movement, the proposed approach sufficiently reduces the communication traffic between services. Since the execution of workflows is performed by the local execution service that lie in the same host as the target application, full control over the application's service is feasible. The execution paradigm also integrates transformation mediators in order to provide dynamic data transformation. On the service level, a distributed fault handling mechanism is supported by the local scheduler located on the same host where the broken service is located. Distributed load balancing can also be achieved through local schedulers available on different Grid nodes. Last but not least, data caching and checkpointing provide support for smart rerun, where only modified tasks will be actually re-executed.

Our next goal is to improve the system usability, functionality, and reliability. First of all, the *Transformation Analysis Tool* needs to be developed, thus allowing to de-

termine where the analyzed transformations should be performed. This will enrich the functionality of the Data Management service. Another aspect that we have noticed is that reference-based data movement is not enough to considerably reduce the communication traffic between services. This fact is due to the nature of scientific workflows where the output of a single task can be rather huge. In order to solve this problem, we study the possibility of supporting *code movement* rather than data movement. Another important step is to *benchmark* our system components with large scale workflow runs. In addition, in order to increase the reliability of our system, we plan to provide a *distributed workflow management* capability by involving more than one scheduler in the process of workflow management and coordination. From the usability point of view, we need to build an advanced workbench for monitoring and steering of particular subworkflows.

## References

- [1] A. Brooke, D. Kendrick, and A. Meeraus. "Gams: A user's guide", 1992.
- [2] E. Deelman et al. "Pegasus: a framework for mapping complex scientific workflows onto distributed systems". *Scientific Programming Journal*, pages 219–237, 2005.
- [3] J. Frey et al. "Condor-g: A computation management agent for multi-institutional grids". *Cluster Computing*, 5(3):237–246, July 2002.
- [4] K. Jasper and J. Schulla. "Model description wasim-eth". 1999.
- [5] B. Ludäscher et al. "Scientific workflow management and the kepler system". *Concurr. Comput. : Pract. Exper.*, 18(10):1039–1065, 2006.
- [6] T. Oinn et al. "Taverna: lessons in creating a workflow environment for the life sciences". *Concurr. Comput. : Pract. Exper.*, 18(10):1067–1100, 2006.
- [7] I. Raicu et al. "Falkon: a fast and lightweight task execution framework". *IEEE/ACM Supercomputing*, 2007.
- [8] R. Rivest. "The md5 message-digest algorithm", 1992.
- [9] S. Shumilov et al. "First steps towards an integrated decision support system". *20th International Conference on Informatics for Environmental Protection*, 2006.
- [10] S. Shumilov, Y. Leng, M. El-Gayyar, and A. Cremers. "Distributed scientific workflow management for data-intensive applications". *12th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDC2008)*, 2008.
- [11] I. Taylor et al. "The triana project". *Cardiff University*, 2003. <http://www.trianacode.org>.
- [12] G. von Laszewski et al. "Java cog kit workflow". In *Workflows for eScience*, pages 340–356, 2007.
- [13] Y. Zhao et al. "Swift: Fast, reliable, loosely coupled parallel computation". *IEEE Workshop on Scientific Workflow (SWF07)*, 2007.

<sup>8</sup>[www.cs.wisc.edu/condor/dagman/](http://www.cs.wisc.edu/condor/dagman/)