

Distributed Management of Grid-based Scientific Workflows

Dissertation
zur
Erlangung des Doktorgrades (Dr. rer. nat.)
der
Mathematisch-Naturwissenschaftlichen Fakultät
der
Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von
Mahmoud El-Gayyar

aus
Kairo, Ägypten

Bonn, Mai 2012

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn.

Erstgutachter : Prof. Dr. Armin B. Cremers, Bonn (B-IT Research School)

Zweitgutachter: Prof. Dr. Thomas Rose, RWTH Aachen (B-IT Research School)

Tag der Promotion: 02.05.2012

Erscheinungsjahr: *2012*

An Eides statt versichere ich, dass
die vorgelegte Arbeit - abgesehen von den ausdrücklich bezeichneten Hilfsmitteln -
persönlich, selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel
angefertigt wurde, die aus anderen Quellen direkt oder indirekt übernommenen
Daten und Konzepte unter Angabe der Quelle kenntlich gemacht sind,
die vorgelegte Arbeit oder ähnliche Arbeiten nicht bereits anderweitig als
Dissertation eingereicht worden ist bzw. sind, sowie eine Erklärung über frühere
Promotionsversuche und deren Resultate, für die inhaltlich-materielle Erstellung
der vorgelegten Arbeit keine fremde Hilfe, insbesondere keine entgeltliche Hilfe von
Vermittlungs- bzw. Beratungsdiensten (Promotionsberater oder andere Personen)
in Anspruch genommen wurde sowie keinerlei Dritte vom Doktoranden unmittelbar
oder mittelbar geldwerte Leistungen für Tätigkeiten erhalten haben, die im
Zusammenhang mit dem Inhalt der vorgelegten Arbeit stehen

(Mahmoud El-Gayyar)
Bonn, 17th Februar 2012

Acknowledgements

Conceiving a doctoral dissertation is a long and tough journey, and I am indebted to many who have supported me along this journey. First of all, I would like to express my sincere appreciation to *Prof. Dr. Armin B. Cremers* for his continual help, insight and encouragement. I also would like to thank the ZEF Institute and the B-IT research school for sponsoring my PhD.

I would like also to thank the members for my PhD committee for the time and effort they spend on reviewing my work: *Prof. Dr. Thomas Rose, PD. Dr. Volker Steinhage, and Prof. Frank Bertoldi.*

In addition, I am grateful to my colleagues *Dr. Serge Shumilov, and Yan Leng* for their nice and important advices, that helped me to refine this work.

On a more the personal level, I would like to describe my gratefulness towards my stars: my lovely wife *Heba Khamis* and my daughter *Fatima*. My great wife really suffered a lot and has provided extreme support and help all over my journey. For sure, I would like also to acknowledge my great parents and parents in law. Without their love, inspiration and prayers, this research would have never been completed: *Mohamed-Hoda El-Gayyar, Nabil Khamis, & Aida Fawzy.*

Throughout the journey, I have been fortunate to enjoy the compassion of several great and wonderful friends: *Basem Elsaka, Abd Elaziz Ali, Nader Boshta, Khaled Abd Al-Ftah, Mohamed Etman, & many others.* I am grateful to all of them.

Finally, I am very thankful for each person who has ever contributed to my education, especially, *Prof. Dr. Mohamed Mahran, Dr. Magdy El-Gably, Dr. Mohamed Farouk, Dr. Ahmed Magdy & Dr. Hany El-Yamany.* Those people have a great influence not only on my educational level, but also on my life in general.

All Praise and Thanks be to Allah

Distributed Management of Grid-based Scientific Workflows

Abstract: Grids and service-oriented technologies are emerging as dominant approaches for distributed systems. With the evolution of these technologies, scientific workflows have been introduced as a tool for scientists to assemble highly specialized applications, and to exchange large heterogeneous datasets in order to automate and accelerate the accomplishment of complex scientific tasks. Several Scientific Workflow Management Systems (*SWfMS*) have already been designed to support the specification, execution, and monitoring of scientific workflows. Meanwhile, they still face key challenges from two different perspectives: system usability and system efficiency.

From the system usability perspective, current *SWfMS* are not designed to be simple enough for scientists who have quite limited IT knowledge. What's more, there is no easy mechanism by which scientists can share and re-use scientific experiments that have already been designed and proved by others.

From the perspective of system efficiency, existing *SWfMS* are coordinating and executing workflows in a centralized fashion using a single scheduler and / or a workflow enactor. This creates a single point of failure, forms a scalability bottleneck, and enforces centralized fault handling. In addition, they don't consider load balancing while mapping abstract jobs onto several computational nodes. Another important challenge exists due to the common nature of scientific workflow applications, that need to exchange a huge amount of data during the execution process. Some available *SWfMS* use a mediator-based approach for data transfer where data must be transferred first to a centralized data manager, which is completely inefficient. Other *SWfMS* apply a peer-to-peer approach via data references. Even this approach is not sufficient for scientific workflows as a single complex scientific activity can produce an extensive amount of data.

In this thesis, we introduce SWIMS (Scientific Workflow Integration and Management System) framework. It employs the Web Services technology to originate a distributed management system for data-intensive scientific workflows. The purpose of SWIMS is to overcome the previously mentioned challenges through a set of salient features: i) Support for distributed execution and management of workflows, ii) diminution of communication traffic, iii) support for smart re-run, iv) distributed fault handling and load balancing, v) ease of use, and vi) extensive sharing of scientific workflows. We discuss the motivation, design, and implementation of the SWIMS framework. Then, we evaluate it through the Montage application from the astronomy domain.

Keywords: Scientific Workflows, Grid Computing, Web Services, Distributed Computing , Distributed Execution, Load Balancing, Cyberinfrastructure

Contents

1	Introduction	1
1.1	Introduction	1
1.2	Problem Statement	2
1.3	Thesis Contributions	3
1.4	Structure of the Thesis	5
2	Scientific Workflow Management Systems	7
2.1	Grid Computing	7
2.1.1	GRIA - Grid Middleware	9
2.2	Grid Scientific Workflows Applications	10
2.2.1	The Montage Application	11
2.3	Workflow Management Systems	13
2.4	Challenges in Workflow Management Lifecycle	15
3	Related Work and Motivation for SWIMS	19
3.1	Scientific Workflows Management Systems	19
3.2	Scientific Workflows Workbenches	24
3.3	Discussion and Motivation: The SWIMS Framework	30
4	SWIMS: General Design	34
4.1	SWIMS Architecture	34
4.2	Mediator Catalog	37
4.3	Workflow Catalog	37
4.4	Service Catalog	40
5	SWIMS Server Side	46
5.1	Language for Representation of the SWIMS's WMS Components	47
5.2	Data Management WS-Resource	49
5.3	Node Management WS-Resource	51
5.4	Scheduler WS-Resource	52
5.4.1	Workflow Partitioning	53
5.4.2	Planning	56
5.4.3	Workflow Steering	59
5.5	Execution WS-Resource	60
5.5.1	Provenance Information	62
5.6	Optimization	63
5.6.1	Code Movement - Node Management Resource	63
5.6.2	Tailoring the Service Behavior - Node Management Resource	66
5.6.3	Clustering - Scheduler Resource	66
5.6.4	Global Data Caching - Execution Resource	70

5.6.5	Distributed Fault Handling - Execution Resource	70
5.7	Fault Handling in SWIMS	71
6	SWIMS Workbench	75
6.1	SWIMS Workbench Architecture	75
6.2	SWIMS Abstract Workflow Language	77
6.3	SWIMS Workbench User Interface	79
6.3.1	The Editing / Composition Mode	80
6.3.2	Exploring the Workflow Catalog	83
6.3.3	The Monitoring Mode	85
6.4	Workflow Validation	86
6.5	SWIMS: The Overall Picture	89
7	Experiment and Performance Evaluation	93
7.1	Montage Workflow Generator	93
7.2	Experiment Design	95
7.3	SWIMS's Server-Side Evaluation	96
7.4	SWIMS's Workbench Evaluation	102
8	Conclusion and Future Trends	106
8.1	Summary of the Thesis	106
8.2	Accomplishments of the Thesis	108
8.3	Future Work	112
A	GRIA Job Description File	114
B	GRIA Job To Abstract Template XSL Transformation	116
	List of Abbreviations	120
	Bibliography	123

List of Figures

2.1	GRID Architecture	8
2.2	GRIA Job Service Architecture [GRIA 2009]	9
2.3	Montage Workflow	12
2.4	The Reference Architecture for <i>SWfMSs</i> [Lin 2009]	14
2.5	Workflow Management Lifecycle	15
3.1	The Taverna Workbench	24
3.2	Kepler Sample Workflow	27
3.3	Unicore Rich Client	28
3.4	Askalon's Workbench	29
3.5	Architecture of SWIMS Framework	33
4.1	SWIMS Architecture	35
4.2	The Flow of the Workflow Management Process in SWIMS	36
4.3	The Service Catalog Architecture and the Database Model	41
4.4	Double-Layer Security in SWIMS	43
4.5	Methods of Keeping Several Service Catalog Instances Consistent	44
5.1	SWIMS WMS Services.	48
5.2	Data Management WS-Resource Interface.	49
5.3	OGSA-DAI Workflow for Data Transfer and Transformation.	50
5.4	Node Management WS-Resource Interface.	51
5.5	Scheduler WS-Resource Interface.	52
5.6	Workflow Scheduling Lifecycle [UML 2011]	53
5.7	Workflow Partitioning	54
5.8	Execution WS-Resource Interface.	60
5.9	Subworkflow Execution Lifecycle [UML 2011]	61
5.10	Grid (GRIA) Service Replication Workflow.	64
5.11	Level-based Clustered Montage Workflow with Cluster Size =2.	67
5.12	Vertically Clustered Montage Workflow with Cluster Size =3.	68
5.13	Fault Tolerance in SWIMS.	72
6.1	SWIMS's Workbench Architecture	76
6.2	SWIMS's Language XML Schema Tree	78
6.3	SWIMS's Workbench Login Screen.	80
6.4	SWIMS's Workbench Editing Mode.	81
6.5	SWIMS's Workbench Properties Views.	82
6.6	SWIMS's Workbench Workflow Catalog View.	84
6.7	Tailoring the Service Behavior	85
6.8	SWIMS's Workbench Monitoring Mode.	86

6.9	Workflow Validation Sample.	87
6.10	SWIMS according to the Reference Architecture in [Lin 2009]	89
6.11	SWIMS's Simplified Sequence Diagram [UML 2011].	90
7.1	Montage Workflow Generator	94
7.2	Average Completion Time (Montage's Experiment)	96
7.3	Amount of Data Transfer (Montage's Experiment)	97
7.4	Average Task's Execution Speed-up	99
7.5	All Overhead in SWIMS	99
7.6	Distribution of SWIMS Overhead	100
7.7	Data-intensive Workflow	100
7.8	Average Completion Time (Data Intensive Experiment)	101
7.9	Amount of Data Transfer (Data-Intensive Experiment)	102

List of Tables

3.1	Comparison of Workflow Management Systems (SERC)	25
3.2	Comparison of Workflow Management Systems (SUC)	31
6.1	SWIMS's Workflow Validation Symbols.	88
7.1	SWIMS Testbed	95
7.2	Number of tasks per level, and average output size of modules in ten Montage workflows (from 0.1 until 1.0 sq. degree)	96
8.1	Comparison of Workflow Management Systems (SERC) - Recalled .	110
8.2	Comparison of Workflow Management Systems (SUC) - Recalled . .	111

Listings

3.1	SwiftScript Example	30
4.1	Workflow Catalog XML DB Structure	38
4.2	Example of a Provenance Report	39
4.3	Abstract Template for a GRIA Job Service	42
5.1	Sample Input Segment.	62
5.2	Low Level Error Logs.	71
6.1	Sample of an Activity Segment	79

Introduction

Contents

1.1	Introduction	1
1.2	Problem Statement	2
1.3	Thesis Contributions	3
1.4	Structure of the Thesis	5

1.1 Introduction

Nowadays, most of the existing scientific research applications require a lot of CPU time, a lot of memory, and some of them even need to communicate in real time. Another important characteristic of these applications is that they are no longer designed as a bulky single executable module, but consolidate multiple dependent computational modules. In scientific workflows, these modules need to be executed in a predefined order, and may entail a transfer of a massive amount of heterogenous data. Scientific workflows are a variant of business workflows with some different features. We retain Ludäscher’s definition[Ludaescher 2006]: *”These are networks of analytical steps that may involve, e.g., database access and querying steps, data analysis and mining steps, and many other steps, including computationally intensive jobs on high performance cluster computers”*. The main goal of this type of workflows is the automation of scientific experiments, and therefore, it should meet their specific requirements. While business workflows are control-flow oriented, scientific workflows are in contrast, data-flow oriented.

An example of a scientific workflow is the ”best source of irrigation water” use case [Shumilov 2006] presented in the context of the GLOWA Volta project¹ that focuses on the development of decision support tools for sustainable management of natural resources in the Volta Basin in West Africa [ZEF 2011]. Management of natural resources is a complex task raising questions touching many different disciplines (biology, hydrology, etc.), that cannot be solved within a single application. In the ”best source of irrigation water” use case, decision makers try to choose the most efficient source of irrigation water from ground water and surface water to get the optimal profit within a given catchment. In order to

¹GLOWA Volta has been financed by the German Federal Ministry of Education and Research (BMBF) as part of the GLOWA research initiative: Global Change in Hydrological Cycle.

achieve this, the use case integrates three simulation systems. The main system is an economic optimization model coded in GAMS [Brooke 1992] that seeks to maximize the profitable value of available water resources through choice of a crop, size of irrigated cropping, choice of irrigation water source and irrigation schedule. To be able to deal with complex hydrological systems the model is coupled with a physical hydrology model made available in WaSiM-ETH [Schulla 1999]. The input of this model is based on a climate model coded in MM5 [MM5 2003] used to simulate climate for current and for near future (up to 2039), and land use data collected, refined, and stored in one of the project's available databases.

As we can see from the previous example, scientific workflows differ from the normal business workflows. They have long lasting execution tasks with heavy data flows and utilize heterogeneous applications from distinct domains whereas business workflows involve more uniform short, transaction processing tasks with small amounts of data.

Recently, computational Grids [Foster 2003], built using Grid Computing technology, have become a dominating approach for resource sharing and system integration, that are needed for sophisticated scientific workflows. A Grid-based scientific workflow can be defined as the composition of Grid application services, that execute on heterogeneous and distributed resources in a well defined order to accomplish a specific task [Yu 2004].

A scientific workflow management system (*SWfMS*) is an environment that consists of a set of software components to construct, execute, and monitor scientific workflows over a Grid infrastructure. A *SWfMS* provides support in both build-time and run-time. At build-time, it helps users to model the workflow by specifying its tasks, initial input, and data / control flow between tasks. During run-time, the *SWfMS* allows users to steer and monitor the execution process, navigate intermediate results, and get notifications about execution failures.

1.2 Problem Statement

Several *SWfMSs* have already been designed to support the execution and monitoring of scientific workflows [Yu 2005]. However, trying to employ some of these systems in practice, we have found that some of the published claims are hardly justified by the real implementations, and some limitations of the existing approaches have been recognized. Issues such as reliability, scalability, ease of use, and shareability of scientific experiments are still key challenges for existing *SWfMSs*. We argue that these weaknesses mainly result from two main reasons:

1. Scientific workflows are in most cases constructed by scientists themselves. While they are experts in their domains, they are not necessary experts in information technology. The existing *SWfMSs* hardly consider this problem in the design of their user interfaces. These should be conserved in a high level of abstraction and should provide end-user robustness both during the built-time and the run-time stages. Another important feature of an efficient

user interface is allowing its users to share and re-use existing workflows thus supporting the common desire of scientists from different domains who tend to share and re-use their scientific experiments. Most of the existing *SWfMSs*, however, do not provide support for storing workflows in a repository accessible by all users.

2. Most of the current *SWfMSs* utilize a single scheduler and / or an execution engine providing centralized coordination and control of the execution process. This architecture has been preferred due to its benefits such as centralized monitoring and auditing, simple synchronization mechanisms, and ease of design and implementation. Meanwhile, a centralized architecture encounters difficulties to satisfy non-functional aspects of the system, in particular:
 - (a) Scalability: In active scientific environments where many workflow instances need to be managed in parallel, the centralized scheduler and / or execution engine may be overloaded with heavy computations and communication, thereby becoming a potential bottleneck. Thus, system performance can be intensely degraded in such environments.
 - (b) Reliability: A centralized scheduler and / or execution engine are considered as single points of failure in the system. The malfunction of any of them may bring the whole system down.

Based on the above discussion, we claim that the centralized architecture encountered in most existing *SWfMSs* is not ideal for supporting computationally intensive scientific workflows. In addition, the nature of scientific work and procedure should be considered while designing their user interface.

1.3 Thesis Contributions

The main goal of this thesis is to develop a new *SWfMS* for Grid-based scientific workflows, entitled **SWIMS**, that tries to close the gaps in the scientific workflow management process discussed in the previous section. **SWIMS** introduces an innovative scientific workflow execution paradigm. Briefly, the supposed paradigm changes the workflow system design by employing several schedulers and execution engines that cooperate with each other in order to provide a reliable, extendible, and distributed management and execution of scientific workflows. **SWIMS** provides new salient features compared to other *SWfMSs*, including:

1. **Distribution:** **SWIMS** provides support for distributed execution, management, and fault handling of scientific workflows.
2. **Diminution of communication traffic:** **SWIMS** deploys data caching, vertical clustering, and data-aware scheduling techniques to reduce the amount of data transferred during the execution process.

3. **Full control over long running remote services:** Every service in SWIMS is executed through SWIMS's local execution engine that has access to low-level error logs (OS-level). This helps service owners to identify and recover service failures.
4. **Automatic data transformation:** SWIMS utilizes existing data transformation mediators to solve the heterogeneity between the communicating services in a transparent manner.
5. **Ease of use:** SWIMS isolates its users from any technical details through a high level of abstraction.
6. **Smart re-run:** Smart re-run means that only the workflow's updated services are actually re-executed. SWIMS achieves this goal by providing two capabilities, checkpointing and global data caching.
7. **Extensive sharing:** SWIMS enriches the scientists working environment by allowing them to share and re-use their scientific experiments.

It is also quite important to study the performance impacts of these additional features on the overall workflow performance in computational Grid environments. We summarize the accomplishments for our work presented in this dissertation as follows:

1. **Exploration of shortcoming of existing *SWfMSs*:** Practical evaluation of some of the existing *SWfMSs* has been carried out in order to identify existing challenges in the different stages of the workflow management lifecycle.
2. **SWIMS framework:** A new scientific workflow execution paradigm has been introduced through the real environment entitled SWIMS (Scientific Workflow Integration and Management System) that has been designed, implemented, deployed, and evaluated for proof-of-concept purposes.
3. **Easy to use workbench:** A graphical workbench with a high level of abstraction is demonstrated that helps users (scientists) to design their scientific experiments while being completely isolated from technical complexities. It also supports the scientists' work by allowing them to share and re-use other scientists' experiments.
4. **Evaluation of SWIMS:** Real world workflows have been built to evaluate the feasibility, usability, capabilities, and the performance of the SWIMS environment.

Along this work, some conference papers have been published [El-Gayyar 2010, El-Gayyar 2009, Leng 2009, Shumilov 2008]. A website, with contents being updated, has been setup to provide the latest technical documentation and software update under the URL: <http://www-student.informatik.uni-bonn.de/~elgayyar/swims>

1.4 Structure of the Thesis

CHAPTER 2: provides some basic knowledge about Grid-based scientific workflows and investigates the challenges of managing and executing them. First, it introduces GRIA [GRIA 2009] as an example of a service-oriented Grid infrastructure. Then, it discusses the Montage [Montage 2011] scientific application, an astronomical image mosaic service, and the structure of its scientific workflows. Thereafter, this Chapter provides an overview of *SWfMSs* and their reference architecture. Last but not least, it identifies a set of challenges and missing requirements in the different stages of the scientific workflows management lifecycle.

CHAPTER 3: explains the related state of the art in the area of scientific workflow management and execution and motivates our work. First, it compares a selected set of currently available *SWfMSs* against the identified challenges in the scientific workflow management lifecycle. Then, it motivates our work to develop an advanced environment for scientific workflow management and execution (entitled SWIMS) that helps to overcome these challenges.

CHAPTER 4: introduces the overall architecture of our Scientific Workflow Integration and Management (SWIMS) environment and highlights its main components. SWIMS components fall into three categories: client-side components, server-side components, and global components. This Chapter also discusses SWIMS's global components in more detail.

CHAPTER 5: focuses on the SWIMS's server-side components: The Workflow Management System (WMS) instances. In order to simplify the design and the implementation of the WMS, we decided to break it into four major subcomponents: Scheduler, Node Management, Data Management and Execution. Each WMS's subcomponent can be realized as a Web Service Distributed Management (WSDM) based service, which controls a set of manageable resources. The Chapter starts with an introduction to the WSDM specification, and then it discusses the main functionalities of each WMS subcomponent in detail.

CHAPTER 6: finalizes the discussion about the SWIMS environment by considering the SWIMS's client side component: the SWIMS workbench. The main target of the SWIMS workbench is to provide a simple environment with a high level of abstraction through which scientists can compose, execute, monitor, steer, re-use, and re-run scientific workflows without considering the complex underlying Cyberinfrastructure used to perform these tasks.

CHAPTER 7: explores the experimental results obtained from a "test deployment" of the SWIMS environment based on Montage workflows from the astronomy domain. The main goal of this deployment was to evaluate the

feasibility, usability, capabilities, and the performance of the server and client side components of SWIMS.

CHAPTER 8: concludes the thesis and suggests possible future work.

Scientific Workflow Management Systems

Contents

2.1 Grid Computing	7
2.1.1 GRIA - Grid Middleware	9
2.2 Grid Scientific Workflows Applications	10
2.2.1 The Montage Application	11
2.3 Workflow Management Systems	13
2.4 Challenges in Workflow Management Lifecycle	15

In most cases, scientific workflows consist of complex tasks that need a large amount of computing and storage resources. Accordingly, it will be very difficult to execute all the tasks of a workflow on a single machine. At the same time, most of the scientific resources are geographically distributed and belong to different administrative domains. Due to these factors, Grid technology is emerging as a dominant execution environment for scientific workflows. A scientific workflow can be managed and executed over a Grid through a Workflow Management System (WMS). In this chapter, we are going to dig into the details of the scientific workflow management and execution process in order to identify the existing challenges in each of its phases.

The chapter is organized as follows: The first section introduces the Grid computing model. The second section gives an overview and some example about scientific applications. The third section briefly discusses the Scientific Workflow Management Systems (*SWfMS*) and their reference architecture. In the last section, we investigate the scientific workflow management lifecycle, and highlight discovered challenges in each of its phases. We defer the comparison of some of the existing *SWfMS* to the next chapter.

2.1 Grid Computing

In the early 1990s, the term **Grid** was invented as a metaphor for technologies that would allow consumers to obtain computing power on demand. The main goal of Grid computing is to "enable resource sharing and coordinated problem solving in dynamic, multi-institutional virtual organizations (VO) " [Foster 2001,

[Foster 2002]. Grid computing provides a viable supplement of super computers and large dedicated clusters to address and solve large scale computation problems.

In order to achieve these functionalities, Grids provide a set of standard protocols, middleware, toolkits, and services built on top of these protocols. Grids provide their protocols and services at five different layers as shown in Figure 2.1 [Foster 2003, Sotomayor 2006]. The **fabric layer** provides the various types of resources (e.g., computational resources, storage systems, network resources, etc.) shared within the Grid. The **connectivity layer** defines core communication, and authentication protocols required for Grid transactions. The **resource layer** builds on the connectivity layer protocols to define protocols for secure publication, discovery, negotiation, control, usage, and accounting of shared resources. The **collective layer** contains protocols that captures interactions across collections of resources (e.g., directory and scheduling services). Finally, the **application layer** incorporates user applications build upon the protocols and services of the other layers (e.g., Grid workflow systems and Grid portals).

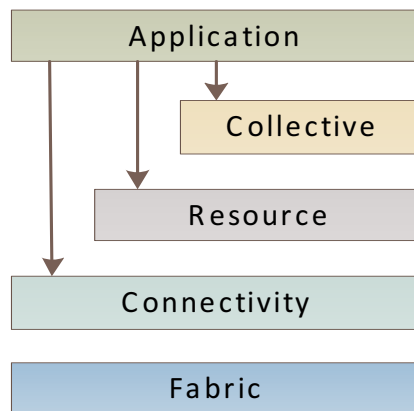


Figure 2.1: GRID Architecture

By 2001, the Open Grid Services Architecture (OGSA)- an architecture for a service-oriented Grid Computing [Foster 2006b], was developed by the Globus Grid Forum (GGF)¹ [OGF 2006]. OGSA is a computing architecture based on services, assuring interoperability between heterogeneous systems so that different types of resources can communicate and share information.

Grid Middleware are software stacks that implement Grid architecture to present disparate compute and data resources in a uniform manner, such that these resources can be shared by remote client software without knowing in advance the systems' configurations. Several Grid middleware have already been developed, e.g., Globus [Sotomayor 2006, Foster 2006a], Unicore (UNiform Interface to COmpute REsources) [Streit 2010], and GRIA [Surridge 2005, GRIA 2009]. In the next subsection, we will focus on GRIA as we have selected it as our Grid infrastructure.

¹Nowadays called Open Grid Forum (OGF) after being merged with the Enterprise Grid Alliance

In 2007, the term **Cloud computing** came into popularity; we believe that Cloud computing has evolved out of Grid computing. The vital difference between both of them is the shift of their main goal from an infrastructure for sharing storage and computational resources (Grids) to an economic environment for delivering more abstract resources and services (Clouds). For comprehensive comparison between Grid and Cloud computing, you can refer to [Foster 2008].

2.1.1 GRIA - Grid Middleware

GRIA [SurrIDGE 2005, GRIA 2009] is an open-source service-oriented infrastructure designed to support B2B collaborations. It provides a service provision across / within organizational boundaries in a secure, interoperable and flexible manner. The aim of the GRIA project has been to increase the usability of Grids for businesses and industrial users.

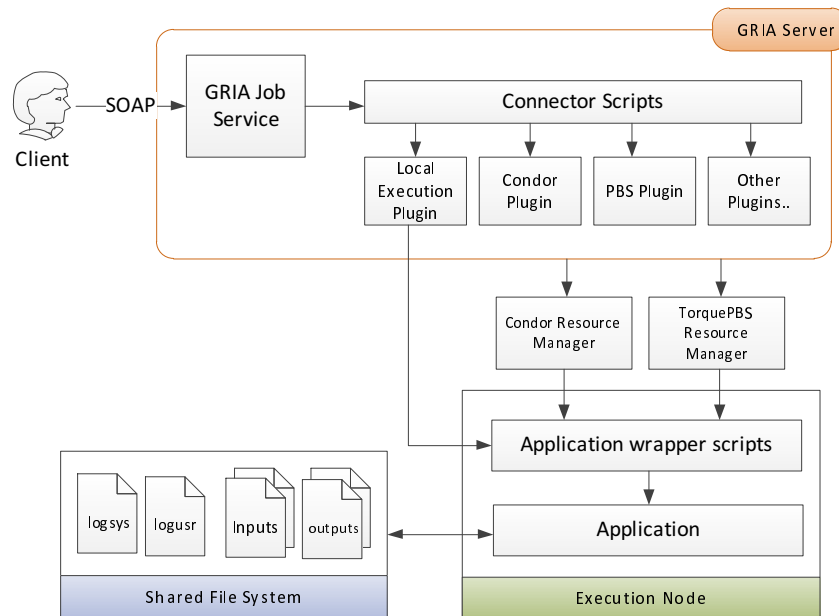


Figure 2.2: GRIA Job Service Architecture [GRIA 2009]

GRIA uses Web service protocols based on key interoperability specifications. The GRIA basic application services package allows service providers to deploy easily legacy applications as managed Grid services. In general, a legacy application is an application developed with outdated technologies. In terms of Grid services, legacy applications refer to platform dependent applications providing a service with its actual application logic. GRIA provides two basic services, as follows:

1. A Data Service: This service allows remote users to upload and download data files to the service provider, and to transfer data between different Data

Services. The Data Service also supports management of access rights granted to other users or service providers.

2. A Job Service: This service allows remote users to start, monitor or kill computational jobs, executed by the service provider. The Job Service fetches input from and writes output to a local Data Service.

As shown in Figure 2.2, service providers integrate applications into the Job Service using wrapper scripts. These provide a consistent abstraction between GRIA and the underlying application. Application wrappers can be simple shell scripts that unzip input files, invoke the legacy application and zip application generated output files for writing to output data services. Application wrappers become more complex when context mapping on input data is required; in these cases other programs or script modules, e.g. Python's modules, can be invoked to provide the necessary functionality. GRIA allows service providers to run applications on various execution platforms, including local execution and computational clusters (e.g. Torque PBS [Staples 2006] and Condor [Thain 2005]), using a wide range of different resource managers depending upon the business need. A part from application wrappers, an application needs a description file. This file contains metadata about the application which is essential for GRIA users to discover and to use the application. An example and more information about application description files are given in Appendix A.

Another advantage of GRIA is supporting connectivity of heterogeneous client applications. It provides a client-side Java API that allows client applications such as workflow tools and portals to access GRIA services. As an example for workflow applications, GRIA provides a workflow plugin for Taverna [Taverna 2011] and the Freefluo workflow enactment engine [Freefluo 2009]. It enables GRIA job and data transfer operations to be composed and executed through Taverna. In addition, GRIA provides the Workflow Application software which includes tools for deploying and running Taverna workflows as GRIA applications.

As we are going to see later, SWIMS has selected GRIA as its underlying Grid middleware due to its flexibility, interoperability, and integrability characteristics. As a result, SWIMS exploits the Freefluo engine as its own workflow enactor seeing that it is already supported by GRIA.

2.2 Grid Scientific Workflows Applications

In recent years, many scientific processes became distributed in nature due to the evolvement of distributed Grid Computing; scientific data are generated and stored across wide area networks, computing resources are distributed and heterogeneous, and scientists tend to share their experiments and research goals while being geographically distributed.

Scientific workflows are emerging as a dominant approach to deal with these types of processes and to handle the complexities of their environments. They

allow scientists to assemble highly specialized applications, and to exchange large amount of heterogeneous datasets to automate the accomplishment of complex scientific tasks [Tsalgatidou 2006]. Scientific workflows have already been exploited to support different scientific applications from diverse scientific domains [Bharathi 2008].

One example of the astronomy domain is the Montage application [Katz 2005] which creates science-grade astronomical image mosaics using data collected from telescopes. As we have selected the Montage workflow as our evaluation use case, we are going to discuss it in more details in the following subsection.

Another candidate from the bio-informatics domain is the Epigenome [USC 2011] which maps short DNA segments collected using high throughput gene sequencing machines to a previously constructed reference genome. The workflow splits several input files into small chunks, reformats, converts, and maps the chunks to a reference genome, merges the mapped sequences into a single output map, and computes the sequence density for each location of interest in the reference genome.

Many other scientific workflows have already been developed: e.g. CyberShake [Graves 2011], The Laser Interferometer Gravitational Wave Observatory (LIGO) [Brown 2007], and SIPHT [Livny 2008].

2.2.1 The Montage Application

Montage [Montage 2011, Taylor 2006] is an astronomical image mosaic service for the National Virtual Observatory. It delivers on demand, science-grade, mosaics that satisfy user specified parameters of projection, coordinates, size, rotation and spatial sampling. Montage uses input images in the Flexible Image Transport System (FITS) format .

The Montage application has been represented as a workflow that can be executed in Grid environments such as the TeraGrid [Berriman 2004, Jacob 2009]. Figure 2.3 shows the structure of a Montage workflow. The vertices represent the processes, and the edges represent the data dependencies between them. The number within the vertices represents the level of the task in the workflow. All tasks that have no parent tasks are at level one. The level of any other task is the maximum level of any of its parents plus one.

The Montage workflow accommodates all different basic structures of scientific workflows: Process, pipeline, data distribution, data re-distribution and data aggregation [Bharathi 2008]. In addition, the number of inputs processed by a Montage workflow may increase over time as more images of a particular region of the sky are available. As such, the structure of the workflow changes to accommodate the increase in the number of inputs, which also translates to an increase in the number of computational jobs. Due to these facts, Montage workflows have been widely used to evaluate scientific workflow algorithms and systems.

In Montage application workflows, the number of mProject jobs is equal to the number of input FITS images to be processed. Each mProject job re-projects its

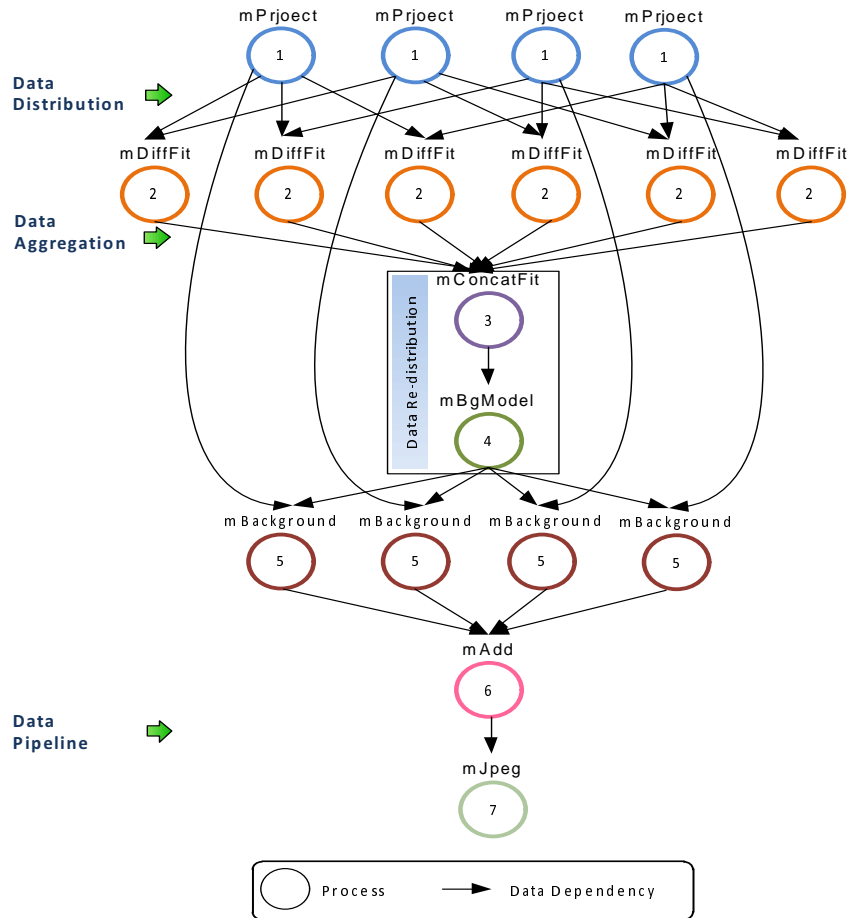


Figure 2.3: Montage Workflow

input image to a common spatial scale, coordinate system and World Coordinate System (WCS) projection. The outputs of each `mProject` job are the re-projected image and an "area" image that consists of the fraction of the image that belongs to the final mosaic. These are then processed together in subsequent steps. An `mDiffFit` job runs an `mDiff` job immediately followed by an `mFitplane` job and check the first to decide whether to run the second. An `mDiff` job analyzes an image metadata table to determine a list of overlapping images. Each image is compared with every other image to determine all overlapping image pairs. A pair of images are deemed to overlap if any pixel around the perimeter of one image falls within the boundary of the other image. In case that two images are overlapped, the `mDiff` job calculate a simple difference between them and runs an `mFitplane` job which uses a least squares algorithm to fit the difference images. The number of `mDiffFit` jobs in the workflow is nC_2 where n is the number of input images. The `mConcatFit` job is a data aggregation job which merges multiple plane fit parameter files (from `mDiffFit`) into one file. This module is only needed in a Grid environment

where the `mDiffFit` jobs may run in parallel on computers that do not share a file system. In this case, the fit parameters have to be merged into one file before the `mBgModel` job can be called. Next, the `mBgModel` is used to determine a correction which should be applied to each image to obtain a good global fit that minimizes the inter-image differences. The specified background correction is applied to each re-projected image by the `mBackground` jobs. The `mConcatFit` and `mBgModel` jobs together can be considered as a data redistribution point. The `mAdd` job is another computationally intensive job, which is responsible for the co-addition of re-projected, background-corrected images into a final mosaic in FITS format. Finally, the generated FITS image is converted to a JPEG format through the `mJPEG` job.

We have used the Montage application workflows to evaluate our system as we are going to show later in detail in Chapter 7.

2.3 Workflow Management Systems

Complex scientific experiments can be held by scientists through putting together data analysis and knowledge discovery "pipelines". These pipelines can be constructed from shared data and computational services through the evolving Grid technology. However, scientists should not bother themselves about the underlying infrastructure and focus only on the development and use of what are called scientific workflows. These are networks of analytical steps that may involve database access, data analysis, computationally intensive jobs (e.g., complex simulations), etc. In other words, a scientific workflow provides a formal specification for automating a scientific process [Tsalgatidou 2006].

The characteristics of scientific workflows are quite similar to those of business workflows. The most important difference between the two models is that business workflows focus more on control flow patterns while data flow has usually a minor concern. On the other hand, scientific workflow execution models are much data-flow oriented.

A scientific workflow management system (*SWfMS*) is an environment which helps scientists to construct, execute, modify, manage and monitor scientific workflows [Yu 2005]. The Workflow Management Coalition (WfMC) proposed a reference architecture [Hollingsworth 2004] which has been adopted successfully in the development of business workflow management systems. However, the proposed architecture is not the best candidate for *SWfMSs*, that tend to be more data-flow oriented, providing new challenges for system development (e.g., transfer of a huge amount of data, data transformation between heterogeneous services, intensive user interaction, etc.). As a deduction, a novel reference model has been introduced, that affords a guidance for the architectural design of a particular *SWfMS* in various scientific domains [Lin 2009].

The reference architecture for *SWfMSs* consists of four main layers as shown in Figure 2.4. In the *Operational Layer* reside both local or remote data sources and

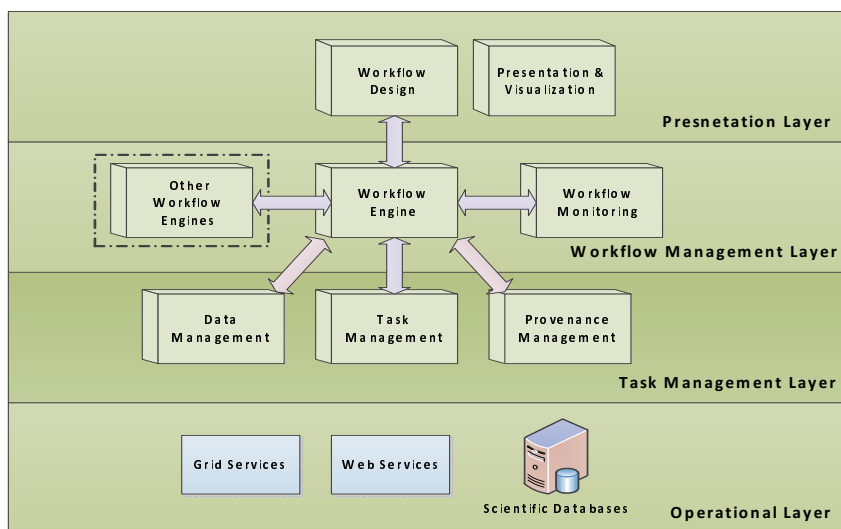


Figure 2.4: The Reference Architecture for *SWfMSs* [Lin 2009]

computational services. The *Task Management Layer* tries to abstract the services and software tools obtained from the *Operational Layer* into workflow tasks. It is also responsible for managing of data, workflow tasks, and provenance information. The separation of the *Task Management* and the *Operational Layers* promotes the extendibility of the *Operational Layer* with new service. The *Workflow Management Layer* is responsible for executing and monitoring of scientific workflows. In addition, this layer considers interoperability issues between the workflow engine and other workflow engines. The final layer is the *Presentation Layer*, which supports the workflow design and visualizations for all assets of the whole system.

The new architecture considers the main differences between scientific and business workflows. First, it accommodates the provenance and data product management components to support scientific data reproducibility and analysis. That was a missing feature in the reference architecture proposed by the WfMC. Second, disconnecting the Presentation Layer from the Workflow Management Layer enables the support of user interaction and user interface customizability. Third, separating the Workflow Management Layer from the Task Management Layer disconnects the workflow management from the task management, therefore, allowing the parallel advancement of them. Finally, the disengagement of the Task Management Layer from the Operational Layer facilitates the separation of management of uniform workflow tasks from the heterogeneous low-level task implementation strategies and execution environments.

2.4 Challenges in Workflow Management Lifecycle

A *SWfMS* follows a four stages lifecycle to construct, execute, and monitor scientific workflows (Figure 2.5). The first stage of this lifecycle is the creation stage where users should be able to create either abstract or concrete workflows and populate them with the initial input data. Workflows can be created from scratch or through editing an existing workflow template.

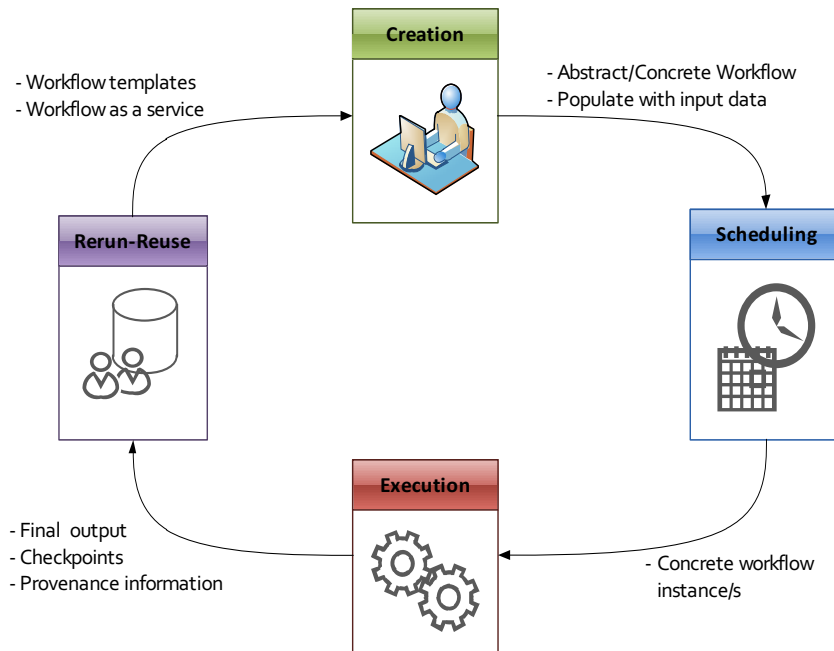


Figure 2.5: Workflow Management Lifecycle

An Abstract workflow is a non-executable workflow as it lacks the execution information. Thus, it requires the second stage, the scheduling stage, which converts it into a concrete executable workflow by mapping its abstract tasks onto computational nodes. The mapping process is usually automated and entails the detection of available resources.

The next step in the workflow management lifecycle is to execute the mapped workflow. As a part of the execution process, data is generated and moved between the workflow's tasks, provenance information is collected, and fault handling mechanisms are used to recover from execution faults. In general, users should be able to monitor and steer the workflow's execution.

Once a workflow has been created and successfully executed, it should be shared in a global workspace to allow other users to re-run / re-use it. In addition, users should be able to deploy their workflows as Grid services, that can be used later as atomic tasks in newly created workflows. The re-run / reuse stage is very important for scientific workflows as normally scientists tend to analyze, re-run or re-use experiments designed by other scientists.

Through a practical evaluation of some existing *SWfMSs* (see Chapter 3), and a review of existing literature (e.g.: [Gil 2007, Deelman 2008, Deelman 2009, Zhao 2008a, Bhagwanani 2005]), we found that there are still some weaknesses and challenges need to be considered in every stage of the workflow management lifecycle. As we have mentioned before, discovered challenges can be divided into two groups: System Efficiency and Reliability Challenges (SERC) and System Usability Challenges (SUC); the following sections explore existing challenges in every stage.

Challenges in the Creation Stage

- ***Ease of use (SUC)***: Scientific workflows are usually constructed by scientists themselves. While they are experts in their domains, they are not necessarily experts in information technology. Consequently, they don't possess the necessary knowledge to deal with complex IT terminologies like Web services, Grids, etc. An efficient *SWfMS* should provide an environment with high level of abstraction and a simple workbench in order to isolate such complicated concepts from its users.
- ***Data heterogeneity (SERC)***: Scientific workflows tend to exchange data between heterogeneous tasks. In existing *SWfMSs*, users have to take care of this type of heterogeneity by manually specifying data shims / mediators necessary for data transformation. The main difficulty here is to determine / create the suitable shims / mediators between two heterogeneous services.
- ***Handling security credentials (SUC)***: For accessing secured Grid services, users may need to follow a complicated process in order to obtain required security certificates.
- ***Workflow validation (SUC)***: Scientific workflows tend to be time consuming due to the data intensive and computationally expensive features of their scientific processes. Thus, it is a very important feature of *SWfMSs* to allow its users to inspect and validate workflows before execution to minimize runtime errors, which can happen due to incorrect workflow specifications.

Challenges in the Scheduling Stage

- ***Reliability/Extendibility (SERC)***: Currently available *SWfMSs* manipulate a single scheduler for the scheduling stage. This creates a single point of failure, forms a scalability bottleneck, and often leads to too much message traffic routed back to the coordinator. In other words, a single scheduler reduces the overall system reliability and extendibility.
- ***Load Balancing (SERC)***: Another problem with most of the current available schedulers is ignoring load balancing while mapping abstract tasks onto computational nodes.

Challenges in the Execution Stage

- **Reliability (SERC):** As in the scheduling stage, we face the reliability problem in the execution stage where only one execution engine is exploited for executing submitted workflows. In addition, by utilizing a single engine, the system will not have control over remotely running services and fault handling is achieved in a centralized fashion through the available engine.
- **Data movement (SERC):** Normally, scientific workflows exchange a huge amount of data during their execution. Some available *SWfMSs* uses a mediator-based approach for data transfer where data must be transferred first to a centralized storage and then to the target node, which is completely inefficient. Accordingly, other *SWfMSs* used a peer-to-peer approach where data can be transferred directly from the source node to the target node. Even this approach is sometimes not sufficient for data intensive scientific workflows, as a single scientific task can produce a huge amount of data.
- **Data caching (SERC):** Data caching is a very important feature which has been ignored by available *SWfMSs*. Several scientific workflows are computationally expensive and are based on long running processes. For this type of workflows, the data caching capability helps to achieve smart re-run since scientists generally tend to re-run scientific experiments while changing only the inputs or configuration parameters of few tasks. In this case, just those tasks with modified inputs or parameters will be actually re-executed.
- **Fault handling (SERC):** To make workflows more resilient, faults during the execution of a workflow must be detected and handled on different levels.
- **Checkpointing (SERC):** A checkpointing mechanism stores a snapshot of the workflow execution state in fixed intervals. These snapshots can be used to resume/restart the execution in case of failures. It can help also to achieve smart rerun capability where tasks that were successfully completed before a selected checkpoint may not have to be re-executed.
- **Workflow Monitoring (SUC):** Providing information about the execution state of a running workflow and notifying about any fatal failures are very important features for a reliable *SWfMS*. The main challenge here is to afford the information at the right level of detail in a form that is easily understandable by users.
- **Workflow Steering (SUC):** Scientific workflows require user decisions and interactions at various steps. Thus, *SWfMS* should allow users to inspect and modify intermediate results before feeding them to subsequent steps.

Challenges in the Re-run/Re-use Stage

- **Workflow Sharing (SUC):** For composing a workflow, scientists often incorporate portions of existing workflows, and just make changes where necessary. Thus, workflows created by other scientists should be stored in a global shared workspace from which they can be searched and retrieved.
- **Smart-Rerun (SUC):** Scientists tend to rerun scientific experiments after changing some parameters. A "smart" rerun would not execute the workflow from scratch, but only those parts affected by the change. This can be easily achieved if we can afford previously mentioned challenges: data caching and checkpointing.
- **Expose workflows as Grid services (SUC):** To act as a scientific collaboration platform, the *SWfMS* should allow its users to expose their workflows as Grid services. This will lead to a better reuse where scientists can utilize these services as atomic tasks in more complex workflows.

Summary

This chapter provides some basic knowledge about Grid-based scientific workflows and investigates the challenges of managing and executing them. First, we introduced GRIA as an example of a service-oriented Grid infrastructure. Then, we discussed the Montage scientific application, an astronomical image mosaic service, and the structure of its scientific workflows. Afterwards, we provided an overview about *SWfMSs* and their reference architecture. Last but not least, we identified a set of challenges and missing requirements in the different stages of the workflow management lifecycle. We categorized these challenges into two groups: System efficiency and reliability challenges that are related to the server-side of the *SWfMS*, and system usability challenges that are relevant to the client-side component (User Workbench) of the *SWfMS*. The list of challenges will form our main criterion for comparing some of the existing *SWfMSs* in the next chapter.

Related Work and Motivation for SWIMS

Contents

3.1	Scientific Workflows Management Systems	19
3.2	Scientific Workflows Workbenches	24
3.3	Discussion and Motivation: The SWIMS Framework	30

This chapter focuses mainly on existing research in the area of scientific workflow management and execution. There are many *SWfMS* currently available in the market. In this chapter, we are going to compare a set of them selected from different categories. We have selected systems dedicated to specific Grid infrastructures (e.g., Unicore [UNICORE 2011]). Other systems were directed to special domains, for example, Taverna [Taverna 2011] which is used for Bio-Informatics and Kepler [Kepler 2011] that was dedicated in the beginning for statistical analysis. Askalon [Fahringer 2011] has been considered as an example of systems based on a service-oriented architecture. Swift [Swift 2011] is another system, that offers a scripting language and provides a virtual data system. Another interesting *SWfMS* which has been also inspected is Pegasus [Pegasus 2011].

The chapter is divided into three sections. In the first section, we investigate the server-side components of the selected *WfMSs* against the system efficiency and reliability challenges (see Section 2.4). Afterwards, we compare the *SWfMSs*' client-side components according to the system usability challenges (see Section 2.4). Last but not least and based on conducted comparisons, we motivate our work to develop SWIMS, an advanced environment for scientific workflow management and execution, that helps to overcome both challenges.

3.1 Scientific Workflows Management Systems

In this section, we compare the set of selected *SWfMSs*' server-side components against the system efficiency and reliability challenges discussed in Section 2.4. The results of this comparison are summarized in Table 3.1.

Taverna

Taverna [Oinn 2006, Hull 2006] is a tool for designing and executing scientific workflows. In the first generation of Taverna (named T1.x), XScufl (Simple Conceptual Unified Flow Language) was the XML-based workflow description language used to construct concrete workflows that has to be executed through the centralized Freefluo enactor [Freefluo 2009]. To meet the evolving requirements of scientific applications, Taverna architecture has been radically re-designed (T2.x) [Missier 2010] providing a new centralized enactor and a new description language.

In T1.x, data produced and consumed by services needs to be entirely loaded into the enactor's process space (centralized approach). On the contrary, the T2.x data architecture is based on the principle that data is only loaded into the execution process space on demand. This can be achieved through the Taverna's Data Manager (DM). The DM indexes data items produced by assigning to them a unique URI and store them into a database, from where other tasks can retrieve them using their references (mediated approach). Data transformations in Taverna can be achieved by composing existing "shim" services or by using "Beanshell" scripts to build specific transformation components.

Taverna supports fault handling through a configurable mechanism. On the task level, users can specify a number of retries and alternate tasks. On the workflow level, users can determine non-critical tasks where execution can be continued even in case of a failure.

Kepler

Kepler [Ludaescher 2006] is a scientific workflow environment based on the PtolemyII system [Ptolemy 2004], a platform supporting multiple models of computation suited to distinct types of analysis. Kepler is based on an actor-oriented paradigm where actors correspond to re-usable workflow components. It uses the Modeling Markup Language (MoML) [Lee 2000] as a workflow specification language. MoML does not provide any control flow constructs. However, flow controls are supported by Kepler's components (actors, directors).

To optimize data flows Kepler allows working with remote data in three ways: GridFTP [GridFTP 2011], Storage Resource Broker (SRB) [SRB 2011] and the scp, which is a shell command that helps users to copy files between systems. Two mechanisms for data integration are applied in Kepler. One is a set of special actors which work as "shims" for data transformations; the other is to convert data into a common data model, the Ecological Metadata Language (EML) [Fegraus 2005], which is only used for the ecological domain.

Kepler provides a centralized workflow execution service through the Ptolemy engine. It also provides an attempt for distributed execution on multiple independent machines through its Master-Slave module [Wang 2008, Wang 2009]. In the illustrated distributed framework, Workflow execution is initiated by a Master node that performs overall coordination of an arbitrary number of Slave nodes that

execute subworkflow tasks. Users have to determine the portions of their workflows, which need to be distributed and model them using the *DistributedCompositeActor*. For every *DistributedCompositeActor*, users have to specify which slaves will be used for this particular actor.

Kepler has no generic capabilities for fault tolerance. It provides only the workflow rescue mechanism which tries to continue the execution of the workflow in case of a failure. In [Mouallem 2010], a fault tolerance framework for Kepler-based workflows has been presented. Nevertheless, this framework has not been integrated in the current version of Kepler.

Unicore

UNICORE 6 [Streit 2010] is a Grid Computing technology, based on the Open Grid Services Architecture (OGSA) specifications, that provides seamless access to distributed Grid resources. BPEL [Scherp 2010] and JSDL [Anjomshoaa 2005] languages are used to describe Non-DAG abstract/concrete workflows in UNICORE. JSDL is used to describe requirements of computational jobs for submission to a specific resource. UNICORE has a resource broker which is capable of distributing jobs to suitable target systems. In this process, specified resource requirements of the job are compared to the target systems' offerings for finding a computing resource that fulfills the specified requirements.

The central component for job and data management inside UNICORE is UNICORE/X. It consists of a Web service engine, and an execution management system (XNJS) which handles the job execution and data management. UNICORE/X offers data transfer using the OGSA random access ByteIO (a slower but widely supported protocol) [Morgan 2005] and the baseline file transfer (a fast HTTP based protocol). For data staging, GridFTP is supported [Rambadt 2008]. In Unicore, data can be integrated using user defined Gridbeans.

Fault tolerance in UNICORE is handled internally by the XNJS that can recognize data movement, input availability, task failures and user defined exceptions. To accomplish failure recovery, the XNJS is used to retry failed jobs on the same resource.

Pegasus

Pegasus (Planning for Execution in Grids) [Deelman 2005, Lee 2008] is designed to map abstract DAG workflows over a wide range of execution environment, including a cluster, or a Grid. To avoid re-running of complex computations, Pegasus tries to simplify the abstract workflow before mapping it into a concrete one. The simplification process tries to reduce the abstract workflow by reusing a materialized dataset which is produced by other users. However, the reduction process is not based on data caching concepts. It is mainly based on logical file names defined in the abstract workflow. In practical, this technique can be inefficient as users may map their physical data to different logical representations.

Pegasus proposes a just in-time planning based on its mapper component. The planning process is based on several Grid information services: the Globus Monitoring and Discovery Service (MDS) [Fitzgerald 2001] to discover available compute resources, the Globus Replica Location Service (RLS) [Chervenak 2002] to discover data locations, and the Transformation Catalog [Deelman 2001] to determine where the application executables are installed. Pegasus includes four basic scheduling algorithms based on information on the predicted execution time of the tasks and data access as well as on information about the resources: Heterogeneous Earliest Finish Time (HEFT) [Topcuoglu 2002], min-min [Blythe 2005], round-robin, and random. Concrete workflows are executed over the Grid through the centralized Condor's DAGMan meta-scheduler [Thain 2005] which submits jobs to Condor-G [Frey 2002] for execution. Pegasus exploits a mediated approach for data transfer in which the intermediate data generated at every step is registered in the RLS, so that input files of every task can be obtained by querying the RLS. The remote data staging is based on the GridFTP protocol [Rambadt 2008].

Fault handling in Pegasus is based on the capabilities of the Condor's DAGMan. In case of a job failure, DAGMan can retry it a given number of times or if that fails, DAGMan generates a rescue DAG (based on checkpointing techniques) that can be potentially modified and resubmitted at a later time. A rescue DAG simply skips jobs that have completely finished.

Askalon

Askalon [Fahringer 2005a, Fahringer 2007] is a programming environment of Grid Computing. Askalon's workflows are translated into the Abstract Grid Workflow Language (AGWL) [Fahringer 2005b], an XML-based language. The AGWL representation is then sent to a WSRF-based runtime system for scheduling and reliable execution on a Grid infrastructure.

The Askalon's scheduler and the resource management system (GridARM) components are responsible for mapping the abstract tasks specified in AGWL into concrete one. The GridARM [Siddiqui 2005] is based on the Globus tools and serves as a data repository which affords the scheduler with all information needed for the scheduling process. The scheduler performs a full graph scheduling through one of the implemented scheduling algorithms [Wieczorek 2005]: a genetic algorithm, the HEFT algorithm, and a "myopic" just-in-time algorithm.

The Askalon's Enactment Service is responsible for the actual execution of a concrete workflow. The Enactment Service is based on a distributed enactment engine (DEE) [Duan 2005] that exploits a master-slave architecture model. The master EE partitions the concrete workflow into several subworkflows which are executed by the slave EEs. The master EE monitors the execution of the entire workflow as well as the state of slave EEs. If a slave EE crashes, the master EE reschedules its subworkflow. In the beginning of the execution process, the master EE elects one of the slaves as a backup engine. If the master crashes, the backup

becomes the master and continues the execution process. One clear drawback here is that Askalon performs partitioning after the mapping step (i.e., on concrete workflow). This means static mapping information is used within the created subworkflows. Given the dynamic nature of Grid environments, this approach may not be suitable. To solve this problem, Askalon attains optimizers to refine the created partitions. However, the optimization process may introduce additional overhead to the whole system.

The Enactment service provides fault tolerance on both task and workflow levels. On the task level, retry and replication methods are used, while on the workflow level checkpointing and workflow rescue techniques are used. Created checkpoints are used only internally by the system to recover from failures. Therefore, Askalon's users will not be able to exploit them for smart re-run of their workflows.

Data in Askalon is transferred between different sites over the GridFTP protocol [GridFTP 2011]. In addition, Askalon has introduced a domain oriented approach [Qin 2008], for scientific workflow compositions, which tries to separate the concepts of data meaning and data representation. In this way, data conversions between different data representations are done automatically based on data semantics of available data ports.

Swift

Swift [Zhao 2007] is a system, based on Globus services [Globus 2011], for the specification, execution, and management of large scale scientific workflows. Swift users can create abstract NON-DAG workflows, which can be scheduled for execution by Cog Karjan [von Laszewski 2005] - a centralized execution engine. Swift also integrates Falkon (Fast and Lightweight Task Execution) execution framework [Raicu 2007], that provides support for efficient execution of large numbers of small tasks in Grid environments.

Swift utilizes an adaptive scheduling strategy [Zhao 2008b] that assigns jobs to Grid nodes that can potentially speed up computational analysis on Grids. Swift uses a feedback system to determine the performance score of Grid sites while scheduling. The adaptive scheduler uses a weighted random function to select randomly a Grid site based on the site score. An optimization has been applied to this strategy that incorporates data locality in Swift. Load balancing was mentioned also as another optimization for the Swift system, nevertheless, it was not clear for us how to enable this during our practical evaluation.

Data in Swift is transferred in a peer-to-peer fashion; however, Swift does not provide any mechanism for automatic data transformation. To achieve reliability, Swift provides a set of mechanisms to handle execution faults. If an application execution fails, Swift will try to retry the failed job. Site selection will occur for retried jobs in the same way that it happens for new jobs, which means that retried jobs may run on the same site or on a different site. Swift's users also can enable the workflow rescue mechanism which will try to continue the execution of the workflow after a task failure. If a complete run failed, Swift can resume the workflow from the

point of failure through a restart log file, that will be created in case of a failure. If the run completed successfully, the restart log file is deleted. Thus, we can't consider this feature as a complete checkpointing mechanism, which should allow users to update the workflow and re-run it from any selected point even in case of a successful execution of this workflow.

3.2 Scientific Workflows Workbenches

In this section, we focus on the client-side components (workbenches) of the selected *SWfMSs*'. We study their abilities to support the system usability challenges discussed in Section 2.4. The outcomes of this study are summarized in Table 3.2.

Taverna

Taverna provides a GUI-based desktop application (see Figure 3.1) that users can use to construct concrete workflows. According to its concrete model, Taverna relies on the user to make the choice of resources or services. Taverna in general deals with DAG workflows while providing implicit iterations, which occur when a process expects fewer inputs than it receives. However, T2.x has implemented a limited form of a while loop construct to address a specific problem of interacting with asynchronous services [Missier 2010], which accept a job request and expect the client to check for result availability at some later time.

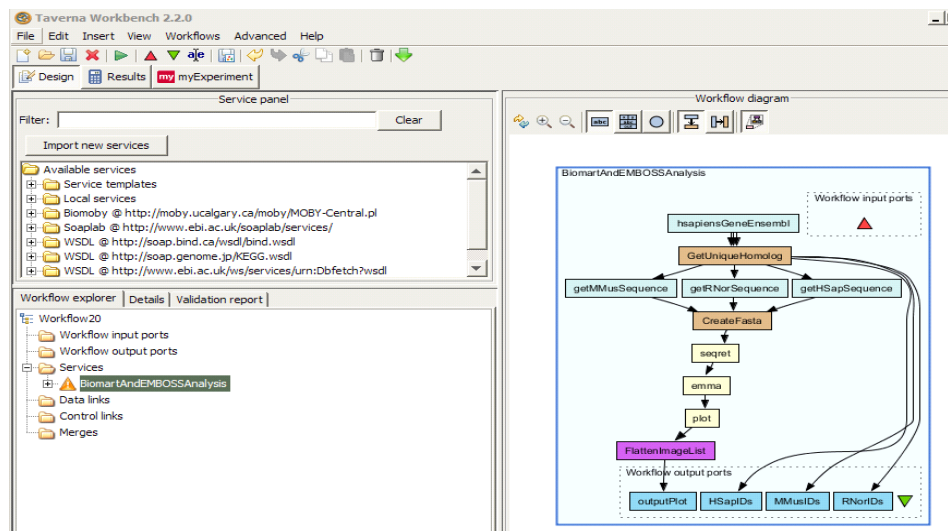


Figure 3.1: The Taverna Workbench

Security credentials in Taverna are handled through its Credential Manager that manages user's credentials and certificates required to access secured services.

Table 3.1: Comparison of Workflow Management Systems (SERC)

SERC	Taverna 2.x	Kepler	Unicore	Pegasus	SWIFT	ASKALON
Reliability	-Single engine -No scheduling	-Single engine -Distributed execution (User directed) -No scheduling	-Single engine -Single scheduler	-Single engine -Single scheduler	-Single engine -Single scheduler	-Distributed execution (Master-Slave) -Single scheduler
Load Balancing	-	-	N/A	N/A	N/A	N/A
Data Movement	mediated	peer-to-peer	peer-to-peer	mediated	peer-to-peer	peer-to-peer based on ontologies and data semantics
Automatic Data Conversion	N/A	-Common model EML	N/A	N/A	N/A	
Fault Handling	-Task Level: -retry -alternate resourc -Wf Level: -wf rescure	-Task Level: N/A -Wf Level: -wf rescue	-Task Level: -retry -user defined Exceptions -Wf Level: N/A	-Task Level: -retry -Wf Level: -rescue DAG	-Task Level: -retry (alternate resource) -Wf Level: -wf rescure -restart	-Task Level: -retry -alternate resource -Wf Level: -wf rescure -checkpointing
Checkpointing	N/A	N/A	N/A	Fault handling	Fault handling	Fault handling
Data Caching	N/A	N/A	N/A	N/A	N/A	N/A

Taverna's workbench keeps the user informed about the progress of their workflows. Users also can interact with the running workflow in terms of stopping, resuming, and canceling it, or inspecting / updating its intermediate results.

Taverna has also begun to share workflows through the myExperiment project [myExperiment 2011] in order to make such workflows available to the community as a whole. Meanwhile, through myExperiment, users can share only workflow specifications; there is no available information for workflows' execution history, checkpoints, or error logs.

Taverna has no capability to expose workflows as Grid services. However, GRIA infrastructure [GRIA 2009] provides tools for deploying and running T1.x workflows as Grid services.

To avoid the execution of problematic workflows, Taverna can check the validity of data types, input and output ports, and scripts involved in services. Taverna also checks whether external Web services are online.

Kepler

Kepler provides a graphical user interface for composing and editing concrete Non-DAG workflows. Users design the workflows using various workflow components, known as actors. They have to determine the semantics of the computation model through *selecting a suitable Director* which imposes an execution order and communication mechanisms on the actors of the workflow; a negative remark here is that selecting the correct Director can be a tedious process for naive CS users. Kepler workflows can be nested, enabling workflow designers to build re-usable, modular subworkflows that can be saved and used for many different applications. Furthermore, Kepler workflows can be saved in an XML representation (MoML) and later passed to Kepler for execution in the absence of the GUI.

Figure 3.2 shows an example of a Kepler workflow that invokes an external Web service and uses two composite actors as data transformation shims. As we can see from the Figure; Kepler is based on a concrete model; users have to specify the URL of the service's WSDL file as well as the required method. Again, this is a clear drawback from the system usability point of view.

Kepler's Web and Grid service actors allow scientists to utilize computational resources on the net in a distributed scientific workflow. Currently, Kepler supports a variety of Grid-based systems, including GriddLeS [Abramson 2005], Globus [Globus 2011], and other systems. As an example, to use Globus services, end users need two X.509 certificates. The first one is the user certificate, which is issued by a certification authority and is used to identify the user. The second certificate is a proxy certificate, which is to support the temporary delegation of the user's privileges to use Grid services.

The "*Animate at Runtime*" option in Kepler helps users to monitor the execution of their workflows by highlighting the actor under processing. Meanwhile, this command works only with the SDF Director. In principle, Kepler workflows

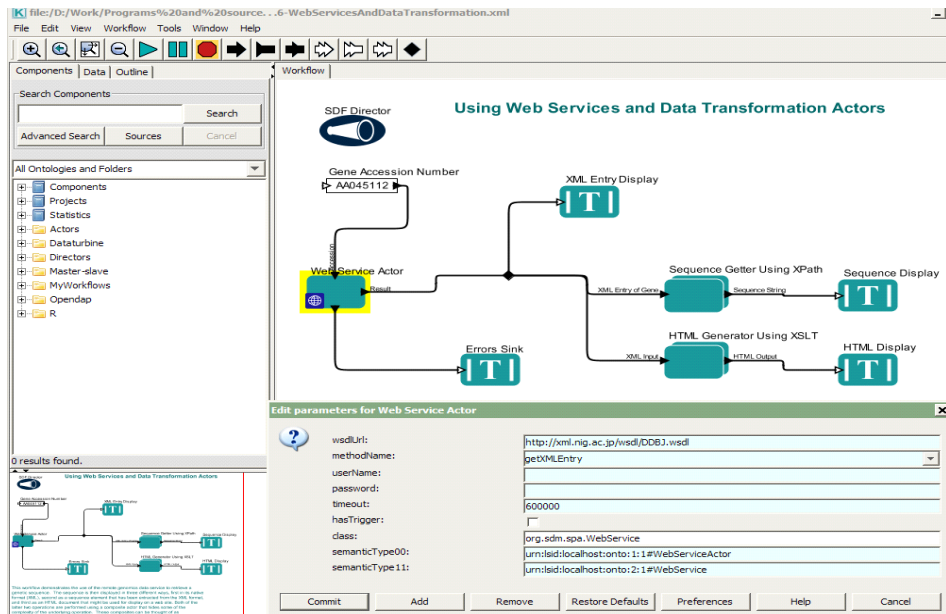


Figure 3.2: Kepler Sample Workflow

can be modified by users during execution, while in practical this can be applied just on the top-level workflow's defined parameters. In Kepler, users can not investigate and modify intermediate results of a certain actor.

Kepler workflows can be exposed to an atomic actor using the Kepler archive format (*KAR*). What is more; Kepler's Component Repository provides a centralized server where components and workflows can be uploaded, downloaded, searched and shared with the community or designated users.

Unicore

On the top layer of UNICORE, a variety of clients is available, ranging from the command-line interface named UCC, to the Eclipse-based UNICORE Rich Client.

The UCC is a command-line tool that allows users to access all features of the UNICORE service layer in a shell or a scripting environment. It allows running jobs, monitoring their status and retrieving generated output. Besides the UCC, UNICORE provides a simple GUI (Rich Client) through which users can design and execute their abstract/concrete workflows (See Figure 3.3).

Before accessing a UNICORE based Grid, each user needs to obtain a valid X.509 certificate [OASIS 2005] which is issued by one of the certificate authorities that the UNICORE servers trust. The client presents this certificate to the server whenever he is asked for authentication.

During the workflow execution, the client displays the execution progress by adding execution state icons to the nodes of the workflow graph. In addition, the user may trace the workflow for finding out where his jobs were submitted.

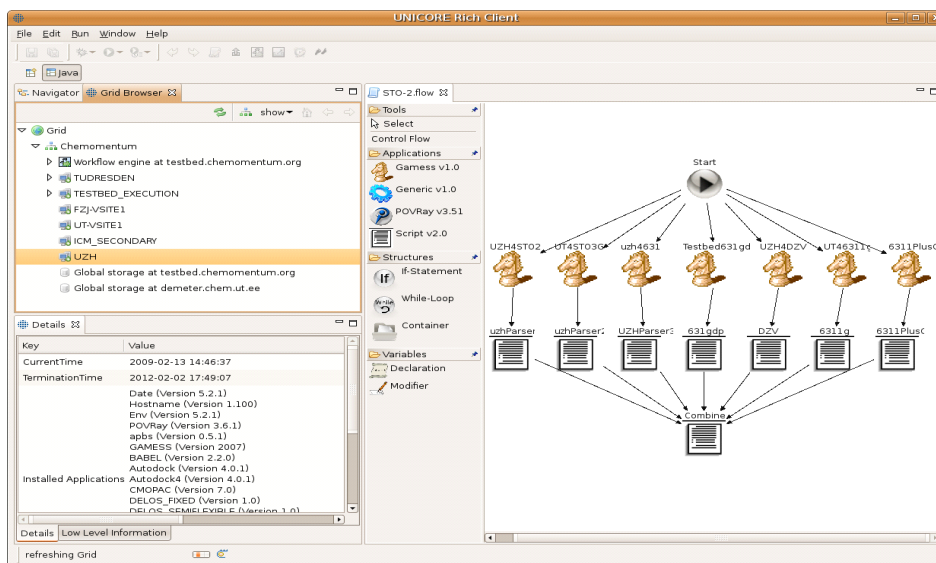


Figure 3.3: Unicore Rich Client

Pegasus

Pegasus's abstract DAG workflows are represented in XML (DAX) format. The DAX format lists the jobs that are to be executed, the inputs and outputs for each of these jobs and the control and data flow dependencies between the jobs. Pegasus's users can generate DAX workflows either using a DAX generation API (in Java, Perl, or Python) or by generating XML directly from their scripts.

Pegasus comes with a set of command-line tools that help users to submit and monitor the progress of abstract workflows and to collect statistics and performance profiles about these workflows.

Authentication in Pegasus is based on the Grid Security Infrastructure (GSI) [Globus 2011]. Users are authenticated using their Grid security credentials. The user first needs to save their proxy credential in the Globus MyProxy server. My-Proxy is a middleware service which maintains user profiles, proxy credentials, policies and preferences [Novotny 2001].

Askalon

In Askalon, users compose their abstract workflows graphically based on the Unified Modeling Language (UML) standard [UML 2011] (see Figure 3.4). The created graph is transformed to the AGWL representation which can express complex workflow graphs containing loops and conditional branches.

Askalon's workbench provides a comprehensive monitoring interface through which the user can observe up-to-date various metrics, that characterize the progress of the overall workflow execution.

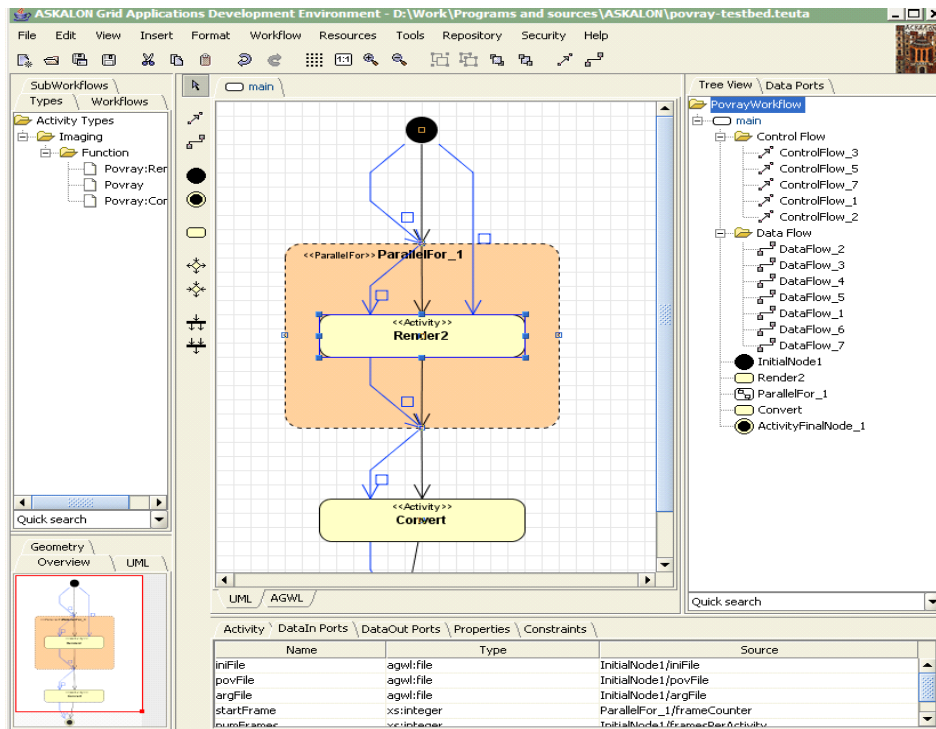


Figure 3.4: Askalon's Workbench

Authorization in Askalon is done via GSI [Globus 2011] and works in coordination with the Globus My-Proxy [Novotny 2001]. The Grid sites are registered by creating a local identity representing the community. This service grants authorization to a user by verifying its access rights for a particular resource ensemble and providing a restricted proxy credential to him.

Swift

Swift's users can create their abstract workflows' specification through a C-like scripting language (SwiftScript [Wilde 2011]). The SwiftScript language builds on XDTM [Moreau 2005] to allow for the definition of typed data structures and procedures that operate on such data structures. The SwiftScript implementation uses mappers to access the corresponding physical data. It also supports arrays and nested iterations. Listing 3.1 shows an example of a SwiftScript which illustrates how SwiftScript utilizes a set of built-in mapping primitives that make a given variable name refer to a file name. Although Swift is based on an abstract model to hide its users from technical issues, we believe that it is a tedious process for naive IT users to learn a new scripting language for modeling their scientific workflows.

For local execution of jobs, no Grid security configuration is necessary. However, when submitting jobs to a remote machine using Globus Toolkit services, Swift makes use of the GSI which requires a certificate/private key pair for authentication.

Listing 3.1: SwiftScript Example

```
file frames [] <filesystem_mapper; pattern="*.jpeg">;
foreach f,ix in frames {
output[ix] = rotate(f, 180);
}
```

During submitting a Swift workflow, the user can specify command-line arguments to show either a graphical or text mode resource monitor which shows the current state of the execution process.

3.3 Discussion and Motivation: The SWIMS Framework

In the context of the comparison of the selected *SWfMSs* against the system efficiency and reliability challenges which has been summarized in Table 3.1, we can see the following:

1. The systems are based on a single scheduler and/or execution engine which violates the reliability / scalability requirement. Kepler has provided an attempt for distributed execution using the Master-Slave paradigm; however, users have to initiate and manage this type of execution. Askalon provided a more advanced distributed execution paradigm where its Enactment Service is responsible for partitioning the concrete workflow into several subworkflows, which can be executed by several enactment engines. However, it depends on a single scheduler which is responsible for mapping the abstract workflow created by the user into a concrete one before the execution process can be started.
2. Some exiting systems still use the mediated approach for data transfer (e.g. Taverna and Pegasus). Regarding data integration, automatic data transformation between heterogeneous services has been introduced only in Askalon while it has been partially supported in Kepler (limited to ecological data) through exploiting a common data model (EML).
3. Fault handling techniques is still amateur in Kepler and Unicore. Checkpointing has been applied as a fault handling technique in Pegasus, Askalon, and Swift; on the other hand, users of these systems can't make benefit of the captured checkpoints to achieve "smart re-run".
4. Remaining requirements, including load balancing and data caching are not supported by any of the compared *SWfMSs*.

Table 3.2: Comparison of Workflow Management Systems (SUC)

SUC	Taverna 2.x	Kepler	Unicore	Pegasus	SWIFT	ASKALON
Ease of Use	- Structure: DAG - Model: concrete - Composition: graph-based	- Structure: Non-DAG - Model: concrete - Composition: graph-based	- Structure: Non-DAG - Model: abstract/ concrete - Composition: graph-based	- Structure: DAG - Model: abstract - Composition: language-based	- Structure: Non-DAG - Model: abstract - Composition: language-based	- Structure: Non-DAG - Model: abstract - Composition: graph-based (UML)
Handling Security Credentials	X.509 certificates	X.509 certificates	X.509 certificates	Globus My-Proxy	X.509 certificates	Globus My-Proxy
Workflow Monitoring	-Graphical Mode	-Graphical Mode (SDF Directory)	-Graphical Mode	-Text Mode	-Graphical Mode -Text Mode	-Graphical Mode
Workflow Steering	supported	Workflow parameters	N/A	N/A	N/A	N/A
Workflow Sharing	myExperiment -Workflow Source	Kepler repository -Workflow Source	N/A	N/A	N/A	N/A
Expose Workflows as Services	N/A	KAR archives	N/A	N/A	N/A	N/A
Workflow Validation	-Data types -Scripts -Services availability	N/A	N/A	N/A	N/A	N/A

Moving to the comparison of the selected *SWfMSs* against the system usability challenges which has been summarized in Table 3.2, we can conclude the following:

1. Some of the evaluated *SWfMSs* are based on concrete workflow models (Taverna, Kepler) which are too complex for scientists to construct and manage even with the fact that such systems afford visual tools to construct and manage their workflows. Other *SWfMSs* are more complex by not providing a visual tool and requiring its users to learn a new scripting language (SWIFT) / generation API (Pegasus). Askalon provides a visual tool for its user in order to construct abstract workflows. This hides Askalon's users from unnecessary complexities. Nevertheless, Askalon has used UML to represent their abstract workflows diagram, which is a computer terminology complex to be understood and managed by non-IT experts.
2. For all evaluated *SWfMSs*, users have to manage X.509 certificates in order to access secured Grid services. This can be a complicated process for naive users with limited IT knowledge.
3. Workflow monitoring is a common feature for almost all available *SWfMSs*. However, workflow steering was fully supported only in Taverna and partially supported in Kepler (only for workflow parameters).
4. Sharing of workflows is possible in both Taverna (myExperiment) and Kepler (Kepler repository); nevertheless, in both cases only workflow specifications are shared between users without any execution information (output results, error logs, etc.).
5. Exposing workflows as services and workflow validation features were limited to Kepler (KAR archives) and Taverna respectively.

Based on the above discussion, we can argue that existing *SWfMSs* still lack some key features, that are required to fulfill all mentioned challenges. In other words, the evolving complexity of scientific applications needs to be reflected better by introducing a *SWfMS* that is possible to provide a new decentralized execution paradigm and other salient features, which can help to overcome these challenges.

In this sense, we have developed the SWIMS framework; a four layered architecture (see Figure 3.5) for scientific workflows management and decentralized execution [Shumilov 2008, El-Gayyar 2010, El-Gayyar 2009]. In the *Workflow Composition Layer*, scientists should be able to create semantically annotated abstract workflows. Annotations in the abstract workflow provide references to a set of ontologies, which can be used in the next two layers to generate the semi-concrete workflow which contains references to mediators required for data transformations between heterogeneous services. The "semi-concrete" here means that workflows still lack execution information that will be obtained later during the execution process. Mediators are created in the *Mapping Layer*, through semantic matching and semantic mapping operations, and indexed in the Mediator Catalog

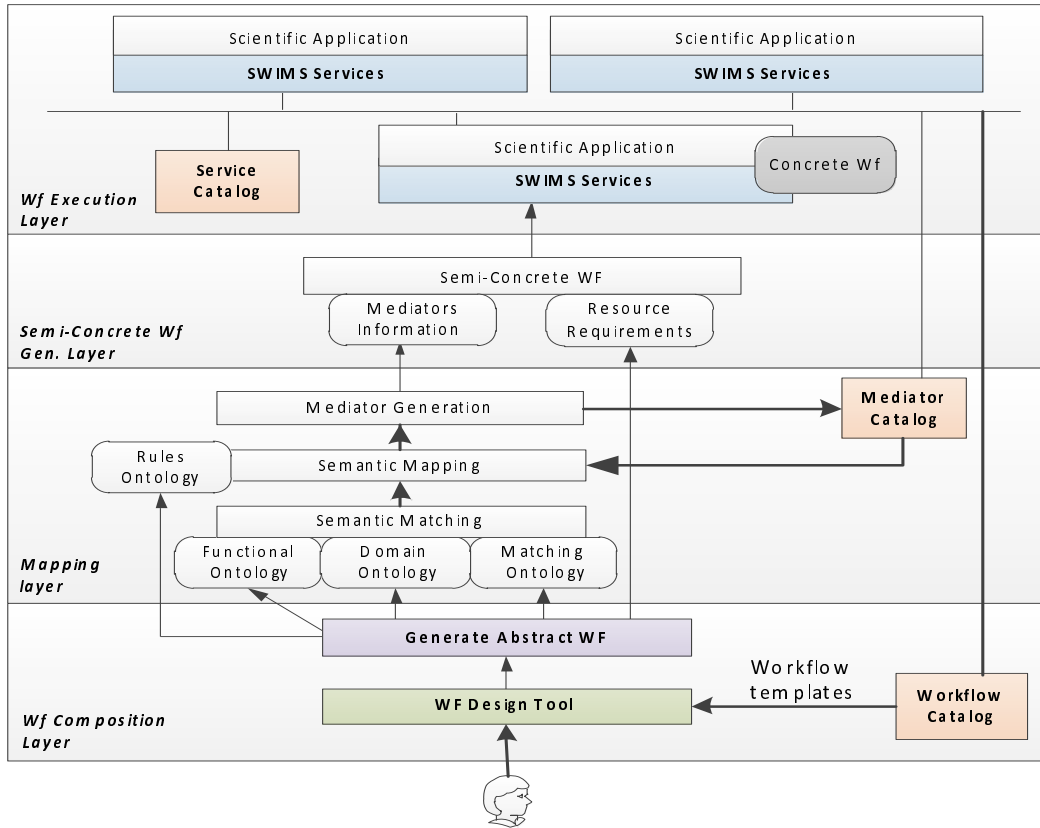


Figure 3.5: Architecture of SWIMS Framework

[Leng 2009]. The semi-concrete workflow is then created in the third layer and passed to the Workflow Execution Layer where it can be concretized and executed in a decentralized fashion.

The mediators' construction process is out of the scope of this thesis. In the coming chapters, we are going to give more details about the features, design, implementation and evaluation, of the workflow composition workbench (1st layer) and the execution environment (4th layer). SWIMS provides a set of novel features trying to overcome discussed challenges: i) support for distributed execution and management of workflows, ii) full control over long running remote services, iii) dynamic data transformation via generated mediators, iv) support for smart re-run through data caching and checkpointing, v) distributed fault handling and load balancing, vi) ease of use based on high level of abstraction, and vii) extensive sharing of scientific workflows and capabilities for re-running and re-using them.

SWIMS: General Design

Contents

4.1	SWIMS Architecture	34
4.2	Mediator Catalog	37
4.3	Workflow Catalog	37
4.4	Service Catalog	40

The SWIMS framework employs Web services technology to originate an execution paradigm that ensures distributed management and execution of data-intensive scientific workflows.

The goal of this Chapter is to introduce the overall architecture of SWIMS and to highlight its main components. SWIMS components fall into three categories: client-side components, server-side components, and global components accessible by all servers and clients in the underlying Cyberinfrastructure. This Chapter also discusses SWIMS's global components in more detail. Server-side and client-side components will be discussed in Chapter 5 and Chapter 6 respectively.

4.1 SWIMS Architecture

SWIMS is optimized to help scientists to efficiently construct, execute, monitor, steer and re-use scientific workflows over the underlying Cyberinfrastructure. Figure 4.1 presents the overall architecture of SWIMS. Mainly, it consists of a client workbench [client-side components], several instances of WMS (Workflow Management System) [server-side components], and three global Catalogs [global components]. The SWIMS WMS component should be deployed on every node wishing to participate in the workflow management and execution process. Every WMS instance consists of four subcomponents: Scheduler, execution, node management and data management.

To understand how all these components cooperate in order to provide a fully distributed management and execution of scientific workflows, we present in Figure 4.2, the flow of the workflow management and execution process in SWIMS. In general, SWIMS deals with services of two different levels of granularity: Concrete services deployed on existing execution nodes and abstract services presented to users in the SWIMS workbench. The SWIMS environment becomes aware of all available concrete services through the WMS's node management subcomponent

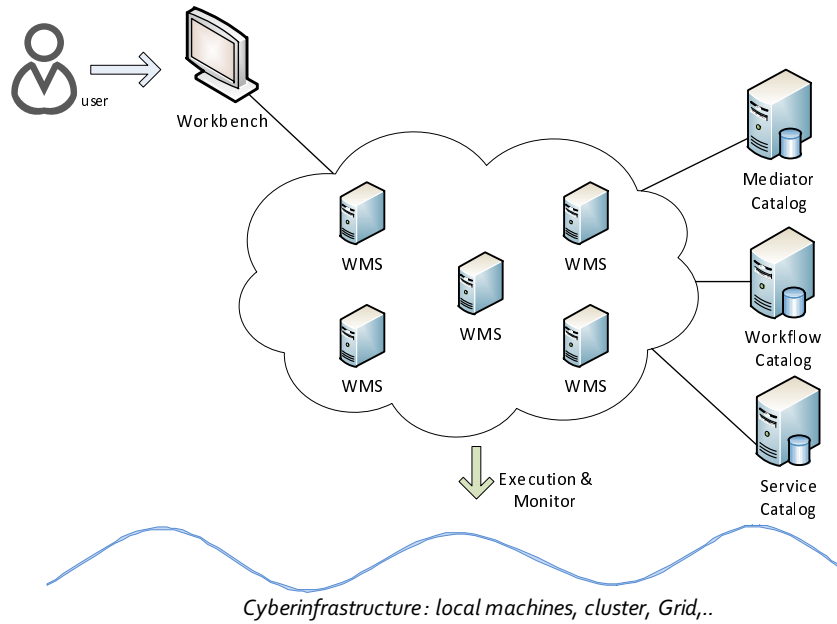


Figure 4.1: SWIMS Architecture

that monitors its underlying node and sends the information about available service to the Service Catalog. This information is used by the SWIMS workbench to construct the list of abstract services that can be used by users to build their abstract workflow specifications. An abstract workflow specification is a set of abstract services connected together through control and data flow dependencies. Users can construct their abstract models either from scratch or starting from a workflow template stored in the Workflow Catalog. SWIMS construct a new workflow template for every execution of a new workflow model.

Whenever the user submits a workflow for execution, SWIMS elects a WMS instance to act as the *main scheduler* of the submitted workflow (through its scheduler subcomponent). The *main scheduler* is responsible for controlling the order of the execution process in the way that ensures the data and control dependencies defined in the workflow specifications. With the help of the Service Catalog, it maps the abstract services to its corresponding concrete ones and selects a certain node for the execution of every concrete service.

The execution of a submitted task is done by the WMS's execution subcomponent in an autonomic way. The execution subcomponent may need to retrieve a data transformation mediator from the Mediator Catalog and apply it on the input data before invoking the concrete service. In case of a failure, the execution subcomponent makes use of its node local scheduler in order to find an alternative node capable of executing the failed task; then, it transparently transfers it to the selected node and sends an update information to the *main scheduler*; it also stores an error report about the failure in the Workflow Catalog.

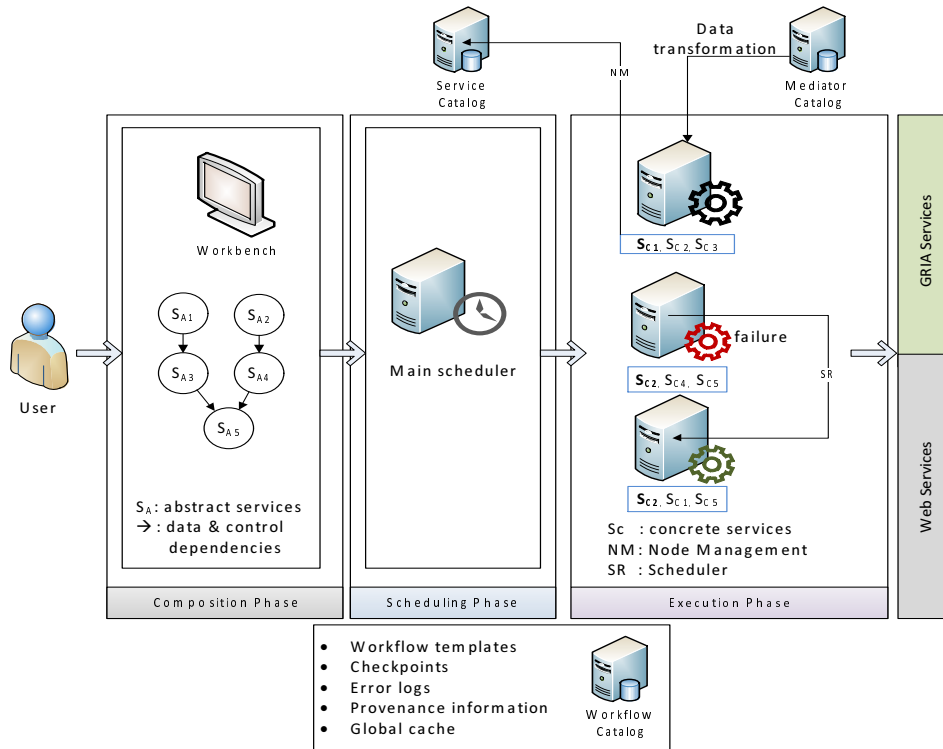


Figure 4.2: The Flow of the Workflow Management Process in SWIMS

After each successful execution of a task, the *main scheduler* stores a checkpoint in the Workflow Catalog. This checkpoint holds a snapshot of the execution state, and a provenance report generated during the execution of this task. Although having a main coordinator for each workflow, SWIMS does not face the single point of failure problem. As in SWIMS, in case that the workflow's main coordinator is down; the workbench transparently elects another WMS instance to continue the coordination of the broken workflow starting from the last recorded execution checkpoint.

SWIMS deals mainly with two types of concrete services: Web services and Grid infrastructure (GRIA) services. The main reasons behind preferring GRIA over other existing Grid middleware are the following:

- **Flexibility:** The GRIA architecture is flexible and can run jobs on a variety of underlying computing platforms: single computers, clusters of workstations or even supercomputers.
- **Interoperability:** GRIA software uses Web services interfaces and is, thus, able to interoperate with other systems through these interfaces.
- **Workflow support:** GRIA provides JAVA API for clients that help to integrate it into different workflow systems. As an example, GRIA

has already provided a Workflow Plugin for XScufl workflows, that can be executed using the Freefluo workflow enactment engine. Moreover, GRIA provides a workflow deployer application that enables the automatic conversion of a user supplied XScufl workflow into a GRIA application, which is then automatically deployed onto a GRIA Job Service.

In the rest of this chapter, we will investigate the SWIMS's three global Catalogs in more depth.

4.2 Mediator Catalog

The Mediator Catalog stores a set of mediators required for data transformation between heterogeneous services. A WMS's execution subcomponent can make use of these mediators during the execution process in order to transform the data to the format required by the target concrete service. The retrieval of necessary mediators and the data transformation processes run in the background without any form of user interaction.

The mediators can be manually created or generated by a semi-automatic system such as the Semantically Enriched Integration System (SEIS) . SEIS is an ontology-based framework grounded from the work presented in [Radetzki 2004, Radetzki 2006]. It tries to enhance the standard Web services technology with semantics and annotations. SEIS generates the required data mediator through a set of semantic matching and mapping operations over the WSDL files of the two underlying heterogeneous services. It can ask for a user advice in case that it can't find an atomic transformation function which solves a certain heterogeneity [Leng 2009]. SEIS also utilizes the Web services added annotations to provide a semantic-enabled composition framework entitled SECPlanner [Leng 2010]. The SECPlanner combines the AI Planning Graph technique with semantics enabled matchmaking algorithm to find the optimal composition candidates of the available services. What is important to be noticed here is that all mediators are created in the composition phase; as a deduction, the construction of data transformation mediators has no impact and doesn't provide any overhead to the workflow's management and execution process.

Describing exactly how the mediators are generated and the details of the SECPlanner are out of the scope of this thesis. However, we try here to emphasize the main role of the Mediator Catalog in SWIMS, that acts as a central repository for data transformation mediators that need to be created once and can be used several times.

4.3 Workflow Catalog

SWIMS uses the Workflow Catalog to form a shared workspace between its users. The Workflow Catalog uses an XML database to index execution snapshots of

workflows executed over the underlying Cyberinfrastructure. Listing 4.1 shows the Workflow Catalog’s database structure. It comprises a list of workflow entries. Each workflow entry is identified by a workflow name and the name of the host that has started the execution of this workflow.

Listing 4.1: Workflow Catalog XML DB Structure

Workflow_Catalog	= [Wf_Entry] (e.g. "gridexp7-messier31.0.1")
Wf_Entry	= [Execution_Snapshot]
Execution_Snapshot	= [workflow_xml, [Subworkflows], data_dependencies, control_dependencies, [Checkpoint]]
Checkpoint	= [subworkflow_xml, executed_subworkflows_IDs, provenance_report, error_report]

Every time a workflow is executed, a new execution snapshot is added into its corresponding workflow entry. Hence, there is a high probability that the user may change the workflow model from one execution to another, each execution snapshot stores the workflow’s XML source. As we are going to see later, the SWIMS WMS breaks a submitted workflow into a set of subworkflows, submits them for execution over available nodes, and creates a checkpoint for every successfully executed subworkflow. Accordingly, an execution snapshot stores a list of the generated subworkflows, the data and control dependencies between them and a list of the constructed checkpoints.

Each checkpoint holds three to four documents. The first document represents the subworkflow XML. The second document lists the identifiers of the already executed subworkflows until this checkpoint; this document helps the system to re-start the execution of a workflow from a certain checkpoint. The third document holds the provenance information collected during the execution of the checkpoint’s subworkflow. Listing 4.2 provides an example of a provenance report. It consists of three sections; the first section gives information about the execution host. The second section provides details about the subworkflow itself, including a list of its activities, its input references, and its output references. Last but not least, the third section focuses on the execution runtime information. A fourth document (error report) is added in case of an execution failure.

Due to limited disk space, each workflow entry can store only a limited number of execution snapshots as specified in the Workflow Catalog configuration file. Whenever we reach this limit, a new execution snapshot will replace an old one according to the following rules:

- If there is more than one execution snapshot with the same workflow model:
 - Replace the one with the least number of checkpoints
- Otherwise replace the oldest execution snapshot

Listing 4.2: Example of a Provenance Report

```

<provenanceReport>
  <host>
    <ipAddress>131.220.149.73</ipAddress>
    <noOfCPU>4</noOfCPU>
    <cpuSpeed>2.40 GHz</cpuSpeed>
    <memorySize> 2.0 GB </memorySize>
    <architecture>x86</architecture>
    <operatingSystem>Windows XP</operatingSystem>
  </host>

  <taskDetails>
    <activities>
      <activity>mBackground1</activity>
    </activities>
    <inputs>
      <InputReference InputName=" mProjectPP1_outputProjectedImage">
        https://131.220.149.169/services/DataService#035c1529 -...
      </InputReference>
      <InputReference InputName=" mBgModel_correctionsTbl">
        https://131.220.149.167/services/DataService#035c1527 -...
      </InputReference>
    </inputs>
    <outputs>
      <OutputReference OutputName=" mBackground1_outputCorrectedImage">
        https://131.220.149.73/services/DataService#035c15c9 -...
      </OutputReference>
    </outputs>
  </taskDetails>

  <executionTime>
    <executionStartTimeStamp>2011/02/23 12:19:07</executionStartTimeStamp>
    <cachingTime>0</cachingTime>
    <executionTime>10734</executionTime>
    <totalTime>10734</totalTime>
  </executionTime>
</provenanceReport>

```

Providing such a type of information in a global Catalog allows scientists to re-use previously designed workflows as a template for creating new ones, or to re-run an experiment from a certain checkpoint, or even to analyze the workflow's provenance information for better understanding of its final results. It also helps us to avoid the single point of failure problem and to support the distributed management of workflows, since when a WMS instance, responsible for coordinating the execution of a workflow instance, fails, another WMS instance can use the information stored in the Workflow Catalog to continue the coordination of the broken workflow starting from the last captured checkpoint.

Along with the workflows' execution snapshots, we have decided to employ the Workflow Catalog as a global cache repository. Data caching entries are stored

in the "SWIMSCache" collection within the Workflow Catalog's XML database. Each document in this collection represents a cache entry for a successively running subworkflow. The document size is rather small as we are not caching the real subworkflow's outputs, but we are just caching their references. For more information about data caching, refer to Section 5.6.4.

4.4 Service Catalog

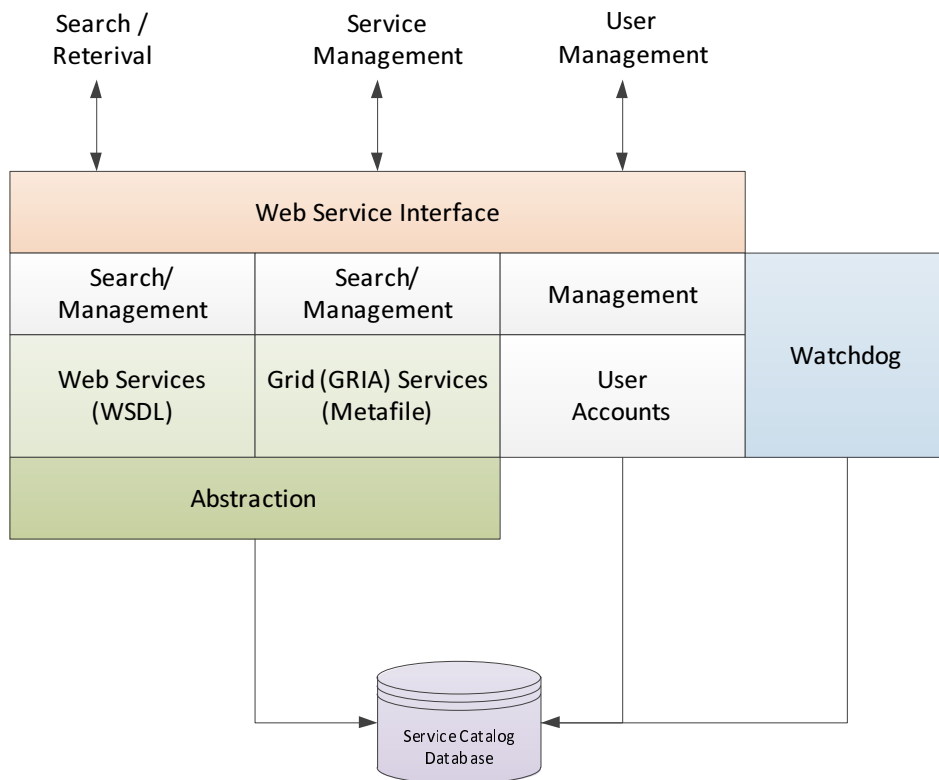
The Service Catalog [Markgraefe 2010] plays an important role in the workflow management and execution processes in SWIMS. It provides four major functionalities that help SWIMS to shield its users from the complications of the underlying systems:

1. Indexing of concrete services
2. High level of abstraction
3. Double-layer security
4. Statistics gathering

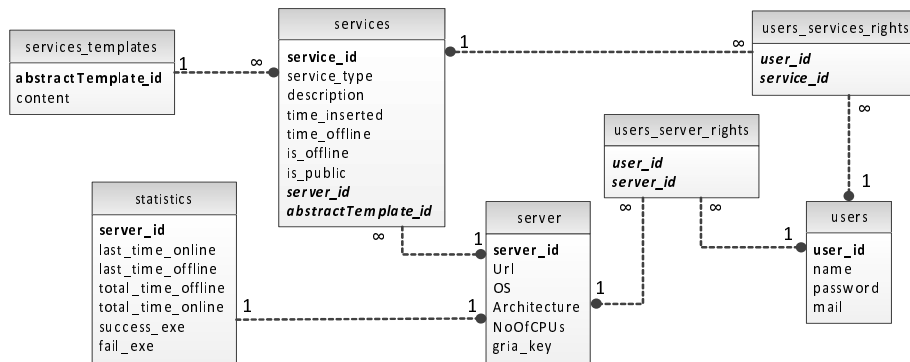
Figure 4.3 shows the overall architecture of the Service Catalog and its database relational model. The Service Catalog interface allows three different classes of functionalities:

- Service Management: server addition/up, server deletion/down, service addition, service deletion, and updating statistics.
- Service Search/Retrieval: retrieving of all available services, retrieving of all available services for a certain user, and searching by keyword.
- User Management: add user, delete user, update user, and assign roles for users.

The Service Catalog keeps track of all available concrete services (Web services and GRIA services) in the underlying Cyberinfrastructure. The availability of services is a major problem for all index services. The study presented in [Al-Masri 2008] investigated the distribution and characteristics of the available Web services on the Web. This study had used a Web Service Crawler Engine (WSCE) , a crawler that is capable of capturing service information from UDDI [UDDI 2004] registries and search engines (e.g. Google, Yahoo, etc.). An intriguing result of this study was that only 63% of the available Web services on the Web are considered to be active. SWIMS tries to avoid this problem and to ensure up-to-dateness by utilizing the WMS instances deployed in every Grid node. The WMS instance keeps monitoring the underlying node for services deployment/undeployment or system shutdown/startup, sends notifications to the Service Catalog in order to update its status.



(a) Overall Architecture



(b) Database Relational Model

Figure 4.3: The Service Catalog Architecture and the Database Model

Upon receiving a server failure or a shutdown notification, the Service Catalog assumes a temporary failure and does not delete the concrete services provided by this server; it only switches their status to offline and sets their offline time. The watchdog component in Figure 4.3(a) is responsible for handling these offline

services. It is a background program responsible for cleaning up and maintaining the Service Catalog. The watchdog runs in fixed periods (as stated in the Service Catalog's configuration file in days). In each run, it performs two tasks:

- Removal of services being offline for a certain period (configuration parameter)
- Ensuring the availability of all other services and switching the status of non-available services to offline.

In case of obtaining a deployment of a new concrete service notification, the notification either holds the WSDL [WSDL 2001] file (for Web services) or the job meta-data file for GRIA jobs. To achieve a high level of abstraction, The Service Catalog is accountable for parsing the attached file to construct an abstract template (e.g. Listing 4.3) for every GRIA job service or for every operation provided by a Web service. The template provides an abstract description of the interface of its task that isolates the workbench/user later from any technical details. The abstract template is obtained by applying an XSL transformation [XSLT 1999] over the obtained file. Appendix B shows the XSL transformation for the GRIA job service's meta-file as an example.

Listing 4.3: Abstract Template for a GRIA Job Service

```
<Activity name="mConvert" type="gria" secured="false">
  <description>
    reduce the size of a FITS file
  </description>
  <inputs>
    <input name="inputImage" type="file">
      <description>...</description>
      <MimeType>image</MimeType>
    </input>
  </inputs>
  <outputs>
    ...
  </outputs>
  <parameters>
    <parameter name="level" type="string">
      <allowed>1</allowed>
      ..
    </parameter>
  </parameters>
</Activity>
```

Another annoying hassle for users of workflow management systems is handling security credentials in order to be able to access secured Grid services. For example, GRIA security is based on the WS-I 1.0 Basic Security Profile [WSI 2007]. Security in GRIA is ensured using a PKI infrastructure based on the X.509 standard [OASIS 2005]. A normal GRIA client will need to create a private key and a

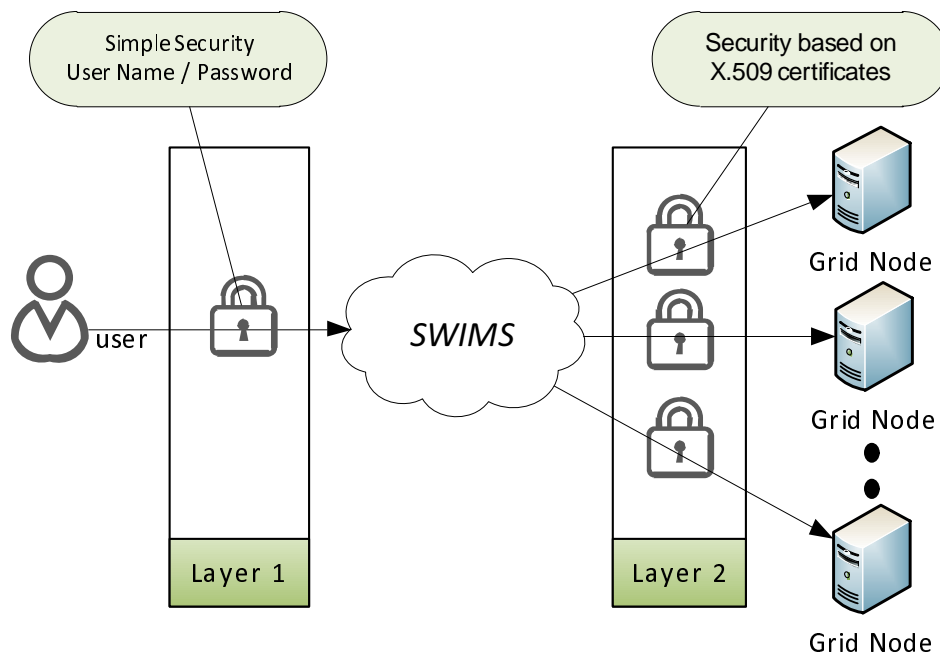


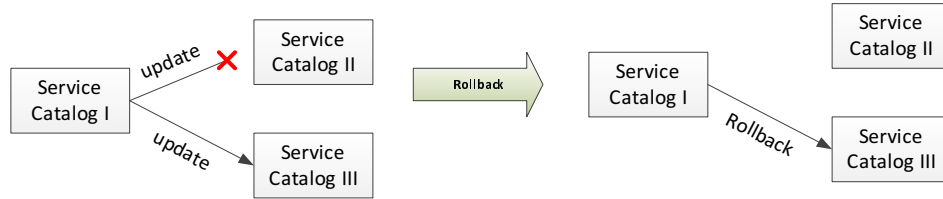
Figure 4.4: Double-Layer Security in SWIMS

certificate for every GRIA server. The certificate needs to be signed by a Certificate Authority trusted by the service providers. This seems to be a complicated process, especially for naive users with limited IT knowledge. SWIMS tries to hurdle the security issue by applying a double-layer security model as shown in Figure 4.4. The first layer defines the security between a user and the SWIMS system that is based on a simple user name and password that have to be defined only once. The second layer represents the security between the SWIMS system and different Grid sites (in our case GRIA servers) based on X.509 certificates automatically generated by the WMS instances deployed on the corresponding servers. The security information is stored and become accessible to all other nodes through the Service Catalog. The administrator of every node should be able, through a simple web-based interface, to classify the node's available services to public and private services, and to assign access roles from the list of available users to their private services. A SWIMS user will be able to execute workflows that contain only public services and private services that have been assigned as accessible to him. By using this approach, we attempt to move the security complications from users to server administrators who in most cases have enough IT knowledge to handle them.

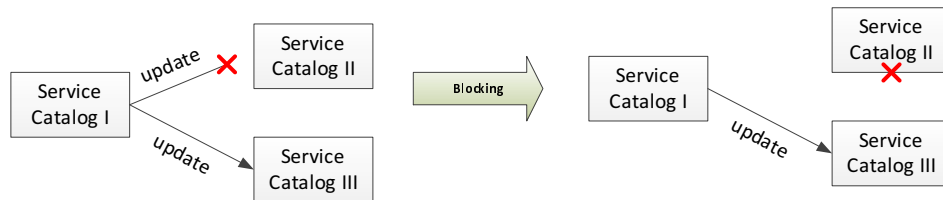
Another important functionality of the Service Catalog is to collect static information (OS, architecture, and number of CPUs) as well as a set of statistics about every available server. For SWIMS, the most important statistics are the number of successfully executed tasks (NSET) and number of failed tasks (NFT). These statistics are collected through notifications received from the WMS server's

component and used to compute the server reliability factor (SRF) (see Equation 4.1); this factor can be used afterwards by a scheduler to select the best node for a given task (see Section 5.4.2).

$$SRF = \frac{NSET}{NSET + NFT} \quad (4.1)$$



(a) Rollback the Failed Update



(b) Block the Inconsistent Instance

Figure 4.5: Methods of Keeping Several Service Catalog Instances Consistent

As far as we can see, the Service Catalog forms the heart of the SWIMS environment as it provides awareness about all available servers and concrete services within the underlying Cyberinfrastructure. Thus, it is highly recommended to maintain more than one instance of the Service Catalog in order to avoid a single point of failure and bottleneck problems. The system administrator can add a new instance by simply adding its URL in the configuration file of one of the available instances. This information and all other updates are forwarded to other instances based on publish and subscribe notification events supported by the Web Services Notification (WSN) framework [WSN 2006]. The administrator also has to specify, in the configuration file, one method to handle errors in instances' updates. This helps to keep all working instances consistent. As shown in Figure 4.5, SWIMS provides two different methods:

- **Rollback:** In this method, the master instance (the instance which has obtained the update notification) sends a rollback notification to all other instances in order to roll back the failed update.
- **Blocking:** In this method, the master instance sends a blocking notification to all instances (including the blocked instance) to block the un-updated

instance and avoid using it for further queries. Whenever a blocked instance receives a query from a SWIMS client, it forwards it directly to a working instance.

In both cases, the failure is logged and a notification is sent to the administrator to maintain the inconsistent instance and to make it available again. A SWIMS client needs to be aware of at least one Service Catalog instance before being able to execute workflows over the underlying infrastructure. Afterwards, the client will automatically receive information about other available instances.

Summary

In this Chapter, we have presented the overall architecture of SWIMS. SWIMS has been designed to enhance the scientific workflow management and execution processes and to shield its users from technical complexities of these processes. SWIMS's components can be classified into three classes: server-side, client-side, and global components.

Furthermore, we have discussed the SWIMS's three global components in detail. First, the Mediator Catalog stores data transformation mediators necessary for data transformations between heterogeneous services. Second, the Workflow Catalog is used to index checkpoints and provenance information of workflows executed over the underlying Cyberinfrastructure. Finally, the Service Catalog keeps track of all available services in the underlying Cyberinfrastructure; moreover, it provides additional functionalities that help to handle security, abstraction, and scheduling issues in SWIMS.

SWIMS Server Side

Contents

5.1	Language for Representation of the SWIMS's WMS Components	47
5.2	Data Management WS-Resource	49
5.3	Node Management WS-Resource	51
5.4	Scheduler WS-Resource	52
5.4.1	Workflow Partitioning	53
5.4.2	Planning	56
5.4.3	Workflow Steering	59
5.5	Execution WS-Resource	60
5.5.1	Provenance Information	62
5.6	Optimization	63
5.6.1	Code Movement - Node Management Resource	63
5.6.2	Tailoring the Service Behavior - Node Management Resource	66
5.6.3	Clustering - Scheduler Resource	66
5.6.4	Global Data Caching - Execution Resource	70
5.6.5	Distributed Fault Handling - Execution Resource	70
5.7	Fault Handling in SWIMS	71

In the last Chapter, we presented the overall architecture of SWIMS and discussed its global components. This Chapter focuses on SWIMS's server-side components: The WMS (Workflow Management System) instances. To simplify the design and the implementation of the WMS, we decided to break it into four major subcomponents: Scheduler, Node Management, Data Management and Execution as shown in Figure 5.1.

Each WMS's subcomponent can be realized as a Web Service Distributed Management (WSDM) [WSDM 2006] based service, which controls a set of manageable resources. Every server node in the underlying Cyberinfrastructure needs to deploy one or more of these subcomponents according to its planned role in the workflow management and execution processes:

- **Workflow coordinator:** It has to deploy at least the Scheduler subcomponent.

5.1. Language for Representation of the SWIMS's WMS Components 47

- **Workflow executor:** It has to deploy the Node Management, Data Management, and Execution subcomponents. In this case, it is highly recommended to deploy the Scheduler subcomponent as well that helps to achieve distributed fault handling as shown later in Section 5.6.5.
- **Workflow coordinator and executor:** It has to deploy all subcomponents.

This Chapter starts with an introduction to the WSDM specification used to implement the different components of the WMS, and then it discusses the main functionalities of each WMS's subcomponent in detail.

5.1 Language for Representation of the SWIMS's WMS Components

Before deciding which technology or standards we are going to use for implementing the WMS's four subcomponents, we have stated a set of requirements that ensure the system flexibility and scalability:

1. **Interoperability:** The subcomponents should be cross platform.
2. **Manageability:** It should be easy to manage the subcomponents as well as other resources used by them.
3. **Communication:** Easy and robust communication is required between the subcomponents themselves and between the subcomponents and other SWIMS's components.
4. **Low overhead:** The WMS should not need massive hardware requirements. In addition, it will be great if its subcomponents are created only when it is needed and released after the completion of its task.

The WSDM specification seems to be a good candidate for achieving these requirements. The WSDM specification is based on the Web services platform and uses open standards to define the methods, structure, and specification of a system for managing different type of resources (e.g. in our case, we are using it to manage a scheduler, a workflow execution engine, a workbench, GRIA servers, and databases). It provides also capabilities for managing Web services used to support the functionality of these resources.

The WSDM specification is made up of two different specifications. Management Using Web Services (MUWS)[[MUWS 2006a](#), [MUWS 2006b](#)] and Management Of Web Services (MOWS)[[MOWS 2006](#)]. While the former specifications lists out what it takes for a resource to be uniformly accessed and managed through a Web service endpoint, the latter allows a Web service endpoint itself to be treated as a WS-Resource. Once a resource is identified as an MUWS manageable resource, it is accessible through a manageable endpoint, which is typically a Web service endpoint [[Kreger 2005](#)]. WSDM relies directly on other standards:

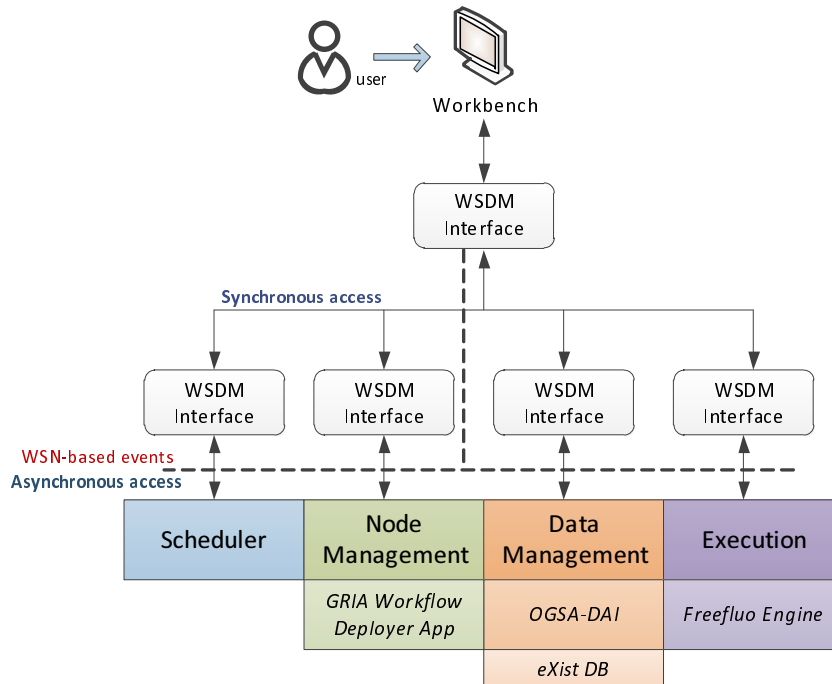


Figure 5.1: SWIMS WMS Services.

- WS-I Basic Profile [[WSI 2006](#)]
- WS-Resource Framework (WSRF) [[WSRF 2006](#)] for the properties and description used to define the capabilities of the WS-Resource.
- WS-Notification (WSN) that defines a set of specifications that standardize the way Web services interact using "Notifications" or "Events".
- WS-Addressing [[WS-Addressing 2006](#)] for service references and to define the endpoints for individual services.

For the implementation of the SWIMS WMS's subcomponents, we are using the Apache Muse Project [[Apache 2007](#)]. Muse is a Java-based implementation of the WSRF, WSN, and WSDM specifications. It is a framework upon which users can build Web service interfaces for manageable resources. Applications built with Muse can be deployed in both Apache Axis2 and OSGi environments, and the project includes a set of command line tools that can generate the proper artifacts for your deployment scenario. In Muse, you can even create or release instances of a WS-Resource programmatically.

Now coming back to our requirements, we can clearly see that WSDM specifications accomplishes all of them. The interoperability requirement is attained through the fact that the WSDM standard uses Web services as a platform which means that you don't have to worry about the technicalities of

platform independence. The second and the fourth requirements are met by the WSDM two specifications: The MUWS and the MOWS. The MUWS helps us to manage available resources, while the MOWS facilitates to reduce the overhead by allowing us to create and release instances of these WS-Resources at runtime. Concerning the communication requirement, as we can see from Figure 5.1, WSDM supports two communication strategies: *Synchronous* communication that can be achieved through the WS-Resource’s manageable endpoint, and *asynchronous* communication based on events supported by the WSN framework. These two strategies afford seamless communication between all SWIMS’s components, including the global Catalogs since they are implemented also as WS-Resources.

5.2 Data Management WS-Resource

OgsaDaiManagement
<pre>+createDataSink() : String +retrieveTransformedData(in ogsaDaiWorkflow : String) : boolean +getDataFromDataSink(in dataSinkID : String) : String +computeInputDigest(in inputReference : String) : String +shutdown() : void</pre>

Figure 5.2: Data Management WS-Resource Interface.

The Data Management WS-Resource (*DMR*) is dedicated to reference-based data movement, and automatic data transformation between heterogeneous services. The *DMR* is based on the Open Grid Service Architecture-Data Access and Integration (OGSA-DAI) , an open source middleware which allows data resources (e.g. relational or XML databases, files) to be federated and accessed via Web services on the Web or within Grids or Clouds. Via these Web services, data can be queried, updated, transformed and combined in various ways [OGSA-DAI 2010, Antonioletti 2007].

In SWIMS, output data is stored in a local eXist-db [Meier 2010], an open-source database management system built using XML technology. The OGSA-DAI wraps the XML database and provides external access to it as a Web service. Figure 5.2 presents the public interface for the *DMR* which reflects two basic functionalities:

1. **Storage of output data:** SWIMS tries to reduce the amount of data transferred within the underlying Cyberinfrastructure through utilizing reference-based data movement. This can be achieved by creating an OGSA-DAI reference for every output stored in the XML database. The reference follows the format "ogsadai_service_url@collection_name:document_id" that consists of three parts; the first part is the URL of the OGSA-DAI service located on the source node; the second part is the collection name in the XML database

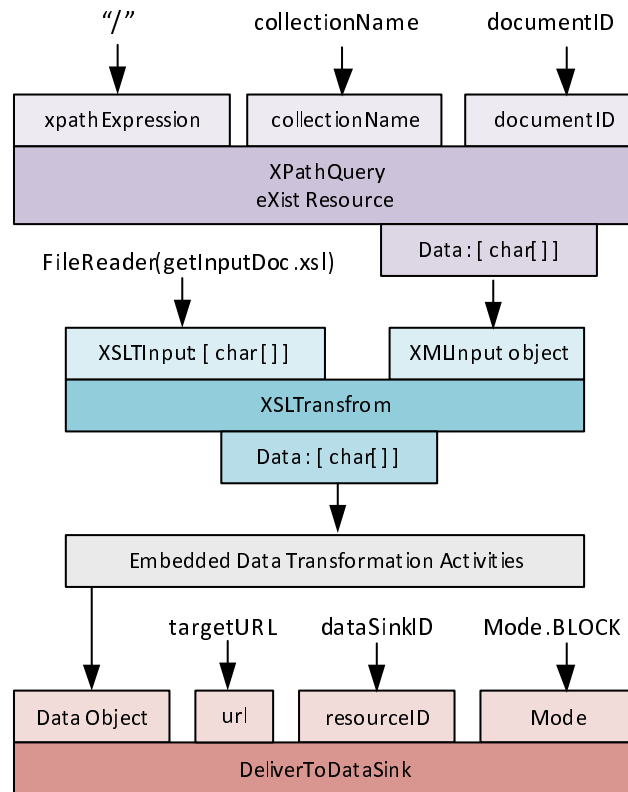


Figure 5.3: OGSA-DAI Workflow for Data Transfer and Transformation.

where the output document is stored, and the third part is the id of the required document. This reference is detected and used to retrieve the data by the Execution WS-Resource (*ER*) instance (see Section 5.5).

2. **Data transfer and transformation:** The *DMR* runs an OGSA-DAI workflow obtained from an *ER* instance (see Section 5.5) to retrieve the requested data from the database, apply required transformation if it exists, and transfer the transformed data to the target node. Figure 5.3 provides a graphical representation for such an OGSA-DAI workflow. The first OGSA-DAI activity in the workflow enacts an XPath expression [XPath 1999] over the local eXist database to retrieve the specified output document. The output from this activity is a series of character arrays known as a ResourceSet, a container for a set of resources. Therefore, the next activity applies a simple XSL transformation over the output to retrieve the original document. Then, the mediator's embedded data transformation activities are applied over the data. Finally, the transformed data is written to a data sink on the target node.

Besides these functionalities, the *DMR's* interface allows SWIMS to create a

new *DMR* instance only when it is needed. Such instance can be destroyed later through the *shutdown* method. Furthermore, to help the data caching process and to avoid unnecessary data movement (see Section 5.6.4), it provides the capability to compute the SHA-2 digest [SHA-2 Standard 2002] of an input stored locally in its underlying Grid node.

5.3 Node Management WS-Resource

DynamicInfoProvider	StaticInfoProvider
+getFreeDiskSpace() : double +getFreeSysMemory() : double +getNumberRunningTasks() : int +increaseNumberRunningTasks() : void +reduceNumberRunningTasks() : void	+getNumberCPUs() : int +getCPUSpeed() : long +getOS() : String +getOSArch() : String
NMRAuxiliary	
+duplicateService(in serviceURI : String, in destinationHostURL : String) : boolean +deployService(in servicePackReference : String) : boolean +deployWorkflowAsGriaService(in workflowXML : String) : boolean	

Figure 5.4: Node Management WS-Resource Interface.

The Node Management WS-Resource (*NMR*) helps SWIMS environment to be aware of existing concrete services without any type of required user interactions. It continuously runs in the background in order to monitor the underlying node trying to keep track of concrete services deployment/undeployment or system shutdown/startup. It notifies the Service Catalog about any detected changes helping it to keep its state up-to-date.

Along with updating the Service Catalog functionality, the *NMR*'s public interface, shown in Figure 5.4, provides other important functionalities for the SWIMS environment:

1. **Support scheduling:** The *NMR* 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). This information is used by the scheduler in order to determine which Grid node is suitable for a given task according to the task specific requirements. The task's requirements can be annotated in the task's description or manually specified by the user during the workflow composition phase. To be more flexible, the *NMR* allows the subscription for a dynamic resource state (memory, disk space, etc.), whenever the state is met, the *NMR* notifies all subscribers.

2. **Deploying workflows as Grid services:** The *NMR* takes advantage of the GRIA Workflow Deployer Application that enables the automatic conversion of a user supplied XScufl workflow [XScufl 2004] into a GRIA application. This tool is responsible for generating GRIA application wrapper scripts and other requisite files from a given workflow which are then automatically deployed onto a GRIA Job Service. The deployment process can be invoked through the SWIMS workbench (see Section 6.3.3).

5.4 Scheduler WS-Resource

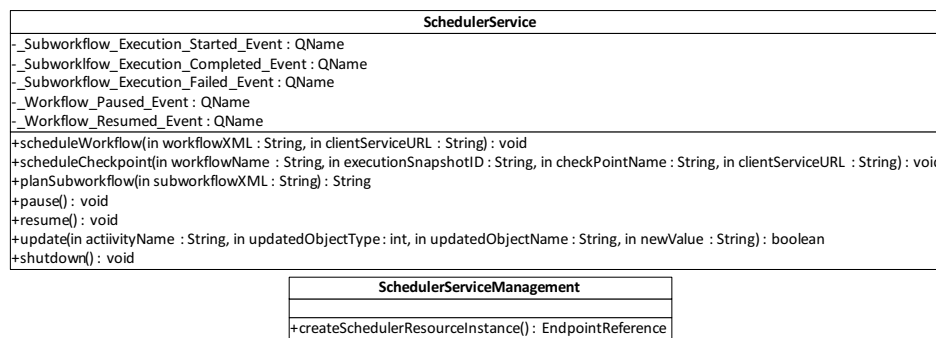


Figure 5.5: Scheduler WS-Resource Interface.

The Scheduler WS-Resource (*SR*) helps to coordinate, monitor, and steer the execution of an abstract workflow. These functionalities are provided through its public interface shown in Figure 5.5. A new *SR* instance (workflow’s *main scheduler*) should be created by a client wishing to submit a workflow for execution. The submitted workflow can be represented by a directed acyclic graph(DAG) $G = (T, E)$. $T = \{t_i, 1 \leq i \leq |T|\}$ is the set of workflow tasks (abstract services) where $|T|$ denotes the number of tasks. The workflow’s tasks are connected to each other through a set of edges $E = \{e_{i,j} = \langle t_i, t_j \rangle, 1 \leq i, j \leq |T|\}$. An edge $e_{i,j}$ indicates that t_j can’t start before t_i is completed due to either data or control dependency.

The *SR* instance follows the lifecycle presented in Figure 5.6 to coordinate and monitor the execution of the submitted workflow. First, the *SR* breaks the submitted workflow into a set of abstract subworkflows and constructs the control and data dependencies tables that reflect the hierarchy of the constructed subworkflows. After partitioning the workflow, the *SR* performs the initial checkpointing phase by adding an ”execution snapshot” entry for the submitted workflow in the Workflow Catalog (see Section 4.3). At the same time it maps ready for execution subworkflows (according to dependencies tables) onto computational Grid nodes. For each mapped subworkflow, the *SR* creates a new Execution WS-Resource (*ER* - see Section 5.5) instance on the Grid node selected during the

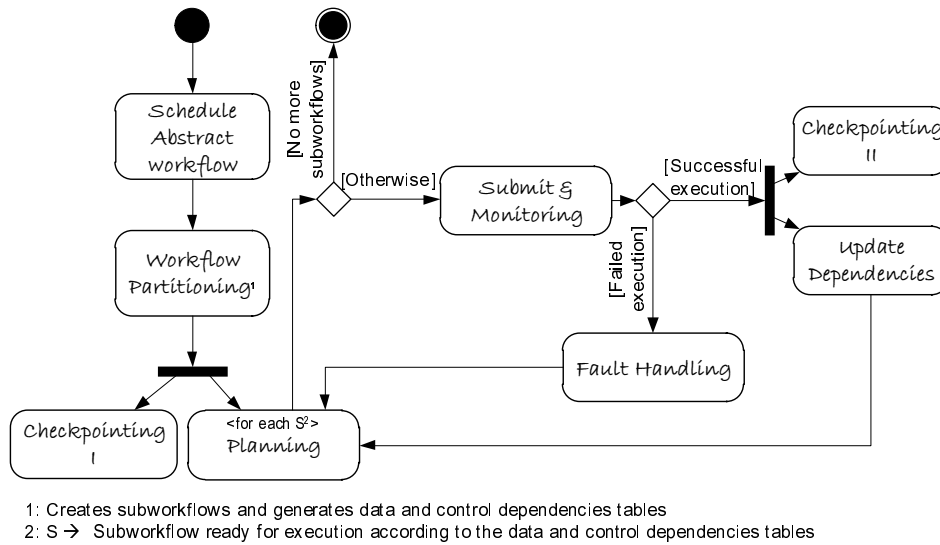


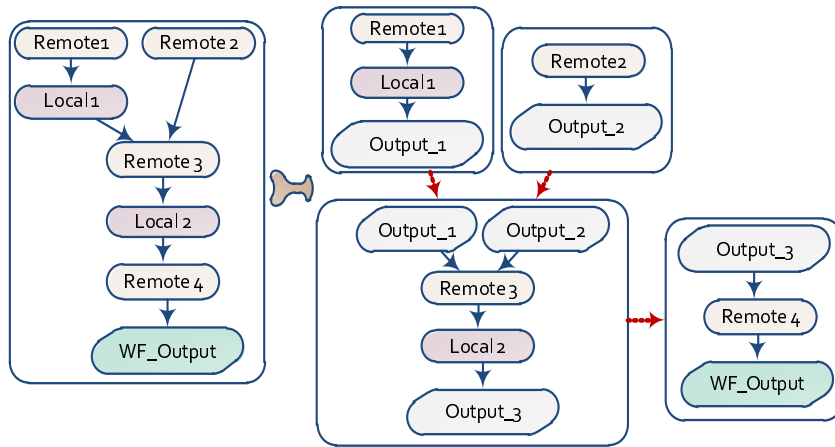
Figure 5.6: Workflow Scheduling Lifecycle [UML 2011]

planning phase and submits the subworkflow to it for execution. For monitoring purposes, the *SR* subscribes itself to the execution status events produced by the constructed *ER* instance. Once receiving an "execution completed" event, the *SR* executes the second checkpointing phase in which it inserts a "checkpoint" entry in the Workflow Catalog. Meantime, it updates the control and data dependencies tables, then it returns to the planning phase for mapping subsequent subworkflows. In case of a failure, the *SR* tries to recover from the error and then switches back to the planning phase. This process is repeated until all subworkflows are successfully executed or an unrecoverable error occurs.

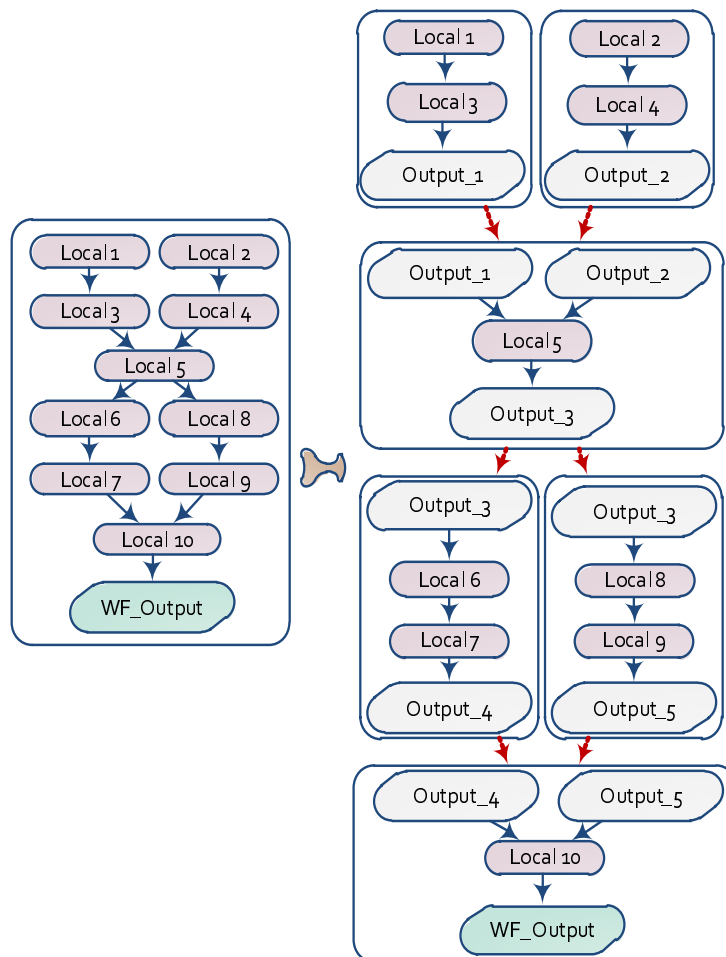
In the following subsections, we are going to discuss the workflow partitioning, and planning phases in more details. Likewise, we are going to introduce the workflow steering capability provided by the *SR*.

5.4.1 Workflow Partitioning

To overcome the scalability as well as the single point of failure problems, SWIMS utilizes several workflow enactors and schedulers to attain distributed management and execution of scientific workflows. This can be achieved by partitioning a submitted abstract workflow into a set of subworkflows. These subworkflows should be distributed among several engines (*ER* instances) which can communicate with each other in order to transfer data based on their provided references.



(a) Single Remote Service per Subworkflow



(b) Consider Parallel Paths

Figure 5.7: Workflow Partitioning

Examples of workflow partitioning in SWIMS are presented in Figure 5.7¹. Our main partitioning criterion is that every subworkflow should have only one remote task (e.g. Web services or Grid services) as shown in Figure 5.7(a). This helps SWIMS to submit each subworkflow to an *ER* instance located on the same Grid node where the remote task is located. As a result, the *ER* will have full control over the task's execution. In all cases, the partitioning algorithm tries to identify and make use of independent paths in the graph to form subworkflows that can run concurrently on different *ER* instances (see Figure 5.7(b)).

Algorithm 5.4.1: PARTITIONWORKFLOW(*wf* : *workflow*)

comment: Constructs a list of subworkflow from a given workflow

procedure PARTITIONWORKFLOW(*WF* : *workflow*)

subworkflowsList $\leftarrow \emptyset$

while \exists unassigned tasks

do $\left\{ \begin{array}{l} \text{for each } t \in WF \\ \text{do if } t \text{ is unassigned and } parents(t) \text{ are assigned} \\ \text{then } \left\{ \begin{array}{l} S \leftarrow \text{empty subworkflow} \\ POPULATESUBWORKFLOW(S, t) \\ ADDINPUTSTOSUBWORKFLOW(S, WF) \\ ADDOUTPUTSTOSUBWORKFLOW(S, WF) \\ subworkflowsList \leftarrow subworkflowsList \cup S \end{array} \right. \end{array} \right.$

CREATEDATADEPEDENCIESTABLE(*WF*)

CREATECONTROLDEPENDENCIESTABLE(*WF*)

procedure POPULATESUBWORKFLOW(*S* : *subworkflow*, *t* : *task*)

if *t* is assigned

then return

if *isRemote*(*t*) **and** *containsRemote*(*S*)

then {**return** (i)}

S $\leftarrow S \cup t$

Mark *t* as assigned

if *number*(*children*(*t*)) > 1

then {**return** (ii)}

c $\leftarrow child$ (*t*)

if *parents*(*c*) $\subset S$

then {POPULATESUBWORKFLOW(*S*, *c*) (iii)}

¹Local tasks are tasks, which can be executed by the execution engine itself, e.g. data encoding tasks.

The pseudo code of the partitioning algorithm is shown in Algorithm 5.4.1, the algorithm goes through the tasks in the submitted workflow, creates a new subworkflow starting from a superior task. A superior task is a task with no parents or all its parents have already been assigned to other subworkflows. For every created subworkflow, the algorithm adds a set of workflow inputs and workflow outputs according to the data dependencies between its assigned tasks and other tasks revealed from the original submitted workflow. After building all subworkflows, the algorithm generates the data and control dependencies tables which reflect the relationship between the generated subworkflows. The data dependencies table is constructed from the relationship between the generated inputs and outputs in each subworkflow while the control dependencies table is built from the original control dependencies defined in the submitted workflow. That is to say, the generated dependencies reflect the original dependencies in the submitted workflow.

A subworkflow is populated starting from a given task, stretching downwards to its children while preserving our partitioning criteria (single remote task) through Statement (i). To identify parallel paths in the workflow, we try to detect two of the scientific workflows' basic structures: data distribution and data aggregation structures [Bharathi 2008]. The output of a data distribution task is consumed by multiple child tasks. Statement (ii) forces the algorithm to stop populating a subworkflow whenever it discovers a distribution task. A Data aggregation task aggregates and process the outputs of several parent tasks. An aggregation task will be added only if all its parents have already been added to the current subworkflow² (Statement (iii)).

5.4.2 Planning

In general, the goal of the planning process is to translate abstract workflows into concrete workflows by mapping the workflow's abstract services into concrete ones. Workflow planning has two different schemes: static scheme and dynamic scheme [Yu 2005]. In a static scheme, tasks are mapped to resources before starting the execution process according to current information about the execution environment; while the dynamically changing state of the resources is not taken into account. A static planning may produce a poor schedule, since Grids are dynamic environments where utilization and availability of resources varies over time. In contrast, a dynamic scheme postpones the mapping of each task until its execution time. In this manner, the mapping process is more adjustable to the dynamic nature of Grid environments. Thus, we have decided to utilize a dynamic (just-in-time) planning schema in SWIMS.

First of all, the scheduler determines which subworkflows are ready for execution according to the data and control dependencies tables. For each ready subworkflow, the scheduler identifies its remote task (abstract service) and the task's attached resource requirements (OS, number of CPUs, etc.) if they exist. Then, the scheduler

²This can be achieved if clustering is enabled (see Section 5.6.3).

sends this information to the Service Catalog in order to retrieve a list of the task's available computational nodes that met the provided requirements.

Upon receiving an inquiry about computational nodes for a specific abstract service, the Service Catalog figures out the list of nodes that deploys the concrete service corresponds to the abstract service. Then, it compares specified service's static requirements against the resource's stored information and discard nodes that don't match any of the specified requirements. If the Service Catalog ended up with an empty list, it will notify the scheduler about a failure due to non-matching requirements / unavailable resources. Otherwise, the Service Catalog contacts the *NMR* of each node in the list to ensue any available dynamic requirements (e.g. available memory, available disk space, etc.) and to retrieve the number of currently running tasks (*task load*). The final list will contain only computational nodes met both the dynamic and the static requirements. Finally, the Service Catalog attaches with each node the value of its reliability factor (*SRF*) (See Section 4.4) and *task load*, and return the final result to the scheduler. According to the result obtained from the Service Catalog, the scheduler will continue the planning process as follows:

- **Failure due to non-matching requirements / unavailable resources:** the scheduler marks the subworkflow as failed and notifies the client about the error.
- **Empty list:** An empty list reflects that the Service Catalog has already found some nodes but none of them fulfills the specified dynamic requirements. In this case, the subworkflow will be added in a queue waiting for a notification from the Service Catalog about available nodes.
- **Non empty list:** the scheduler runs the *node selection algorithm* over the list to decide the best computational node for the underlying task.

There are two well known types of scientific workflows, thus we have decided to provide two different node selection algorithms which in all cases try to put into consideration load balancing between different computational resources. Users can select between the two provided algorithms according to the nature of their workflows:

- **Computation-intensive workflows:** A computationally-intensive workflow consists of several long running tasks. In this case, the cost of task executions dominates the cost of data transfers. Therefore, the node selection process for this type of workflows should focus on nodes' reliability besides the load balancing criterion (Algorithm 5.4.2). The algorithm tries to assign a given task to a reliable node with the least number of running tasks. A reliable node can be determined through a *THRESHOLD* parameter that is compared against the node's reliability factor (Statement 1). A

Algorithm 5.4.2: RELIABILITYBASEDNODESELECTION($nodes$: $ListOfComputationalNodes$)

```

minTasksReliableNodes  $\leftarrow \infty$ 
minTasksUnreliableNodes  $\leftarrow \infty$ 
reliableNode  $\leftarrow \emptyset$ 
leastLoadedNode  $\leftarrow \emptyset$ 
for each  $n \in nodes$ 
  do {
     $nTasks \leftarrow n.noRunningTasks$ 
    if  $nTasks \leq minTasksReliableNodes$ 
      comment: Find least loaded reliable node
      if  $n.SRF \geq THRESHOLD$  (1)
        then {
           $minTasksReliableNodes \leftarrow nTasks$ 
           $reliableNode \leftarrow n$ 
        }
      comment: Find least loaded unreliable node
      else if  $nTasks \leq minTasksUnreliableNodes$ 
        then {
           $minTasksUnreliableNodes \leftarrow nTasks$ 
           $leastLoadedNodes \leftarrow n$ 
        }
  }
if  $reliableNode \neq \emptyset$ 
  then return ( $reliableNode$ )
  else return ( $leastLoadedNode$ ) (2)

```

node's reliability factor can have a value between 0 and 1. Thus, when $THRESHOLD > 1$ or $THRESHOLD = 0$, the algorithm considers all nodes and returns the least loaded one while ignoring the reliability criterion. The $THRESHOLD$ value should be determined according to the nature of the users' workflows and how sensitive their tasks are. If the reliability factors of all resources are under the specified $THRESHOLD$, the algorithm conveys the least loaded node (Statement 2) providing it with a chance to improve its reliability factor.

- **Data-intensive workflows:** A data-intensive workflow stages huge amount of data between its computational resources. In this case, the cost of data transfer dominates the cost of task executions. Hence, a node selection process based on data-awareness (Algorithm 5.4.3) will be more suitable for this type of workflows. The main target of the algorithm is to reduce the data transfer overhead. The $computerDataOverlap(t, r)$ in the algorithm calculates the number of bytes of input data of a task t that are available on a node r . Again judging about a node with large data overlap depends on a $THRESHOLD$ parameter that determines the minimum number of bytes of input data must

reside on a node in order to be considered in the selection process. Note that when $THRESHOLD = 0$, the algorithm selects the node with the maximum overlap regardless of the size of this overlap. The value of the $THRESHOLD$ parameter in this algorithm should be determined by users according to their network bandwidth. For a task with no inputs or with inputs dispersed in small chunks smaller than the specified $THRESHOLD$, the algorithm simply returns the least loaded node.

Algorithm 5.4.3: DATAAWARENODESELECTION($nodes$:
ListOfComputationalNodes, t : Task)

```

minRunningTasks  $\leftarrow \infty$ 
maxDataOverlap  $\leftarrow THRESHOLD$ 
overlapNode  $\leftarrow \emptyset$ 
leastLoadedNode  $\leftarrow \emptyset$ 
for each  $n \in nodes$ 
  {
    dataOverlap  $\leftarrow$  COMPUTEDATAOVERLAP( $t, n$ )
    nTasks  $\leftarrow$  n.noRunningTasks
    comment: Find least loaded node
    if  $nTasks \leq minRunningTasks$ 
      then {
        minRunningTasks  $\leftarrow$  nTasks
        leastLoadedNode  $\leftarrow$  n
      }
    comment: Find node with maximum data overlap
    if  $dataOverlap > maxDataOverlap$ 
      then {
        maxDataOverlap  $\leftarrow$  dataOverlap
        overlapNode  $\leftarrow$  n
      }
    comment: if data overlap on some node is larger than the specified
    comment: threshold, return the node with maximum overlap
    comment: otherwise, return the least loaded node
  }
if overlapNode  $\neq \emptyset$ 
  then return (overlapNode)
  else return (leastLoadedNode)

```

5.4.3 Workflow Steering

Some scientific projects are large scale and involve large teams of scientists and technicians. They may engage in experimental methods or procedures that take a long time to complete and require human intervention throughout the process.

Thus, *SWfMSs* should become more dynamic and open to interventions by users. This is also referred to as computational steering [Vetter 1996].

SWIMS meets this significant requirement through its *SR*. The *SR*'s public interface allows a client to "pause," "update," and "resume" a running workflow. The SWIMS workbench makes use of all these methods in order to provide the steering capability for its users as will be shown in Section 6.3.3.

5.5 Execution WS-Resource

The Execution WS-Resource (*ER*) is responsible for the actual execution of a submitted subworkflow over the deployed execution engine; in our case, we are using the Freefluo enactor [Freefluo 2009]. Thus, SWIMS concrete workflow specification language is XScufl [XScufl 2004]. Figure 5.8 shows the public interface for the *ER*. An *SR* instance can use the provided functionalities to create a new *ER* instance, submit it a subworkflow for execution and subscribe itself to the events published by the created *ER* instance. The *ER* publishes two types of events: subworkflow execution completed and failed events. Upon receiving an event from an *ER* instance, the *SR* sends an acknowledgement using the *ackOutputRecieved* method; otherwise, the *ER* instance will try to re-publish the same event again after a certain time period (configuration parameter). The *SR* can destroy an *ER* instance if it is not needed any more through the *shutdown* method.

ExecutionService	ExecutionServiceManagement
-_Subworkflow_Execution_Completed_Event : QName -_Subworkflow_Execution_Failed_Event : QName +execute(in workflowID : String, in workflowXML : String) : void +pauseExecution() : void +resumeExecution() : void +cancelExecution() : void +isPaused() : boolean +getOutputRefs() : String +getProvenanceReport() : String +getErrorMsg() : String +ackOutputRecieved() : void +getStatus() : String +shutdown() : void	+createExecutionResourceInstance() : EndpointReference

Figure 5.8: Execution WS-Resource Interface.

Figure 5.9 illustrates the subworkflow execution lifecycle by an *ER* instance in SWIMS. It is started by the retrieval of a subworkflow for execution, and then it follows five stages for the normal execution process:

1. **Check global cache:** The *ER* contacts the Workflow Catalog to check for a cached output for the input of the submitted subworkflow. If an output has been found, it jumps to the scheduler notification stage. Otherwise, it continues to the next stage.
2. **Input retrieval:** For every subworkflow input (see Listing 5.1), the *ER* follows the following steps to retrieve it:

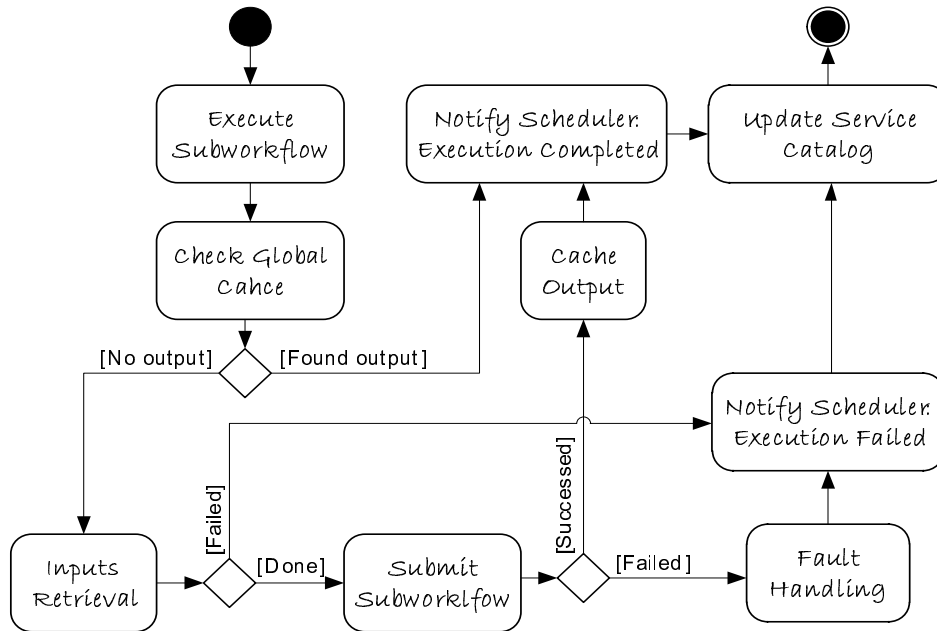


Figure 5.9: Subworkflow Execution Lifecycle [UML 2011]

- (a) The *ER* uses its local *DMR* to create a data sink in which the data will be retrieved.
 - (b) The *ER* submits the data reference, the data sink id, and a mediator reference if specified to the *DMR* located on the input's source node.
 - (c) The *DMR* contacts the Mediator Catalog and retrieves the indicated mediator that is represented as a workflow of OGSA-DAI activities [Leng 2009]. Then, it constructs an OGSA-DAI workflow (see Figure 5.3) in order to apply the required transformation and to send the transformed data back to the *ER* instance. We have decided to apply the transformation on the source node to ensure a *distributed transformation* in case of a node which requires several heterogeneous inputs from distinct remote nodes.
3. **Workflow execution and output caching:** After retrieving all input data, the *ER* submits the subworkflow for the workflow enactor for execution. In case of a successful execution, the *ER* exploits its local *DMR* to store the subworkflow's output and generate its corresponding OGSA-DAI references. Next, the *ER* contacts the Workflow Catalog to store the subworkflow's output in the global cache repository.
 4. **Scheduler notification:** The *ER* must notify the submitting scheduler instance by sending either an "execution completed" event containing the output references, or an "execution failed" event holding the failure cause.

The "execution failed" event may contain also information about subworkflow re-scheduling as we are going to show in Section 5.6.5.

5. **Update Service Catalog:** Last but not least, the *ER* informs the Service Catalog about the execution state in order to update the statistics of the underlying server. At the same time, in case of a failed execution due to a non-accessible concrete service, the *ER* sends the information of the failed service to the Service Catalog that changes the service's status to offline and alerts the service's owner about the failure.

Listing 5.1: Sample Input Segment.

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

In the following subsection, we are going to focus on provenance information collected during the execution process in SWIMS.

5.5.1 Provenance Information

Provenance represents the ancestry of an object; for instance, the provenance of a data product contains information about the process and datasets used to derive the data product [Braun 2008]. This type of provenance is essential for scientific workflows management system as it can help to reproduce as well as to interpret and validate scientific results.

There are two main categories of data provenance [Cruz 2009]:

1. **Prospective provenance:** captures the specification of the workflow procedure calls and data dependencies. It corresponds to the steps that need to be followed to generate a data product or class of data products.
2. **Retrospective provenance:** such as the recordings of when and where each procedure ran, and how each invocation behaved; it captures the steps that were executed as well as information about the execution environment used to derive a specific data product. In other words, it can be seen as a detailed log of the execution of a computational task.

SWIMS captures both data provenance types and stores them globally in the Workflow Catalog. Prospective provenance is stored for each workflow's execution snapshot while retrospective provenance is collected on the subworkflow-level

and can be accessed through the provenance report saved in the subworkflow's checkpoint. As we have shown before in Listing 4.2, the provenance report consists of three different sections. Since the subworkflow's execution is done locally by the *ER* instance located on the execution node, we are able to collect OS-level provenance (e.g., number of CPUs, memory size, etc.), which is provided in the report's first section. The second section gives details about the subworkflow's activities, including their inputs, outputs and configuration parameters whenever they exist. Finally, the third section records the execution time related information. Considering we are using the XML-based approach to store our provenance information, XPath/XQuery[XQuery 2010] technologies can be used to build queries over the data.

5.6 Optimization

Due to the separation of functionalities and the autonomic nature of the WMS's components, we could optimize them in different directions. The main goals of our optimizations are either to improve the performance of the WMS or to increase the overall reliability of the SWIMS environment. Most of our optimizations can't be applied on a traditional *WfMS* that employs a single scheduler or an execution engine. In the following subsections, we are going to discuss the optimizations we have applied on the different components of the SWIMS's WMS.

5.6.1 Code Movement - Node Management Resource

Normally, scientific workflows exchange a huge amount of data during their execution. Some available *SWfMS* uses a mediator-based approach for data transfer where data must be transferred first to a central repository and then to the target node which is completely inefficient. Accordingly, other *SWfMS* including SWIMS used a peer-to-peer approach where data can be transferred directly from the source node to the target node. Even this approach is not sufficient for scientific workflows as a single scientific service can produce a huge amount of data. Therefore, we have thought about "Code Movement" in which we try to move the "concrete service" rather than the data. This can be achieved for GRIA services in SWIMS as the *NMR* has full access and control of the GRIA server installed on its underlying Grid node. The main goal of this part of work³ was to figure out how to support transparent migration of Grid (GRIA) concrete services among different hosts. This yields to three abstract requirements:

- **Transportability:** A service needs to be able to relocate to different hosts.
- **Adaptability:** A service needs to be able to adapt to different hardware and software settings.
- **Transparency:** The migration of a service needs to be transparent to clients.

³I would like to thank Stefan Gasten for supporting me in this part of my work.

Our work in this part can be considered as a general idea which needs to be refined through a further deep research in order to form stable and consistent solutions of the existing problems. We can describe the problem scenario as follows: Consider host *A* owning the required input and host *B* owning the service know-how. In this case, *B* should act as a code server which prepares the service know-how and sends it to *A* where it can be deployed. Then, *A* executes the newly deployed service locally over the available input. This scenario adheres to the so called "code on demand paradigm" [Carzaniga 2007].

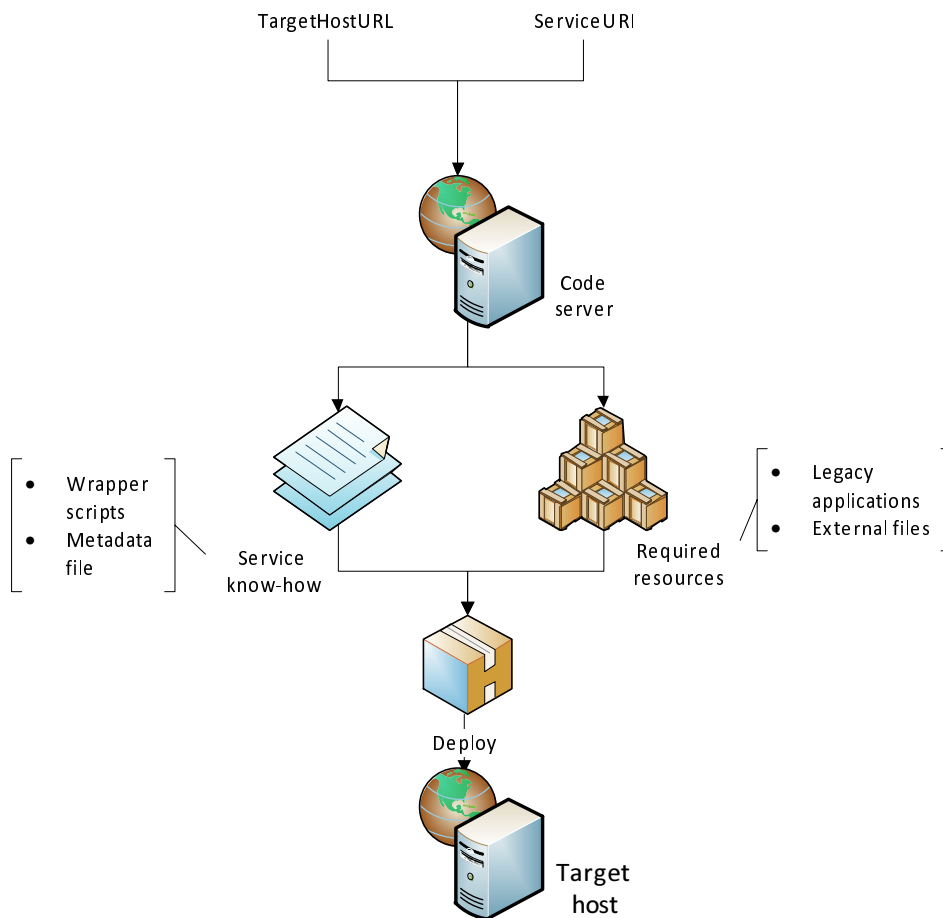


Figure 5.10: Grid (GRIA) Service Replication Workflow.

Figure 5.10 shows the workflow which can be used to replicate a given GRIA service on a target host. The workflow is initiated through the invocation of the code server's "*duplicateService*" method giving the URI of the service should be duplicated and the URL of the target host. Then, the code server packs the service know-how and all required resources for the service execution. For a GRIA service, the service know-how is the service wrapper scripts and meta-data file (see Section 2.1.1). The required resources refer to general host-specific dependencies

such as files, or legacy applications⁴. These required resources can be determined by analyzing the wrapper scripts to extract calls to legacy applications or accesses to external files. Finally, both the service know-how and the collected resources are zipped, and the target host's "*deployService*" method is invoked giving a data reference to the zipped file. At the target host, first, the zipped file is downloaded and unpacked. Then, the directory containing the legacy applications is added to the system's search path to allow the service's wrapper scripts to call them. Finally, the service know-how is deployed as a new GRIA job service.

Following the provided workflow, we conform to the transportability and transparency requirements as all steps should be done through the *NMR* instances on the code server and the target host. However, there is still a big question mark over the adaptability requirement. The main problem here is moving a host-dependent legacy application from one host to another, which is inapplicable between hosts with different architecture or operating system. Even in case of having the same architecture and operating system, an adaptable transfer of a legacy application can't be always achieved. For example, in Windows an application can have a tight coupling with the operating system through libraries and registry entries.

We have evaluated the capability of automatically creating a portable version of a Windows-based legacy application. Our idea was to determine all application's system libraries and registry dependencies, and to build a wrapper which should be able to run the application on the target host through the following steps: 1) install the system libraries, 2) back up import affected registry keys for later restoration, 3) import all registry dependencies, 4) execute the application, 5) and restore the registry. All of these tasks are typical for Windows installers. Thus, a wrapper can be created using the Nullsoft Scriptable Install System (NSIS) [NSIS 2011].

For determining application's system libraries dependencies, we have used a free-to-use software: Dependency Walker [Miller 2010]. Dependency Walker analyzes a given executable file for implicit, forwarded and delay-load module dependencies. In addition, Dependency Walker can detect run-time and system hook module dependencies.

In contrast to a library dependency analysis, the analysis of an application's registry keys dependencies generally requires the execution of the application. One reason for this is that an application might dynamically build registry calls at runtime. An evaluation of available registry monitoring software showed that most of them use registry snapshots to analyze registry keys modified by an application. Actual monitoring of registry calls requires hooking the Windows API in order to intercept calls to its registry functions [Microsoft 2011]. Among all evaluated software; this was a unique feature of Process Monitor [Rusinovich 2011].

The first problem with the analysis of registry dependencies process is that we need to make an automatic run of the application either using a sample input

⁴In terms of Grid services; legacy applications refer to platform dependent applications providing a service with its actual application logic

if available or without any inputs. In fact, successful execution on a target host can never be guaranteed with this approach if the application does not work on the same input as the one used on the source host while monitoring. Another non-trivial problem is determining the set of registry keys, which really belongs to the monitored application. A registry call monitor application delivers a complete set of registry values accessed by an application. Including all the retrieved keys in the wrapper can lead to serious problems as the retrieved set may contain critical keys with respect to the execution of other processes on the target host. A further issue regarding the set of registry values a wrapper contains is the concurrent execution of two different portable applications. If the intersection of the associated sets is not empty and the portable applications origin from different source hosts, concurrent execution will generally fail due to inconsistent registry values. A positive result, however, is that the analog does not necessarily hold for module dependencies. The reason is that an application's module search path typically begins in its working directory. Thus, the execution of such application does not require a system-wide installation of its module dependencies.

To conclude, the evaluated approach can be guaranteed just for applications with only system libraries dependencies. A daring approach to solve these issues could be to try to run the application within a virtual box on the target host. This virtual box should provide the minimal source host's requirements for running the application. However, we have to answer the question: How will the Grid service be able to communicate with the application within its virtual environment?

5.6.2 Tailoring the Service Behavior - Node Management Resource

Based on our previous work on Tailorability of BPEL-based workflows [El-Gayyar 2008, Alda 2007] that allow users to adapt BPEL-based workflow compositions at runtime, we thought about adding the Tailorability capability to SWIMS. As a matter of fact, through the Workflow Catalog, SWIMS users can access all stored workflows, adapt their compositions or re-execute them. However, as the *NMR* instances have control over their nodes' Grid servers, we thought about even going beyond that by allowing advanced SWIMS users to adapt the behavior of the available Grid services. For example, this can be achieved for GRIA services by simply adapting the service's script (see Section 2.1.1). Users can use the SWIMS's workbench to retrieve a service's script, adapt it and redeploy it as a new service (see Section 6.3.2). The validation, and the deployment of the modified script is done by the *NMR* instance of the original service's nodes.

5.6.3 Clustering - Scheduler Resource

The main partitioning criterion of the SWIMS scheduler is that every subworkflow should have only one remote task (Web/Grid services). This can lead to a large number of subworkflows for workflows with a large number of tasks. For instance, a typical Montage workflow (See Figure 2.3) accommodates hundreds of

tasks. Initialization and scheduling of a large number of these subworkflows can pose significant overhead. In order to reduce this overhead, several tasks can be aggregated into a single subworkflow to form a cluster.

Pegasus has already offered a similar mechanism to aggregate the tasks in a concrete workflow into clusters where every cluster is executed as a single job so that the remote resources can be utilized more efficiently [Singh 2008]. This approach reduces the number of subworkflows that reduces the load and accounting cost on the machine handling remote job submission. In Pegasus, there are two different clustering techniques: level- and label- based clustering.

In Pegasus’s level-based clustering, tasks at the same level can be clustered together according to either the number of clusters to be created per level (cluster factor) or the number of tasks to be grouped in a cluster (cluster size). SWIMS can provide, according to users’ needs, a similar technique while partitioning subworkflows to achieve a level-based (*horizontal*) clustering subject to a given cluster size. Figure 5.11 shows the Montage workflow in Figure 2.3 horizontally clustered with cluster size = 2.

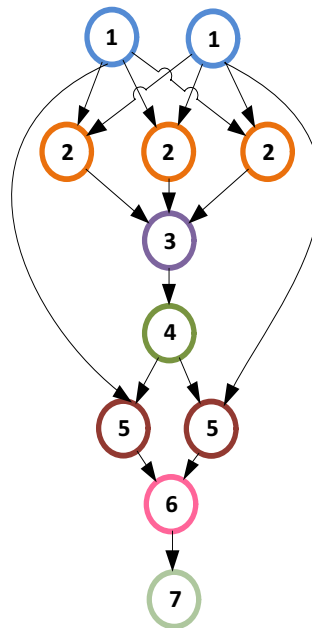


Figure 5.11: Level-based Clustered Montage Workflow with Cluster Size = 2.

In Pegasus’s label-based clustering, the user can label the tasks in the workflow to be clustered together; the tasks in the workflow with the same label are grouped into a single cluster. This type of clustering is not preferable in SWIMS as it requires user interaction while SWIMS tries to hide its users from any technical issues. However, we have implemented another type of clustering: *vertical* clustering, that combines dependent tasks from different levels while holding two conditions:

1. It should maintain the provided cluster size.

- Each constructed cluster must satisfy the convexity requirement that states that all paths between any two tasks in a cluster must be completely contained within it to avoid co-scheduling between clusters.

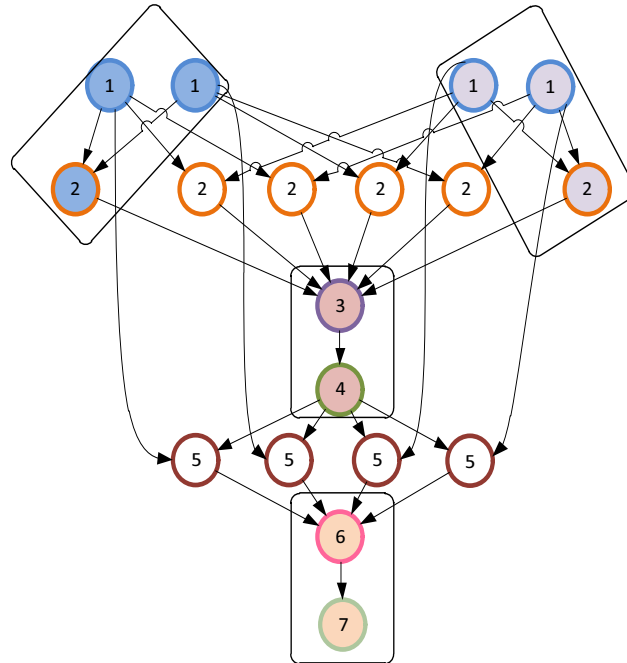


Figure 5.12: Vertically Clustered Montage Workflow with Cluster Size = 3.

Vertical clustering can lead to a workflow planning problem as the planner has to assign each cluster to a computational node where all tasks within the cluster have been deployed on it. Nevertheless, this condition can be prevailed in SWIMS if we are sure that the remote services within our workflows can be duplicated using the *Code Movement* technique introduced in Section 5.6.1 as the case for Montage workflows. In this case, a workflow planner can simply assign a cluster to a node according to the subworkflow’s topmost task while any missing services will be duplicated during the execution process.

Figure 5.12 illustrates the Montage workflow in Figure 2.3 vertically clustered with cluster size = 3. This type of clustering can be achieved by replacing the *populateSubworkflow* procedure in Algorithm 5.4.1 by the one provided in Algorithm 5.6.1. For sure we have to omit the condition of having one remote task by subworkflow and replace it by a condition that maintains the given cluster size (Statement (1)). Moreover, we should consider parent nodes that can’t be reached by simply following the path downwards from the starting node. This can be achieved by checking the parents of each child and check, whether these parents can be added to the current model. To keep the algorithm simple, we consider only the child’s topmost parents (parents without parents). The child and its parents will be added to the model if and only if:

Algorithm 5.6.1: POPULATESUBWORKFLOWVCLUSTERING(S : $subworkflow, t$: $task$)

```

if  $t$  is assigned
  then return
if  $numberOfTasks(S) = ClusterSize$ 
  then { return (1)
   $S \leftarrow S \cup t$ 
  Mark  $t$  as assigned
  for each  $c \in children(t)$ 
    do {
       $parentsList \leftarrow \emptyset$ 
       $addParentsFlag \leftarrow true$ 
      for each  $p \in parents(c)$ 
        do {
          if  $p = t$ 
            then continue
          if  $p$  is not assigned and  $parents(p) = \emptyset$ 
            then {  $parentsList \leftarrow parentsList \cup p$  (2)
            else {  $addParentsFlag \leftarrow false$ 
               $break$ 
            }
          if  $addParentsFlag$ 
            then if  $(size(parentsList) + 1) \leq ClusterSize$  (3)
              for each  $p \in parentsList$ 
                do {  $S \leftarrow S \cup p$ 
                  Mark  $p$  as assigned
                }
              POPULATESUBWORKFLOWVCLUSTERING( $S, c$ )
        }
    }
  
```

- The child has no parents that have been assigned to another subworkflow (to avoid co-scheduling) as stated in Statement (2).
- Adding the child and its parents to the current model will not violate the cluster size condition (Statement (3)).

The remaining tasks at each level can be executed individually or further clustered using level-based clustering. Vertical clustering can reduce the degree of parallelism; however, it is very useful for workflows with short tasks that exchange huge amount of data. In this type of workflows, avoiding data transfer between tasks within subworkflows can be much more advantageous than preserving parallel execution.

Deciding on the right size of clusters is still a challenging problem. If clusters are too small, the benefits of clustering are limited as the number of jobs that need to be managed is not significantly reduced. If clusters are too large, then the workflow can be more vulnerable to failures. If a failure occurs within a cluster, then the entire cluster needs to be re-computed.

5.6.4 Global Data Caching - Execution Resource

Global data caching is a significant functionality of the *ER*. It is a very important feature for scientific workflows that are based on long running processes. It helps to achieve *smart re-run* since scientists generally tend to re-run scientific experiments while changing only the inputs or configuration parameters of few tasks. In this case, only those updated tasks will be actually re-executed. SWIMS users are able to force the system to re-execute a service even if exists a cached output for its given input.

Data caching in SWIMS is accomplished through creating an SHA-2 fingerprint [SHA-2 Standard 2002] from the subworkflow's inputs and configuration parameters to work as a hash value to the produced output. A detailed description of how an SHA-2 fingerprint is created for a subworkflow is given in Algorithm 5.6.2. To avoid unnecessary movement of a large amount of data, SWIMS takes advantage of the remote *DMR* instances located on each input's node to compute locally the input's SHA-2 digest. The inputs' computed digests are concatenated with the configuration parameters of all subworkflow's activities into a single string. The SHA-2 digest of this string forms the final fingerprint used either to cache the subworkflow's output or to retrieve the cached output from the global cache. In scientific workflows, the size of a single input can be very large. Therefore, we decided to use only around a kilobyte snapshot of each large input to compute its digest. This snapshot is constructed by taking three hundred fifty bytes from the beginning, the middle, and the end of the input's content.

After computing the hash value of the subworkflow's inputs, the *ER* instance submits the hash value and the subworkflow's output references to the Workflow Catalog where they can be stored globally and become accessible to all other *ER* instances.

5.6.5 Distributed Fault Handling - Execution Resource

Each *ER* instance is responsible for handling exceptions occur during the execution of submitted subworkflows on its underlying node. It is also responsible for killing or rescheduling running/planned subworkflows if the node is heavily loaded. A heavily loaded node can be determined according to a set of configuration parameters specified by the node administrator. These configuration parameters identify the upper limit of node resources (e.g., memory size, disk space, etc.), which can be consumed by the SWIMS framework. In other words, the *ER* affords *distributed fault handling and load balancing* mechanisms over the subworkflow level. The *ER* uses two different techniques for handling faults:

1. **The retry technique:** This technique is the simplest failure recovery method, as it simply tries to re-execute the failed task on the same node after some delay. This technique is used only if the subworkflow's required concrete service is not broken (accessible).

2. **The re-scheduling technique:** Here, the *ER* submits the failed task to another computational node. If an *SR* is already deployed in the *ER*'s node, the *ER* will use it to find the required node and transfer the information about the new node to the *main scheduler* within its "subworkflow failed event". Otherwise, the rescheduling process is propagated to the workflow's *main scheduler*

Often scientific workflow systems simply rely on the fault tolerance capabilities provided by their third party invoked services. When failures occur during the execution of these services, a workflow system typically sees them only as failed steps in the process without additional details about the failure causes which can increase the ability of the system to recover from those failures. However, in SWIMS, the execution of the third party services is done by the local *ER* instance that has access to low-level error logs produced by the invoked concrete service. This helps to provide low-level error reports about occurred failures at the OS-level (e.g. missing modules/libraries, incorrect input/outputs, failed authentication, out of disk space, out of memory, etc.). As an example, Listing 5.2 shows snapshots of two error logs collected from GRIA jobs. The first snapshot shows a missing application "unzip" while the second shows a failure in the underlying legacy application.

Listing 5.2: Low Level Error Logs.

```
[Snapshot 1]
*** Failed to run comand '['unzip', 'inputs/inputFITsImage']':
*** The system cannot find the command specified
*** Hint: check that unzip is in your PATH

[Snapshot 2]
Archive:  inputs/inputCorrectedImages-0
  inflating: unzip/c2mass-atlas -971024n-j0080033.fits
  inflating: unzip/c2mass-atlas -971024n-j0080033.area.fits
Archive:  inputs/inputCorrectedImages-1
  inflating: unzip/c2mass-atlas -981123n-j0720033.fits
  inflating: unzip/c2mass-atlas -981123n-j0720033.area.fits

*** Exception: Command ['mImgTbl', 'unzip', 'correctedImages.tbl']
*** [struct stat="ERROR", msg="Can't open tmp (in) table."]
```

5.7 Fault Handling in SWIMS

Before ending up this chapter, we would like to provide a global view of fault handling in SWIMS. In our environment, workflows are executed in a distributed manner through several execution engines. During the execution of a workflow, execution failures may be caused by many reasons, such as failure/shutdown of

nodes, network failures, Grid services failure, etc. To handle failures flexibly and to support reliable execution, SWIMS provides fault handling techniques on three different levels as presented in Figure 5.13.

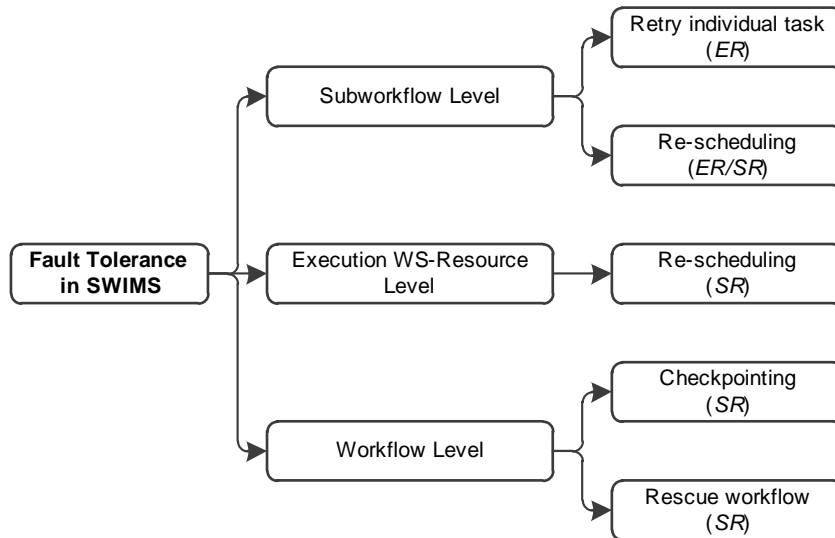


Figure 5.13: Fault Tolerance in SWIMS.

As discussed in Section 5.6.5, the *ER* employs either the retry or the rescheduling technique to recover from errors on the **subworkflow level**.

The *SR* exploits the rescheduling technique to handle errors on the **ER level**. To detect such a type of errors the *SR* runs a background watchdog program in fixed periods. This program is responsible for checking the status of all currently running subworkflows. Whenever a certain *ER* instance is unreachable, all its mapped subworkflows are rescheduled to another available *ER*. Meanwhile, the *SR* notifies the Service Catalog about the failed node in order to update its servers table.

On the **workflow level**, SWIMS provides two fault handling techniques through its *SR*: light-weight checkpointing⁵ and workflow rescue. The *SR* captures a snapshot of the running workflow after the execution of each subworkflow and sends it to the Workflow Catalog. This can help clients to backtrack (in the case of an input/parameter change or even an execution failure) to a previously saved state without starting over from scratch. Even more, having these checkpoints onto a global Catalog supports the distributed management of workflows, since when a workflow *main scheduler* itself fails; another *SR* instance can use this information to continue the coordination of the broken workflow starting from the last captured checkpoint. To achieve the most benefit from the checkpointing facility, SWIMS utilizes the rescue workflow technique that ignores the failed tasks and continues to execute the remainder of the workflow until no more forward progress can be made.

⁵Data items are represented through their references

Summary

In this chapter, we have explained the different elements of the SWIMS server-side components: The WMS instances deployed on Grid nodes wishing to participate in the workflow management and execution process. SWIMS's WMS can be realized as a deployment of a bundle of WSDM-based services, which controls four manageable resources: data management (*DMR*), node management (*NMR*), scheduler (*SR*), and execution (*ER*).

The *DMR* is dedicated to reference-based data movement, and automatic data transformation between heterogeneous services. The *NMR* helps the SWIMS environment to transparently keep track of all available concrete services. As the *NMR* has full control over the underlying node and its Grid server, it can help also to collect information about the node, move services from one node to another, deploy workflows as services or to tailor the behavior of an installed service.

For every submitted workflow, an *SR* instance should be elected to act as the main coordinator of the workflow's execution. This instance breaks the submitted workflow into a set of subworkflows, and submits each subworkflow for execution over a remote Grid node through its deployed *ER* instance.

Every *ER* instance is completely responsible for the execution of submitted subworkflows. In case of a failure, the *ER* instance can retry the task on its local node or use its node's local *SR* instance to find an alternative Grid node to which it can move the failed subworkflow. Moreover, as the *ER* instance is located at the concrete service's node, it can generate low-level error reports that can include errors occurred at the OS-level. This can help servers' administrators to identify and recover the failure causes.

The workflow's main coordinator also ensures that all active *ER* instances are working smoothly; otherwise, it re-submit the subworkflows of any failed *ER* instance to another running one. Besides, it stores an execution checkpoint in the Workflow Catalog after the execution of every subworkflow.

If workflow's main scheduler has failed, another *SR* instance will be elected to continue the execution of the workflow starting from the last captured checkpoint. To sum up, several *SR* and *ER* instances cooperate to ensure reliable and extendible workflow management and execution in the SWIMS environment.

Algorithm 5.6.2: CHECKGLOABLCACHE(S : *subworkflow*)

comment: Check SWIMS global cache for an output of the given
comment: subworkflow input

procedure CHECKGLOBALCACHE(S : *subworkflow*)
comment: Use remote DMR instances to compute the SHA-2 Digest
comment: of each individual input

$cacheSupportList \leftarrow \emptyset$
for each $inputRef \in S$
 do $\begin{cases} inputDigest \leftarrow \text{DMR.COMPUTEINPUTSHA2DIGEST}(inputRef) \\ cacheSupportList \leftarrow cacheSupportList \cup mediatorRef \\ cacheSupportList \leftarrow cacheSupportList \cup inputDigest \end{cases}$
comment: Besides the input, we consider also the configuration
comment: parameters for each task

for each $task \in S$
 do $\begin{cases} configParameters \leftarrow \text{GETCONFIGURATIONPARAMETERS}(task) \\ cacheSupportList \leftarrow cacheSupportList \cup configParameters \end{cases}$
comment: Sort the list to avoid disorder

$\text{SORT}(cacheSupportList)$
comment: Concatenate collected data in one string, the SHA-2 digest
comment: of this string represents the subworkflow's input fingerprint

$strBuffer \leftarrow \emptyset$
for each $s \in cacheSupportList$
 do $\text{APPEND}(strBuffer, s)$
 $inputDigest \leftarrow \text{COMPUTESHA2DIGEST}(strBuffer)$
 $cachedOutput \leftarrow \text{WFCATALOG.GETCACHEDOUTPUT}(inputDigest)$
return ($cachedOutput$)

comment: SHA-2 digest computation: for big contents we consider
comment: only snapshots to avoid heavily computational process

procedure COMPUTESHA2DIGEST($content$)
 $SegmentLength \leftarrow 350$
 $MaximumInputForDigestLength \leftarrow SegmentLength * 3$
if $length(content) \leq MaximumInputForDigestLength$
 then return ($\text{SHA2DIGEST}(content)$)
 else $\begin{cases} strBuffer \leftarrow \emptyset \\ append(strBuffer, initialSegement(content, SegmentLength)) \\ append(strBuffer, middleSegement(content, SegmentLength)) \\ append(strBuffer, endSegement(content, SegmentLength)) \\ return (\text{SHA2DIGEST}(stringBuffer)) \end{cases}$

SWIMS Workbench

Contents

6.1	SWIMS Workbench Architecture	75
6.2	SWIMS Abstract Workflow Language	77
6.3	SWIMS Workbench User Interface	79
6.3.1	The Editing / Composition Mode	80
6.3.2	Exploring the Workflow Catalog	83
6.3.3	The Monitoring Mode	85
6.4	Workflow Validation	86
6.5	SWIMS: The Overall Picture	89

In the last chapter, we considered the SWIMS’s server-side components. Here, we finalize our discussion about SWIMS by considering the SWIMS’s client-side component: the SWIMS workbench.

The main target of the SWIMS workbench is to provide a simple environment with a high level of abstraction through which scientists can compose, execute, monitor, steer, re-use, and re-run scientific workflows without considering the complex underlying Cyberinfrastructure used to perform these tasks.

We start with illustrating the overall architecture of the workbench. We will then discuss the SWIMS’s abstract workflow language. Next, we will give a deeper look at the SWIMS workbench’s user interface highlighting its main features, that help users to easily edit, monitor, and validate workflows. Finally, we draw an overall picture of SWIMS and show how all its different components cooperate with each other in order to provide a reliable and distributed management and execution of scientific workflows.

6.1 SWIMS Workbench Architecture

Figure 6.1 introduces the SWIMS workbench architecture; it is based on the Eclipse Rich Client Platform (RCP) framework [RCP 2010, McAffer 2005] built over the equinox OSGi runtime environment [Equinox 2011]. RCP is a well-suited platform for most Java-based stand-alone applications. Compared with many other corresponding platforms, RCP has a number of advantages: First of all, applications developed on top of RCP are completely portable and will run equally well on different available execution platforms (e.g., Windows, Mac or Linux). In addition,

RCP provides a set of basic functionality that is needed by many client applications: a sophisticated help system, a well thought-out look and feel, and an efficient service oriented framework. Last but not least, RCP provides the concept of "extension points" that facilitates loosely-coupled extensions to functionality.

Besides the well known Eclipse's extension points, SWIMS provides its own extension points to help developers to add domain specific functionalities to the workbench. One example is the extension points provided by the UI plug-in which help to add new editor commands, palette entries, or palette categories. Another example is the extension point provided by the serialization plug-in for adding additional language converters.

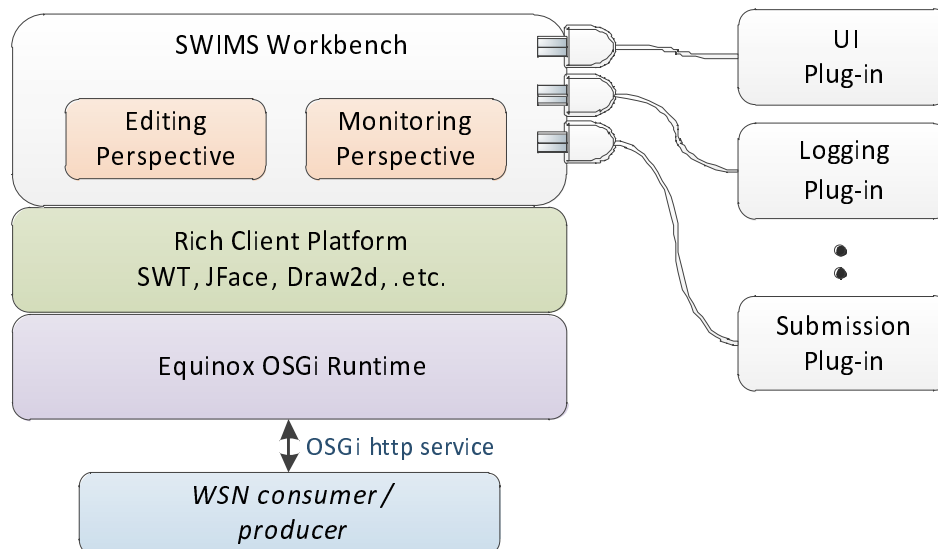


Figure 6.1: SWIMS's Workbench Architecture

As all SWIMS's components communicate through exchanging WSN notifications, we have implemented a Web service in the background of the workbench capable of consuming and producing such notifications. The service is built over the OSGi [OSGi 2011] http service provided by the Equinox framework.

SWIMS workbench consists mainly of two main modes, the editing and the monitoring modes. The editing mode enables scientists to compose and edit a scientific workflow in a graphical way from a set of abstract activities. Moreover, within the editing mode, scientists can explore and utilize workflows stored in the Workflow Catalog and their attached information. Whenever submitting a workflow for execution, the monitoring mode is activated in order to allow users to monitor and steer the submitted workflow. Sections 6.3.1 and 6.3.3 provide more detailed information about the two modes.

6.2 SWIMS Abstract Workflow Language

SWIMS uses an XML-based language to represent abstract workflows constructed through the SWIMS workbench. We try to keep our language as simple and generic as possible to support human readability and to allow users (scientists) to describe their scientific workflows at a high level of abstraction that shields them from the complications of the underlying system components. In this way, scientists can focus on the design of their scientific experiments without being dispersed with the complexity of any specific technology (e.g., Web or Grid services). Keeping the language simple also strengthens the capability for automatic conversion between SWIMS's language and other abstract/concrete languages used by specific workflow enactors. In our case, we are using the XSLT technology to convert SWIMS's language into XScufl, the language supported by Freefluo: the SWIMS's workflow enactor.

Figure 6.2 shows a simplified representation of the XML schema tree of the SWIMS language. SWIMS workflows consist of workflow inputs, workflow outputs, activities, data transitions and control transitions.

The "WorkflowInputs" and "WorkflowOutputs" elements are used to declare top level workflow inputs and outputs. In case of a workflow input/output of type file or list of files, an optional "MimeType" element can be used to classify the file(s) type. The "isZip" attribute is also valuable only in case of inputs/outputs of type file. It is used internally by the SWIMS workbench to decide whether the provided file(s) need to be automatically zipped before being submitted to the execution environment. Some Grid services require/produce a zipped version of their inputs/outputs either as a way of compression or to preserve the original input/output files' names (see Appendix A).

A SWIMS activity is represented by activity resource requirement and description sections. The resource requirements section allows users to explicitly define the set of minimal requirements of the task's execution node, which is composed of static information (e.g., OS type, architecture and number of CPUs), and dynamic information (e.g., available memory and disk space). The activity description section holds the service's abstract template generated by the Service Catalog (See Section 4.4). It consists of a set of inputs and outputs ports that provide logical representations of the corresponding input and output data. It also includes a set of parameters that helps to attune the behavior of the underlying activity. The activity "secured" attribute determines whether the activity is accessible or not for the current user as we are going to show later in the next Section. An example of an activity segment is shown in Listing 6.1.

SWIMS activities are connected by data and control transitions. Conceptually, a data transition transfers data from an output of one activity to the input of another activity. The absence of an output/input attribute value of the source/target element infers that the link is referring to a top-level workflow input/output respectively.

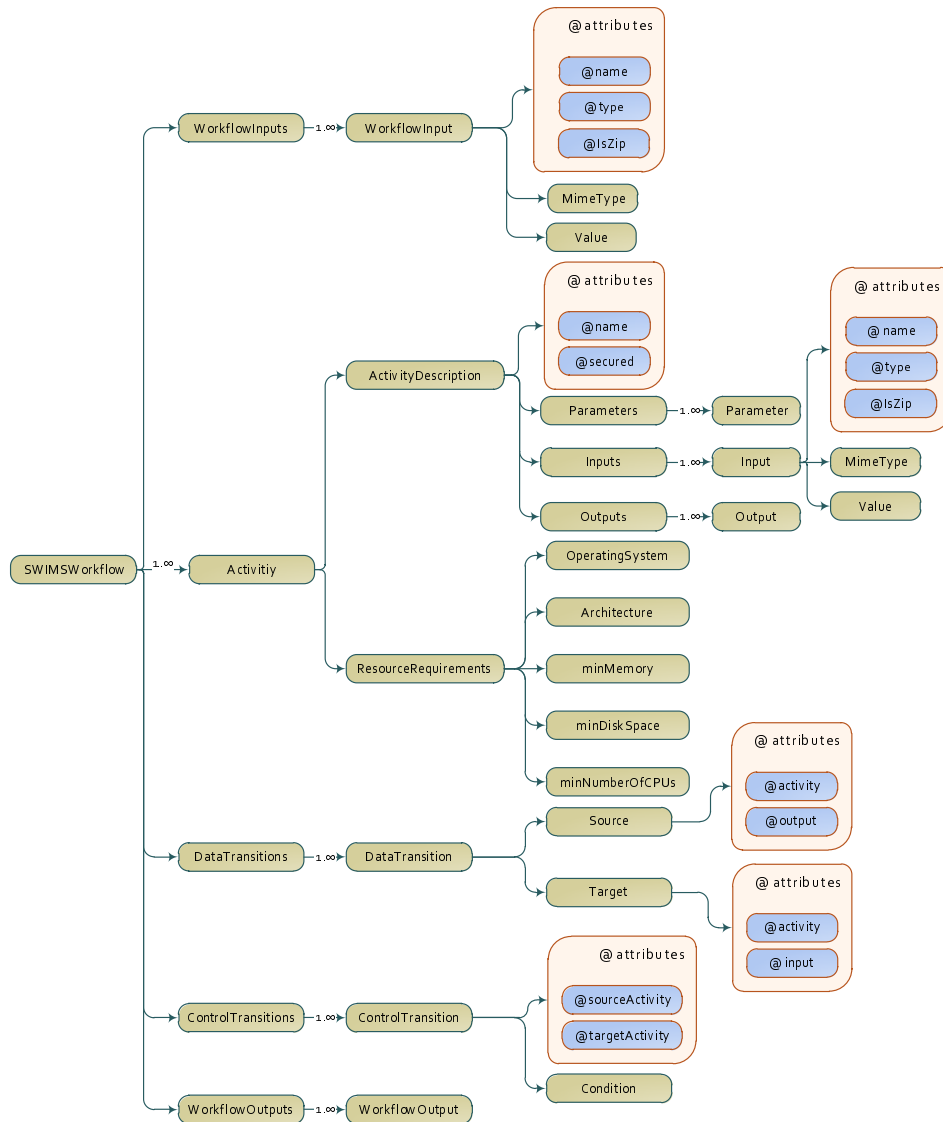


Figure 6.2: SWIMS's Language XML Schema Tree

Control transitions are used to prevent the transition from the source activity to the target activity until some constraint condition has been satisfied. Currently, conditions are limited to simple expression over the outputs of the source activity. The default constraint condition is blocking the transition to the target activity until the source activity has completed. To keep the language simple, we do not support explicit subworkflow construct. Subworkflows is implicitly supported as SWIMS provides the capability to deploy workflows as Grid services, that can be then used as atomic activities in more complex workflows.

The drawback of the SWIMS's language is being limited to DAG workflows as it does not support iterations constructs (e.g., do while, for loops). It should be

Listing 6.1: Sample of an Activity Segment

```

<Activity name="mJPEG1">
  <ActivityDescription name="mJPEG" secured="false">
    <Description>Generates a JPEG image from FITS files</Description>
    <Parameters>
      <Parameter name="isColorful" type="boolean">
        <Description>image in grayscale or true colors</Description>
        <value>true</value>
      </Parameter>
    </Parameters>
    <Inputs>
      <Input name="inputMosaicImage" type="list" zip="true">
        <Description>FITS images need to be converted</Description>
        <MimeType>Image</MimeType>
        <Value>...</Value>
      </Input>
    </Inputs>
    <Outputs>
      <Output name="outputMosaicJpegImage" type="file" zip="true">
        <Description>The output JPG image</Description>
        <MimeType>Image</MimeType>
      </Output>
    </Outputs>
  </ActivityDescription>
  <ResourceRequirments>
    <operatingSystem>Linux</operatingSystem>
    <architecture>32 Bits</architecture>
    <minMemory>0.0</minMemory>
    <minDiskSpace>100.0</minDiskSpace>
    <minNumberOfCPUs>0</minNumberOfCPUs>
  </ResourceRequirments>
</Activity>

```

part of the future work to support such a type of constructs while trying to keep the language as simple as possible.

6.3 SWIMS Workbench User Interface

The SWIMS workbench enables users to graphically create and edit descriptions of abstract workflows to be executed over the Grid. A workflow is a set of activities (abstract services), interconnected by transitions that define the order in which the activities must be performed. Every activity corresponds to a computer program which may be encapsulated by a Grid service (concrete service) and is located on one of the available nodes in the Grid. Once a workflow has been created, the workbench can be used to submit it for the execution over the available Grid. The remote execution of a workflow can be monitored or steered, and output files can be downloaded to the user's computer. Apart from these basic features, the SWIMS workbench offers a bunch of additional functions like browsing and utilizing the

contents of the Workflow Catalog, workflow validation, deployment of workflows as Grid services, tailoring of the service behavior and transferring files transparently to Grid nodes.

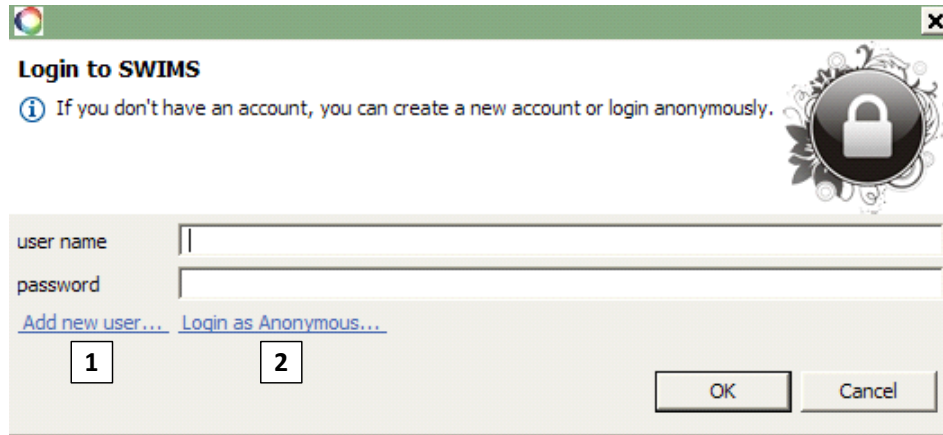


Figure 6.3: SWIMS's Workbench Login Screen.

When the workbench is started, it displays its login screen (Figure 6.3) that allows users either to login to the workbench using their user names and passwords, create a new user [1] or login as an anonymous [2]. The provided user information is sent to the Service Catalog in order to retrieve the list of available activities and their accessibility status for the current user. The obtained list of activities' abstract templates (see Section 4.4) is used to initialize the activities section in the workbench palette (see Section 6.3.1) that can be used by the user to construct abstract workflows. In case of an anonymous login, only templates of services declared as public will be returned by the Service Catalog. SWIMS workbench has two different modes, the editing and the monitoring modes. The features and functionalities provided in each mode are discussed in the following sections.

6.3.1 The Editing / Composition Mode

The SWIMS workbench editing mode (see Figure 6.4) provides a graphical editing tool for abstract workflows, offering features like undoing changes, performing automatic graph layouts, zooming, and printing of diagrams. Each workflow is created in its own project in the Workflows Workspace [1] which is a directory, usually located on the local hard drive. When creating a new workflow project or opening an existing workflow file, a new workflow editor instance [2] is opened for building the workflow description.

Workflow descriptions are graphs consisting of nodes (workflow inputs, workflow outputs, and activities) and edges (data and control transitions). New elements can be added to the workflow through the provided palette [3]. Activities retrieved from the Service Catalog are organized into two groups (public and secured) according

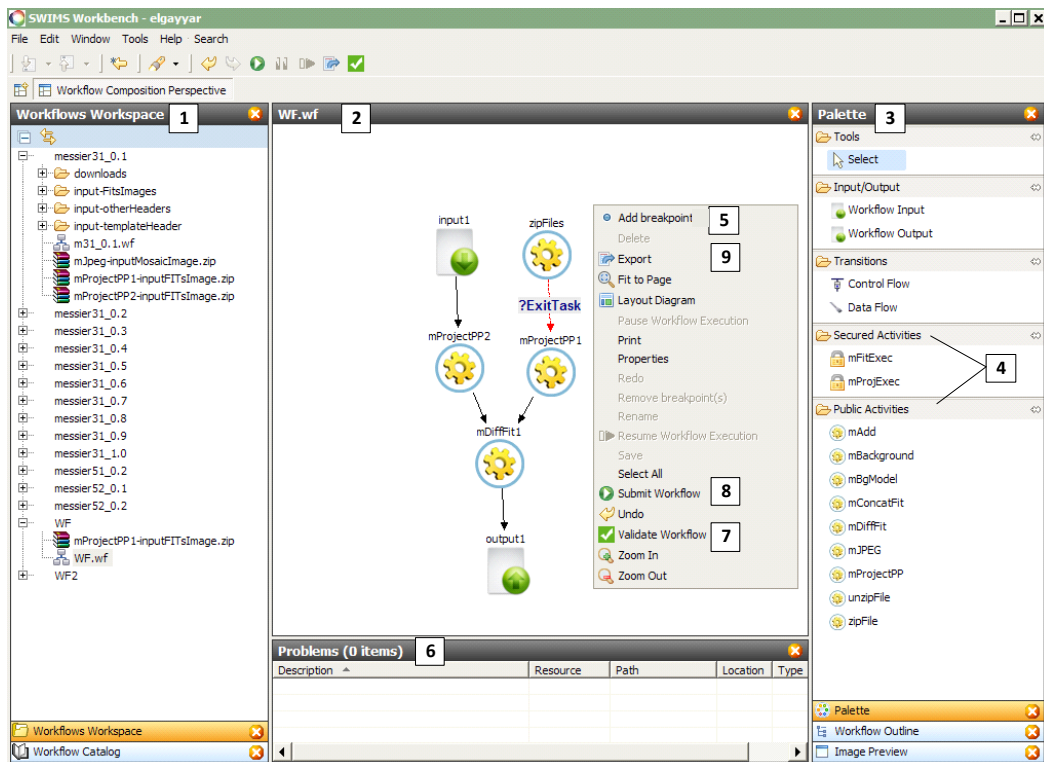


Figure 6.4: SWIMS's Workbench Editing Mode.

to their accessibility status for the current user [4]. As shown in the Figure, the workbench provides a uniform representation of activities regardless of their types (local, web services, Grid services .etc). This helps to hide the underlying system complications from SWIMS's users. Users can use either the secured or the public activities to construct an abstract workflow; *however, they will not be able to submit a workflow with secured activities for execution*. In order to get access to a secured activity, the user has to send a request through the workbench to the Service Catalog which forwards this request to the owners of concrete services that match this activity. Then, owners have to decide whether this user can access their services or not. Whenever the user connects two activities with a data dependency, the workbench contacts the Mediator Catalog to check for a mediator between these two activities; if a mediator has been found, its reference will be added to the workflow's specifications. Breakpoints [5] can be added to the workflow to pause the execution of the workflow before submitting the breakpoint's activity. Users can also validate [7] their workflows against different violations (see Section 6.4). All identified problems will be listed in the problems view [6]. After handling all discovered errors, the workflow can be submitted [8] for execution, which switches the workbench's mode to the monitoring mode. The workflow can also be exported [9] to the SWIMS's language or any other supported languages (in or case XScufl).

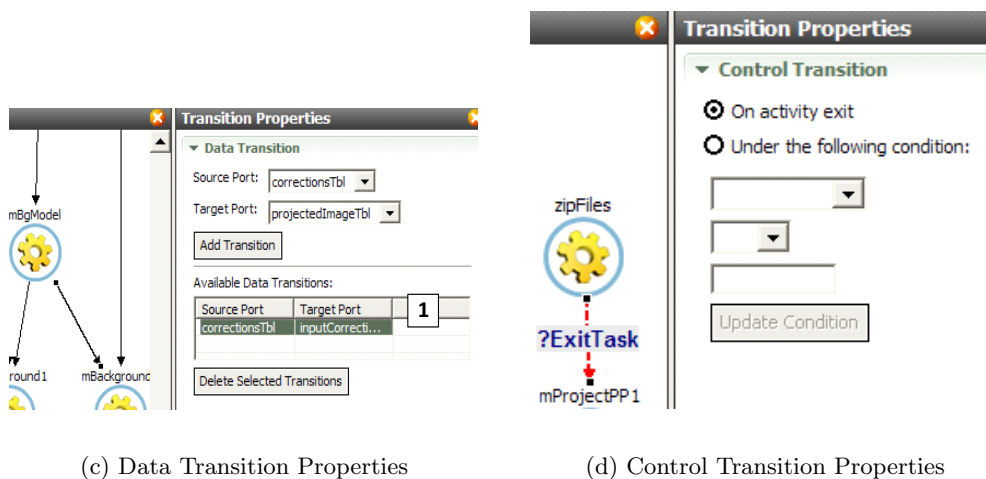
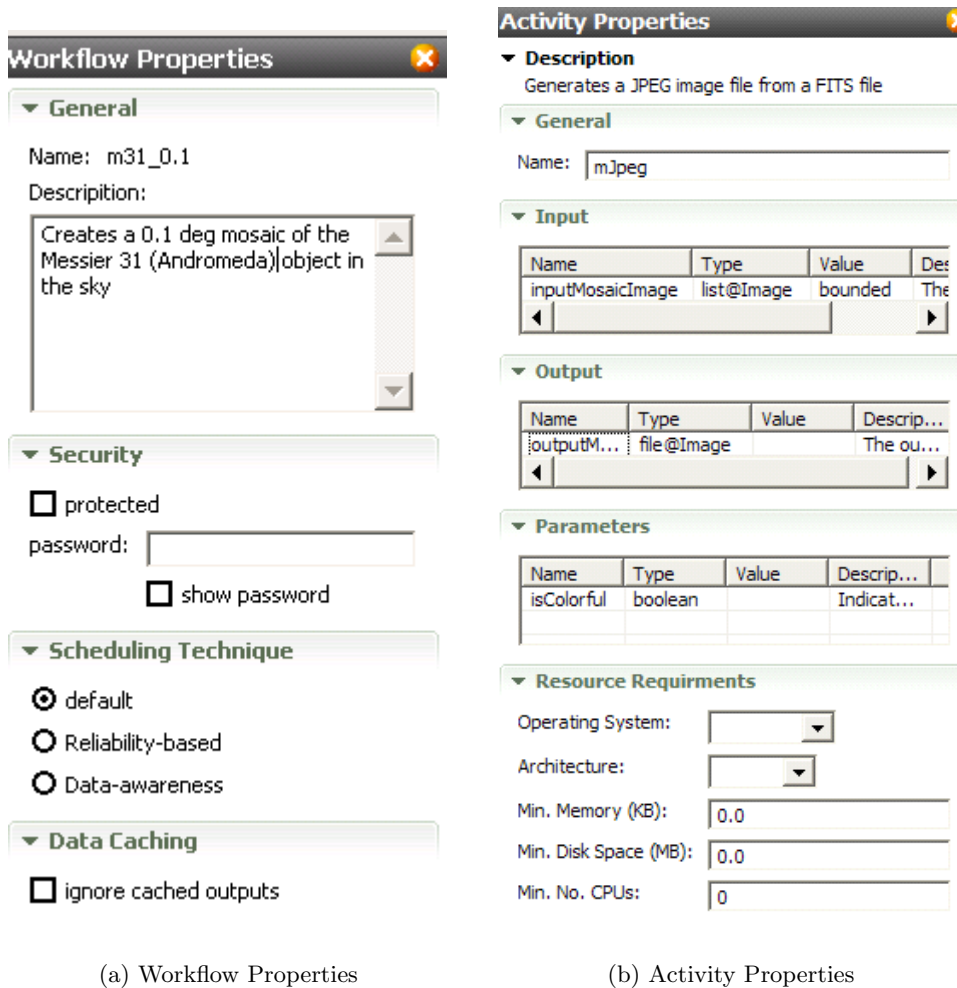


Figure 6.5: SWIMS's Workbench Properties Views.

Users can use the workbench's different properties views (See Figure 6.5) to adjust the properties of the workflow's elements:

- **The workflow properties view** (Figure 6.5(a)): This view is divided into three sections. In the general section, the user can add a detailed description of the workflow's functionality. Utilizing the security section, users can specify a password to secure the access of the workflow's information which will be stored in the Workflow Catalog. In addition, through this view, users can disable data caching and select a specific scheduling technique for the next run. In this way, users can execute the same workflow with different configurations to determine the best one of them.
- **The activity properties view** (Figure 6.5(b)): It enables users to adjust an activity's inputs, outputs, parameters and resource requirements. To avoid ambiguity, we allow users to define resource requirements for all activities while they are meaningless for local activities. Thus, resource requirements for local activities are ignored by the *SR* while planning the workflow.
- **The data transition properties view** (Figure 6.5(c)): This view is used to specify data transitions between two activities. To keep the diagram simple, only one link is used to represent a data transition between any two activities, while this link can hold many data transitions between outputs of the source activity to the inputs of the target activity. These transitions are added to and can be deleted from the available data transitions table [1].
- **The control transition properties view** (Figure 6.5(d)): Users can use this view to formalize the condition of a control transition.

6.3.2 Exploring the Workflow Catalog

Another important view that can be accessed in the editing mode is the Workflow Catalog view (Figure 6.6) [1]. It is used to outline the Workflow Catalog's contents as a tree structure representing execution snapshots captured for different workflows executed over the underlying Cyberinfrastructure.

Through this view, users can explore the details of checkpoints recorded during the execution of different execution snapshots. By selecting a certain checkpoint, the workbench updates its associated workflow to show which activities have already been executed until the selected checkpoint [2]. Users also can check the provenance report to get more insight information about the execution process (see Section 5.5.1). There are other important actions provided by this view:

- **Workflow re-run:** Users may be interested in a certain workflow model and would like to retest the model after applying some changes to the inputs or parameters of some activities. This can be achieved through the "Submit Checkpoint" command [3]. This command will execute the required workflow

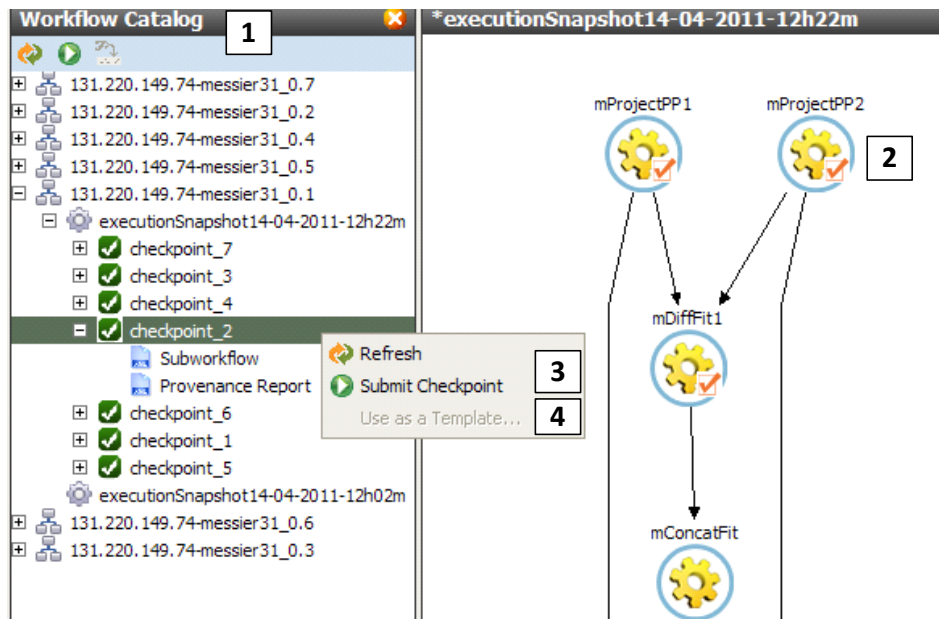


Figure 6.6: SWIMS's Workbench Workflow Catalog View.

starting from the selected checkpoint while considering the changes that have been applied over all non-executed activities.

- **Workflow sharing:** Users may be also concerned with doing some modification over the model itself by adding or deleting different workflow elements. This scenario is supported by the "Use as a Template" command [4] that creates a new project within the user's local workspace and copies the required workflow into it making it ready for further modifications.

By providing these functionalities, the Workflow Catalog enriches SWIMS with a tailoring capability that allows users to adapt and share existing workflow models. This is similar to our previous work published in [El-Gayyar 2008, Alda 2007] about tailorability of BPEL-based business workflows. However, as we have mentioned in Section 5.6.2, we can go further with the help of the *NMR* instances that have control over their nodes' Grid servers. This helps us to allow users with granted permissions from services' owners to retrieve a GRIA service's script (application wrapper script, see Section 2.1.1) and adapt it in order to change the service's behavior (see Figure 6.7). To keep the original service and to avoid breaking down any workflows that make use of this service, users are enforced to deploy any adapted service only as a new one. They are allowed also to change the interface of the new service by adapting the service's description (meta-data, see Section 2.1.1) file. Whenever the user asks for a deployment of an updated service, the workbench contacts the Service Catalog to retrieve the list of nodes that hosts the original service. Then, it submits a deployment request to their *NMR* instances.

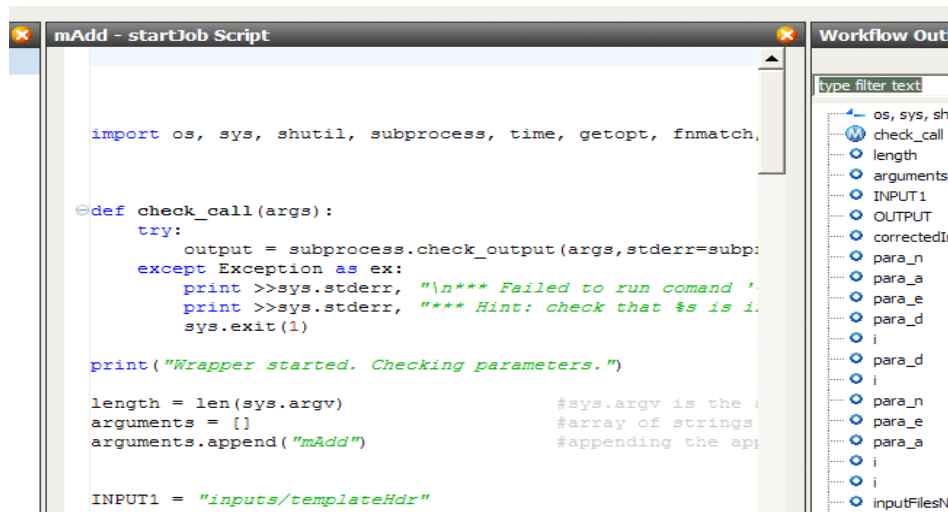


Figure 6.7: Tailoring the Service Behavior

6.3.3 The Monitoring Mode

As soon as a workflow is submitted for execution, a new window is opened to run the workflow in the monitoring mode (see Figure 6.8). In this mode, users can pause the workflow's execution and adapt its model [1]. The can change a non-executed activity's inputs or parameters, add new activities, or delete existing ones. Later, the execution process can be simply resumed [2] while considering the applied modifications. This enables users to achieve what is referred to as "workflow steering".

The progress of the workflow execution process can be identified either by events shown in the logging view [3] or through small icons added to the workflow's activities that reveal the activities' execution states [4]. An activity's execution state may be one of the six values: *un-submitted*, *running*, *completed*, *rescheduling*, *paused*, and *failed*. During the execution process, the SWIMS WMS requests to upload local input files to execution Grid nodes, this is done in the background without any user interaction [5].

The workflow outline view [6] provides a tree representation of a workflow's different elements. This view can be used by users in the monitoring mode to download an executed activity's final output through its provided reference [7]. The downloaded file will be saved within the "downloads" folder located in the workflow's project local folder.

After a successful run of a workflow, users can request the workbench to deploy the workflow as a Grid service [8]; accordingly, the workbench contacts the Service Catalog to determine the Grid node with the least current load and tries to deploy the workflow on it. Subject to a successful deployment, the deployed workflow's service will be shown as an atomic activity in the activities list of all users.

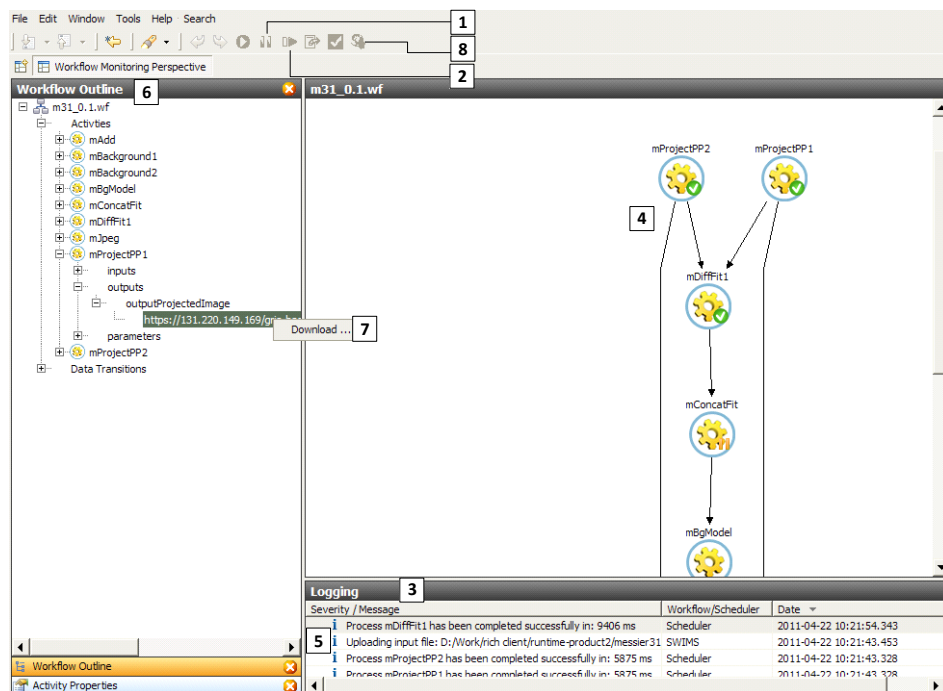


Figure 6.8: SWIMS's Workbench Monitoring Mode.

During the monitoring mode, the SWIMS workbench runs a timer in the background. This timer is reset whenever a message is obtained from the workflow's main coordinator (*SR* instance). The main goal of this timer is to ensure that the workflow's main coordinator is still running in fixed periods (configuration parameter). In case that the main coordinator is broken, the workbench contacts the Service Catalog to find another scheduler through which it can continue the execution of the current workflow. After getting the URL of the new *SR* instance, the workbench retrieves the information about the last recorded checkpoint from the Workflow Catalog and submits it to the new *SR* instance. This process is completely transparent to the user. The user will be notified only if the workbench can't find any other schedulers to continue the execution process.

6.4 Workflow Validation

Scientific workflows tend to be time consuming due to the data intensive and computationally expensive features of their scientific processes. Thus, it is very important to inspect and validate workflows before execution to minimize runtime errors that can happen due to incorrect workflow specifications.

In SWIMS, workflows are validated automatically whenever it is opened or before submission for execution. Meanwhile, users can perform manual validation at any time through the workbench's toolbar or the editor's context menu. A sample

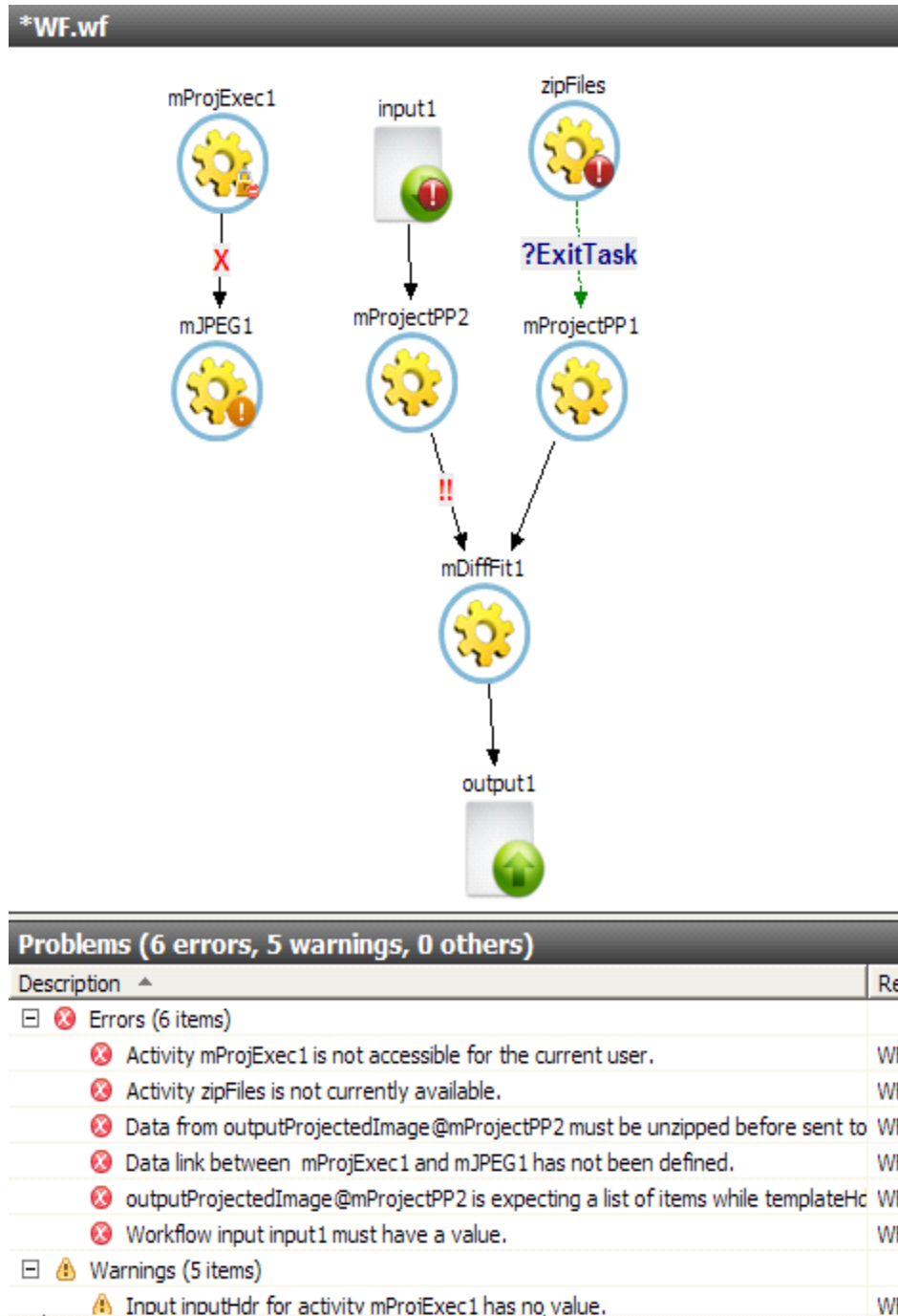





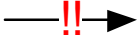



Figure 6.9: Workflow Validation Sample.

for SWIM's workflow validation is shown in Figure 6.9. As shown in the Figure, users are notified of any detected validation errors or warnings through the problems view. Table 6.1 presents the meaning of different validation symbols provided by SWIMS. Validation is done against missing inputs or parameters, accessibility of activities, and correctness of data transitions. The system reports a data transition's type mismatch warning if it can't find a suitable data transformation mediator in the Mediator Catalog.

Table 6.1: SWIMS's Workflow Validation Symbols.

Symbol	Description	Possible Causes
	Faulty workflow input.	workflow input has not assigned a value.
	Faulty activity.	<ul style="list-style-type: none"> • Currently, no available Grid nodes provide such activity. • Required parameter is missing.
	Activity with warnings.	<ul style="list-style-type: none"> • Activity is isolated from the workflow (it has no data or control dependencies) • Some activity's inputs have no value and not bounded with data dependencies.
	Secured activity	The current user has no access to this activity.
	Undefined transition	The transition's source and target ports have not been specified yet.
	Faulty data transition	Source data must be zipped/unzipped before being sent to the target activity.
	Data transition with warnings	Type mismatch between the transition's source and target ports.

6.5 SWIMS: The Overall Picture

Before we introduce the overall picture of SWIMS, we would like to discuss how the SWIMS's server-side and client-side components fit into the *SWfMSs*' reference architecture presented in Section 2.3. SWIMS components have been mapped to the different layers of the reference architecture in Figure 6.10. As stated before, the current version of SWIMS supports Web services and GRIA services; in addition to the services, the *NMR* should also reside in the *Operational Layer* so that it can have low level access and control over the GRIA server. Two components exist in the *Task Management Layer*, the *ER* which is responsible for the execution of tasks and provenance management and the *DMR* which is dedicated for data management purposes. In the *Workflow Management Layer* lies the *SR* that coordinates and monitors the overall execution of workflows. Finally, in the *Presentation Layer*, the workbench allows users to design, modify, and share scientific workflows; in addition, it provides visualization tools for data products and provenance meta-data.

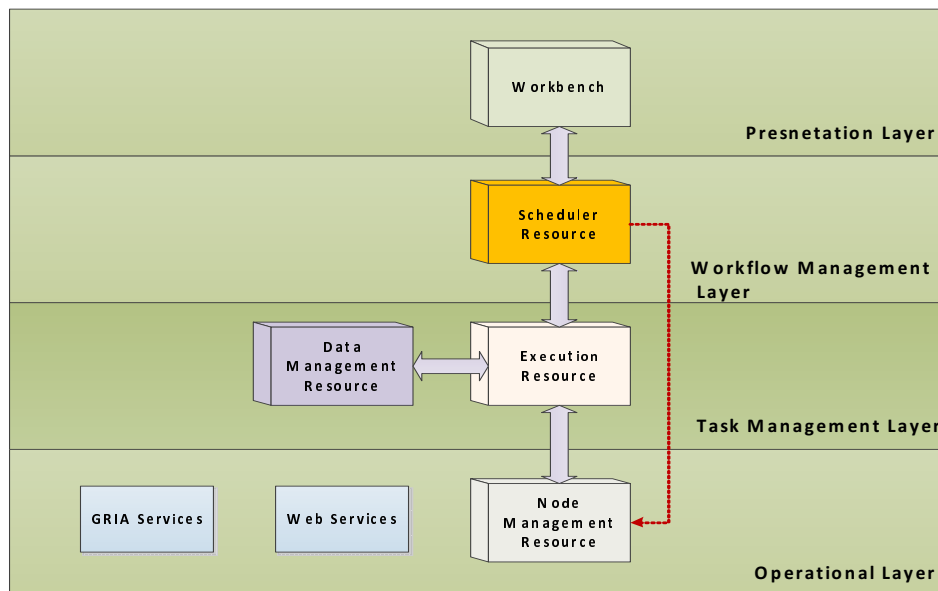


Figure 6.10: SWIMS according to the Reference Architecture in [Lin 2009]

In order to make it clearer how all SWIMS's components work together, we present a simplified sequence diagram for SWIMS usage in Figure 6.11. As noted from the figure, all our WMS resources' instances except the *NMR* are constructed only when they are needed and destroyed after the completion of their job to reduce our system load over the underlying Grid node.

In general, users must first login to the SWIMS workbench. The provided login information is sent to the Service Catalog in order to retrieve the list of available activities and their accessibility status for the current user. The obtained list of activities' abstract templates is used to initialize the activities section in the

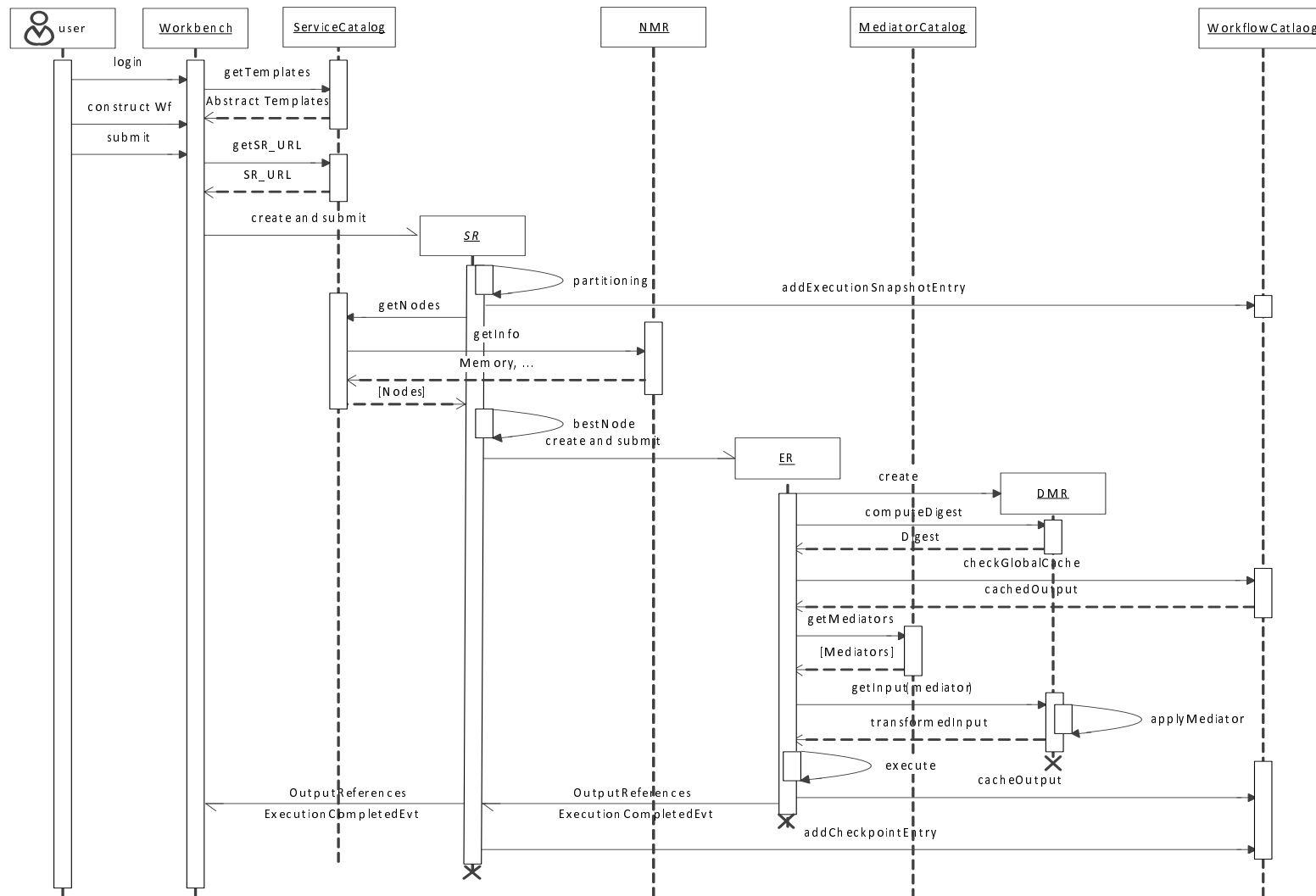


Figure 6.11: SWIMS's Simplified Sequence Diagram [UML 2011].

workbench palette which is used by the user to construct an abstract workflow and submit it for execution.

For submitting a workflow for execution, the workbench needs to find a Grid node where it can create a new *SR* instance and submit the workflow to it. This can be achieved by contacting the Service Catalog to get the node's URL. Upon receiving the workbench request, the Service Catalog checks all nodes where *SR* has already been deployed, and returns the URL of the node with the least number of running *SR* instances. If the workbench fails to create an *SR* instance over the returning node, it has to inform the Service Catalog about this failure to update its status tables.

The *SR* instance follows the lifecycle introduced in Section 5.4 to coordinate the execution of the submitted workflow. First, it divides the submitted workflow into a set of subworkflows and adds its execution snapshot entry in the Workflow Catalog. Then, it plans each subworkflow to an *ER* instance by contacting the Service Catalog to get all nodes which meet the subworkflow's tasks resource requirements and selecting from them the best node according to one of the planning algorithms presented in section 5.4.2. Finally, the *SR* creates a new *ER* instance over the selected node and submits the subworkflow to it. After the completion of the execution of each subworkflow, the *SR* instance stores a checkpoint entry in the Workflow Catalog and notifies the workbench about the subworkflow's final results.

Every created *ER* instance adheres to the lifecycle inferred in Section 5.5 to execute the submitted subworkflow. First of all, it computes the SHA-2 fingerprint of the subworkflow's inputs which is sent to the Workflow Catalog to check whether there exists a cached output for the given fingerprint. In the event of having a negative response from the Workflow Catalog, the *ER* instance contacts the Mediator Catalog to fetch any required mediators, then it contacts the remote *DMR* instances to transform and download required inputs. The retrieved inputs are used to execute the subworkflow over the underlying workflow enactor. A successful execution is followed by a caching of the subworkflow's output and a notification to the *SR* instance holding the output references.

In the previous example, we have focused only on the SWIMS's basic functionalities. However, in a real environment, the workflow management and execution processes can be quite different. For example, fault handling mechanism may be involved to recover from execution failures or the user may decide to interrupt the execution process to apply some modifications.

Summary

In this chapter, we have demonstrated the workbench provided by SWIMS enabling scientists to construct, execute, monitor, steer and share scientific workflows. We focused on showing how the workbench utilizes a high level of abstraction to shield scientists from technical complexities: Users compose their workflows from a set of abstract activities without considering the type of these activities or where they have

been deployed. In addition, authentications between users and secured activities are done through a simple user name and password approach.

The workbench also helps users to enrich their work by allowing them to share and re-use other scientists' experiments stored in the Workflow Catalog. Furthermore, users can adapt the service's behavior or/and interface. They can then deploy the adapted service as a new one and use it in their workflows. Users can also deploy any successfully running workflow as a standalone service that can be used as an atomic activity within more complex workflows. Moreover, to minimize the probability of runtime errors, the workbench allows its users to validate workflows before submission.

In addition to exploring the features of the SWIMS workbench, the chapter highlighted the architecture of the workbench and the structure of the SWIMS abstract XML-based language used to represent constructed workflows. We closed the chapter with a simple usage scenario of SWIMS in order to show how SWIMS's different components (global, server-side and client-side components) cooperate in order to provide a reliable and distributed execution of scientific workflows.

Experiment and Performance Evaluation

Contents

7.1 Montage Workflow Generator	93
7.2 Experiment Design	95
7.3 SWIMS's Server-Side Evaluation	96
7.4 SWIMS's Workbench Evaluation	102

The previous three chapters described the design and implementation of SWIMS, a Grid-based scientific workflow management system. As a part of my thesis work, I have conducted a "test deployment" of SWIMS over a small Testbed constructed based on the Bonn-Aachen International Center for Information Technology (B-IT) resources. The main goal of this deployment was to evaluate the feasibility, usability, capabilities, and the performance of the server and client side components of SWIMS. The performance evaluation was based on Montage workflows from the astronomy domain presented in Section 2.2.1. In order to simplify the construction of the experiment's needed workflows; we supplied the SWIMS's workbench with the capability to generate large scale Montage workflows. The rest of this chapter is organized as follows. First of all, we discuss the Montage workflow generator added to the SWIMS workbench. Second we explain the structure of the Testbed and the different workflows used in the evaluation process. Third we focus on the evaluation of the SWIMS's WMS, and finally; we end up with the evaluation of the SWIMS's workbench.

7.1 Montage Workflow Generator

To facilitate the evaluation of SWIMS environment on a set of Montage workflows with different sizes, we have added a Montage workflow generator to the SWIMS workbench. As we can see in Figure 7.1, the user has to select a set of parameters that describe the mosaic to be constructed, including an object name, a sky survey, a mosaic region size, a size of a pixel (in arcsec) and a coordination system.

The first action of the workflow generator is trying to download the input files required by the requested workflow. This can be achieved through two steps. First, the generator uses the *mArchiveList* Montage module that contacts the

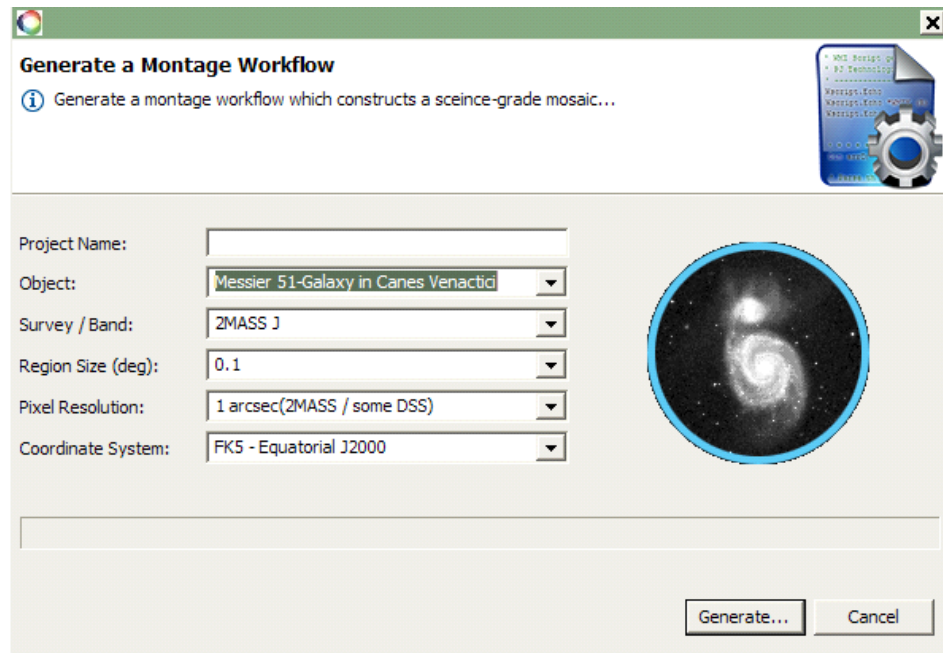


Figure 7.1: Montage Workflow Generator

NASA/IPAC Infrared Science Archive (IRSA) [IRSA 2011] server to retrieve a list of archive images given a location on the sky, and size in degrees.

```
mArchiveList survey band object|location width height region.tbl
```

Then, the table generated by the *mArchiveList* is passed to the *mArchiveExec* module that calls the *mArchiveGet* module on each image in the list to retrieve all required files.

```
mArchiveExec region.tbl
```

After retrieving the input images, the workflow generator uses the *mHdr* Montage module that contacts the IRSA service *HdrTemplate* to create a header template based on a coordination system, resolution, location, and size.

```
mHdr [-s system] [-p pixsize] object|location width template.hdr
```

Another set of headers are also required by the Montage workflow activities. These headers can be obtained through the Montage Grid module *mDAGTbls* which given an image table and a header template; it constructs the tables of projected and background corrected images. The required image table itself can be generated by applying the *mImgtbl* module over the downloaded input images.

```
mImgtbl directory-input-images images.tbl  
mDAGTbls directory-input-images images.tbl template.hdr projectedImages.tbl  
correctedImages.tbl
```

Now and after downloading all required input files, the workflow generator is ready to build the required workflow based on the information about the Montage workflow structure described in Section 2.2.1. The workflow generator creates an mProject job for each input image, and for each pair of input images, an mDiffFit job is created to represent the calculation of the differences of the reprojected images. Then an mConcatFit job is used to merge multiple plane fit parameter files into one file, which is used by the next job, the mBgModel, to determine the background correction required for each image. Afterwards, the workflow generator adds an mBackground job for each re-projected image that applies the recommended correction over the image. The final stages of the workflow, denoted by the pipeline structure in Figure 2.3, represent the creation of the final mosaic. These stages are common to all Montage workflows, and therefore, are also included by the workflow generator.

7.2 Experiment Design

In order to evaluate the feasibility, usability, capabilities, and the performance of SWIMS, we have deployed it on a small Testbed shown in Table 7.1. The Testbed has been constructed over the BIT's 100 Mbps Ethernet LAN. All computational resources have small communication latency ($\leq 1ms$). The execution service instances in the deployed SWIMS server bundles have been configured to run maximum four subworkflows at a time.

Table 7.1: SWIMS Testbed

Node	#CPU	CPU, GHz	Memory	OS	SWIMS Bundle
1	1	Pentium 4, 3.6	2 GB	XP SP3 (32 bits)	Global Catalogs ¹
2	4	Intel Core 2, 2.83	3 GB	Ubuntu 10.04 (64 bits)	Server ²
3	4	Intel Xeon, 2.66	3 GB	Ubuntu 10.04 (32 bits)	Server
4	4	Intel Xeon, 2.66	3 GB	XP SP3 (32 bits)	Server
5	4	Intel Xeon, 2.66	3 GB	XP SP3 (32 bits)	Server
6	2	Intel Core 2, 2.4	2 GB	XP SP3 (32 bits)	Client

The evaluation has been conducted using a ten real world Montage workflows that are used to create (0.1 to 1.0 square degrees) mosaics of the M31 region of the

¹Service, workflow, and mediator Catalogs

²Scheduler, Task, Node Management, Data Management resources

sky. Table 7.2 shows the name of the module and number of tasks at each level of the Montage workflows and the average output size of each module. More details about the functionality of each of these modules can be found at [Montage 2011].

Table 7.2: Number of tasks per level, and average output size of modules in ten Montage workflows (from 0.1 until 1.0 sq. degree)

1	Module	#t (0.1)	#t (0.2)	#t (0.3)	#t (0.4)	#t (0.5)	#t (0.6)	#t (0.7)	#t (0.8)	#t (0.9)	#t (1.0)	Avg. out. (KB)
1	mProject	2	6	8	12	16	24	28	32	36	45	1970
2	mDiffFit	1	15	28	66	120	276	378	496	630	990	0.4
3	mConcatFit	1	1	1	1	1	1	1	1	1	1	72
4	mBgModel	1	1	1	1	1	1	1	1	1	1	1.5
5	mBackground	2	6	8	12	16	24	28	32	36	45	1820
6	mAdd	1	1	1	1	1	1	1	1	1	1	38000
7	mJpeg	1	1	1	1	1	1	1	1	1	1	387
	Total	9	31	48	94	156	328	438	564	706	1084	

7.3 SWIMS's Server-Side Evaluation

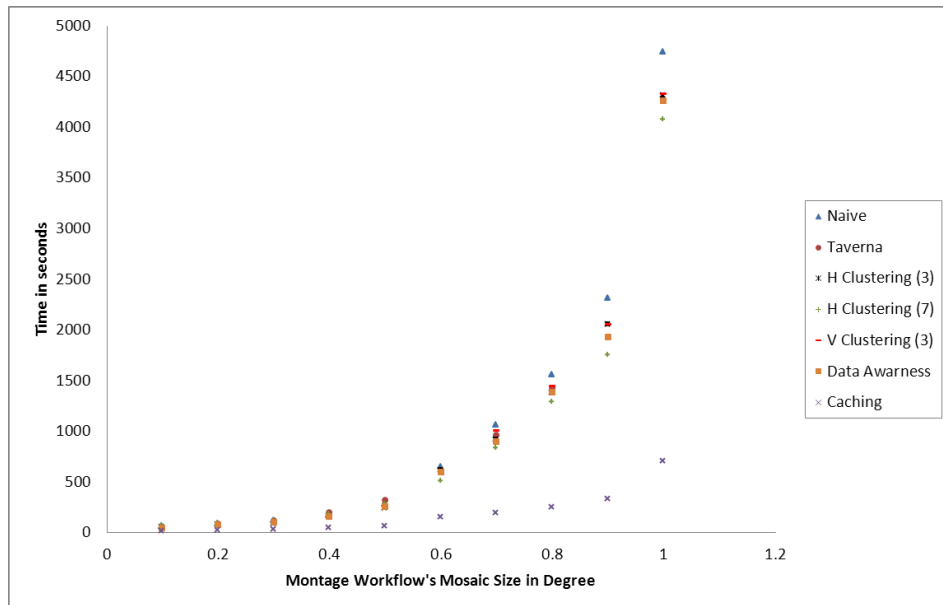


Figure 7.2: Average Completion Time (Montage's Experiment)

In order to evaluate the SWIMS's WMS performance, we have executed the workflows represented in Table 7.2 over the SWIMS's Testbed. Figure 7.2 shows

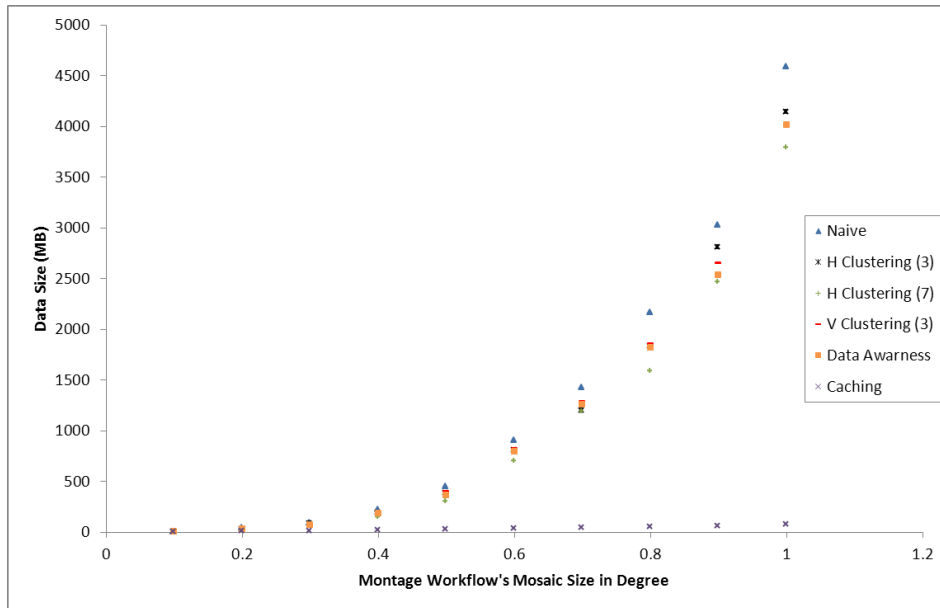


Figure 7.3: Amount of Data Transfer (Montage's Experiment)

the workflows completion times on Taverna 1.7³ and on SWIMS's Testbed with different conditions:

1. Naive run without applying any clustering techniques and using the reliability-based scheduling presented in Section 5.4.2.
2. Planning based on data-aware scheduling.
3. Horizontal clustering with clustering sizes three, and seven.
4. Vertical clustering with clustering size three.

The completion time shown in the Figure is the average of three runs. According to the obtained results, we can conclude the following:

1. For small and medium scale workflows, the execution time of the SWIMS's naive run is quite close to the execution time over Taverna, that employs a single centralized execution engine. For large workflows (0.9 and 1.0 degrees), Taverna produced an "Out of Memory" exception. While Taverna's engine could execute medium workflows (0.6 - 0.8 degrees), its workbench hanged due to the need of rendering a quite large number of tasks; in this case, we had to collect the execution information from Taverna's log files.
2. Data-aware scheduling decreases the execution time of large workflows (0.7 - 1.0 degrees) from 10% up to 16%. This improvement is due to the reduction

³Taverna 1.7 uses the same execution engine exploited in SWIMS's WMS (Freefluo enactor)

of the amount of data transferred as shown in Figure 7.3. As observed from the Figure, the diminution of data is rather small and only observable in large workflows as the transferred data between Montage’s workflows individual tasks is quite small as perceived in Table 7.2 (Avg. out.).

3. Clustering techniques can also depreciate the execution time of large workflows as they lead to less scheduling overhead as we will show later and less data transfer as stated in Figure 7.3. The reduction percentage ranges from 7% up to 11%, 9% up to 14%, and 14% up to 24% for vertical clustering (cluster size 3), horizontal clustering (cluster size 3), and horizontal clustering (cluster size 7) respectively. An important remark of this result is that increasing the clustering size in case of horizontal clustering leads to improvement in the overall performance, especially for workflows with large numbers of tasks in one level as the case for Montage’s workflows (in levels one, two, and five).
4. A second run of a workflow while enabling global caching can be extremely fast (up to 85% faster). In an execution fully based on cached outputs, only initial inputs need to be actually transferred. Intermediate data is not needed to be transferred because the *DMR* instances, locally located on the nodes where the data resides, are responsible for computing the data’s SHA-2 digest required by the caching algorithm (see Section 5.6.4).

We also plot the average speedup of tasks for all runs in Figure 7.4. The average speedup of an execution of type x is defined as $(avg_runtime_x / avg_runtime_{naive})$. As we can see, clustering techniques can have a higher impact on tasks that run simultaneously in a large number (e.g. *mProject*, *mDiffFit* and *mBackground*) because in this case, the workflow enactor needs to create a single workflow instance for all clustered tasks rather than creating a workflow instance for each task as the case in the naive execution. Data-aware scheduling has more effect on tasks with a large number of inputs (e.g. *mConcatFit*, and *mAdd*) or tasks with a quite large input (e.g. *mJpeg*).

Figure 7.5 shows that the overall overhead in SWIMS does not exceed 7% of the workflow’s execution time. Furthermore, the effective parallelism of tasks in SWIMS almost neglects this overhead; the naive execution of Montage workflows over SWIMS is very close to the execution of them over a centralized engine (Taverna) as discussed before. The figure also emphasizes that clustering techniques can help to reduce the overhead as it leads to fewer numbers of subworkflows.

The measured overhead includes workflow partitioning, checkpointing and other scheduling tasks, including workflow planning, and data caching. Figure 7.6 emphasizes that the percentage of these tasks varies according to the type of the execution. For example, in horizontal clustering, the workflow partitioning process dominates the overhead time, whereas we need to compute the level of every task in the workflow in order to partition it. The computation of a task’s level is a quite expensive recursive process.

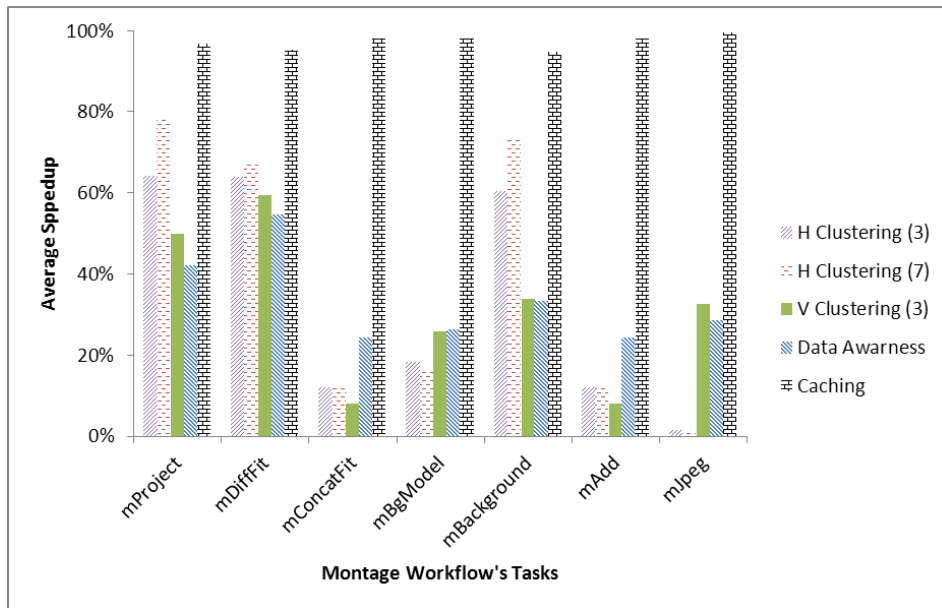


Figure 7.4: Average Task's Execution Speed-up

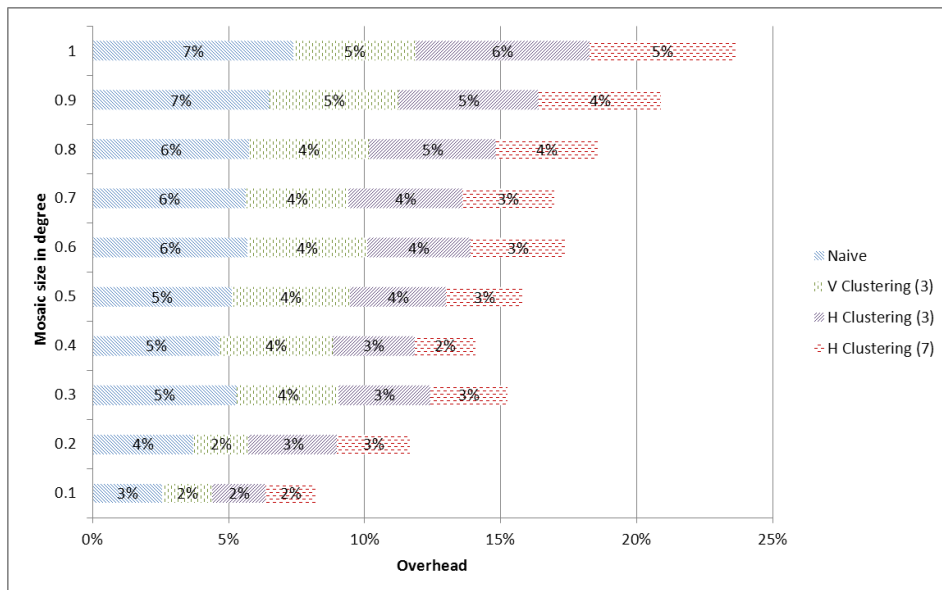


Figure 7.5: All Overhead in SWIMS

The Montage workflows-based experiment ensures SWIMS's capability to manage large scale workflows from the number of tasks point of view. In order to emphasize SWIMS's ability to handle a large amount of data, we have carried out another experiment using a simple workflow shown in Figure 7.7. The workflow

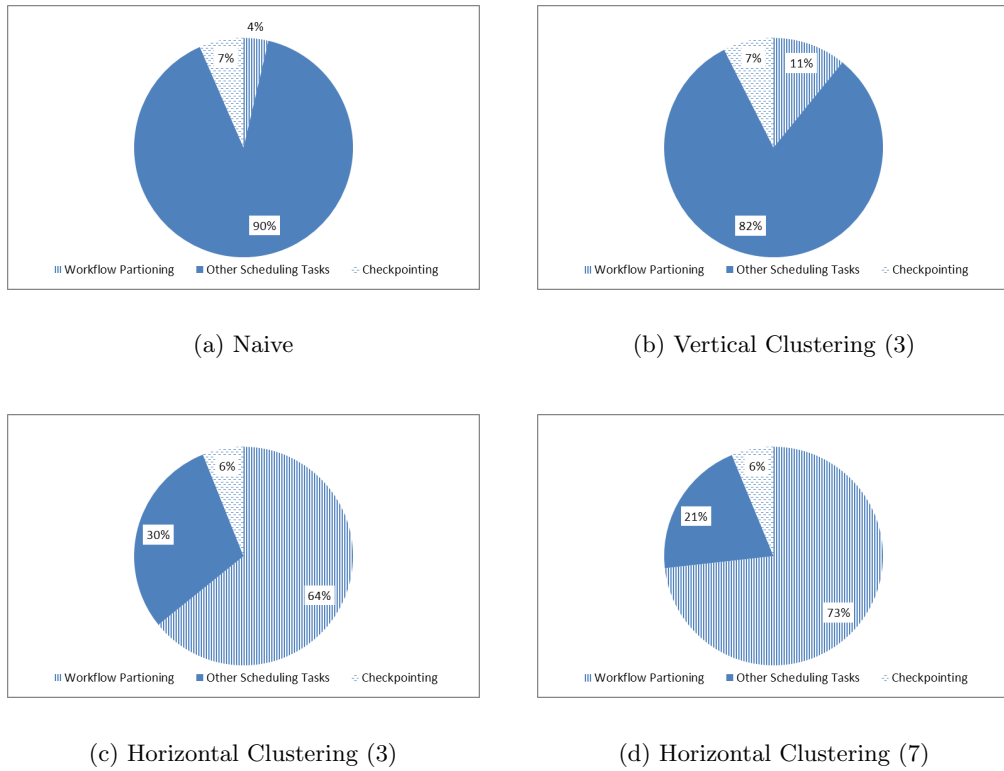


Figure 7.6: Distribution of SWIMS Overhead

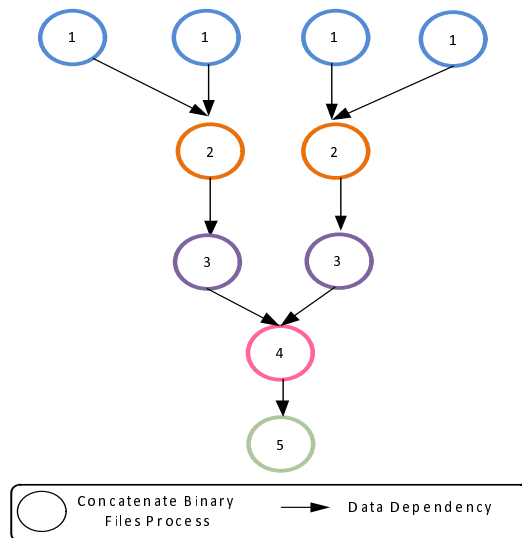


Figure 7.7: Data-intensive Workflow

consists of a set of "concatenate_binary_files" used to concatenate a given set of binary files into a single file. For this experiment, we used ten of this workflow

with different input data size. In each workflow, the tasks in level one have been populated with two input files of varying size 200, 400,...,2000 MB. The maximum output size of a task in this experiment was 16,000 MB while the maximum amount of data transferred during a workflow execution was 80,000 MB.

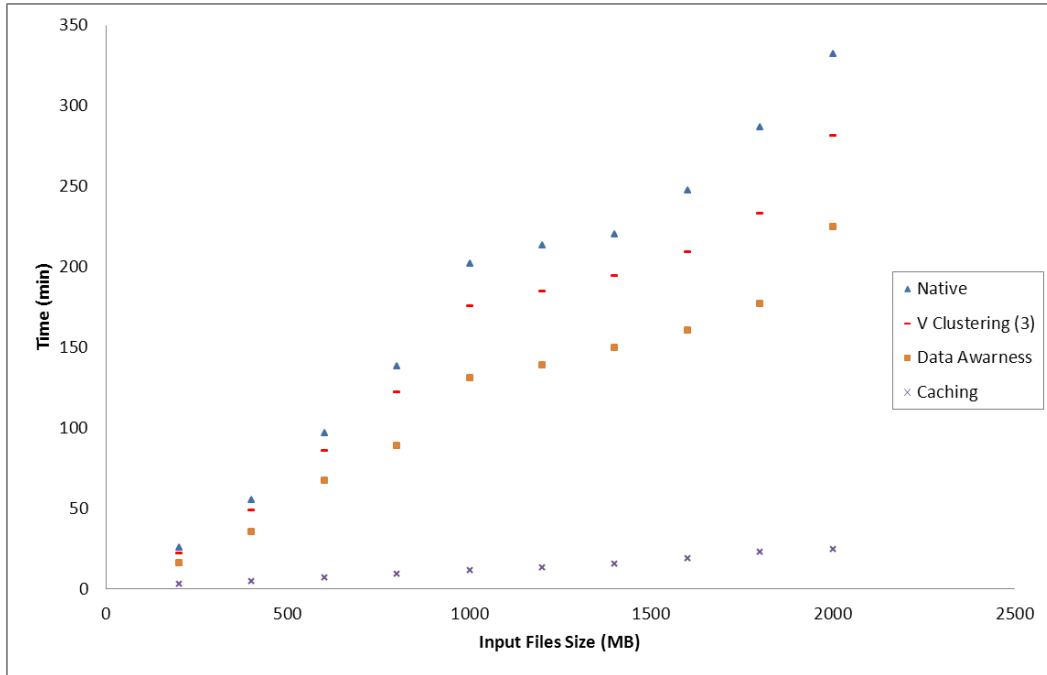


Figure 7.8: Average Completion Time (Data Intensive Experiment)

Figure 7.8 represents the workflow completion times on SWIMS Testbed while Figure 7.9 depicts the amount of data transferred during the execution of each of these workflows. As indicated from the figures, we have only plotted the results of data-aware scheduling, vertical clustering and data caching executions as they are more valuable for data intensive workflows. They lead to 30% up to 38%, 12% up to 18%, 90% up to 94% reduction in the workflow execution time respectively. We believe that these techniques can lead to better performance improvement when the underlying Cyberinfrastructure is distributed over a wide area network where data transfer is more expensive. Horizontal clustering techniques make no sense for this type of workflows that contains a limited number of tasks at each level.

During the experiment, we have also enforced a set of exception events to test the SWIMS's fault handling capabilities discussed in Section 5.7 and to evaluate the overall system reliability. The encountered events include:

- Shutting down an *ER* instance while executing a subworkflow.
- Deploying faulty Gria services. In this case, checkpoints were used to resume the workflow after correcting the faulty services.

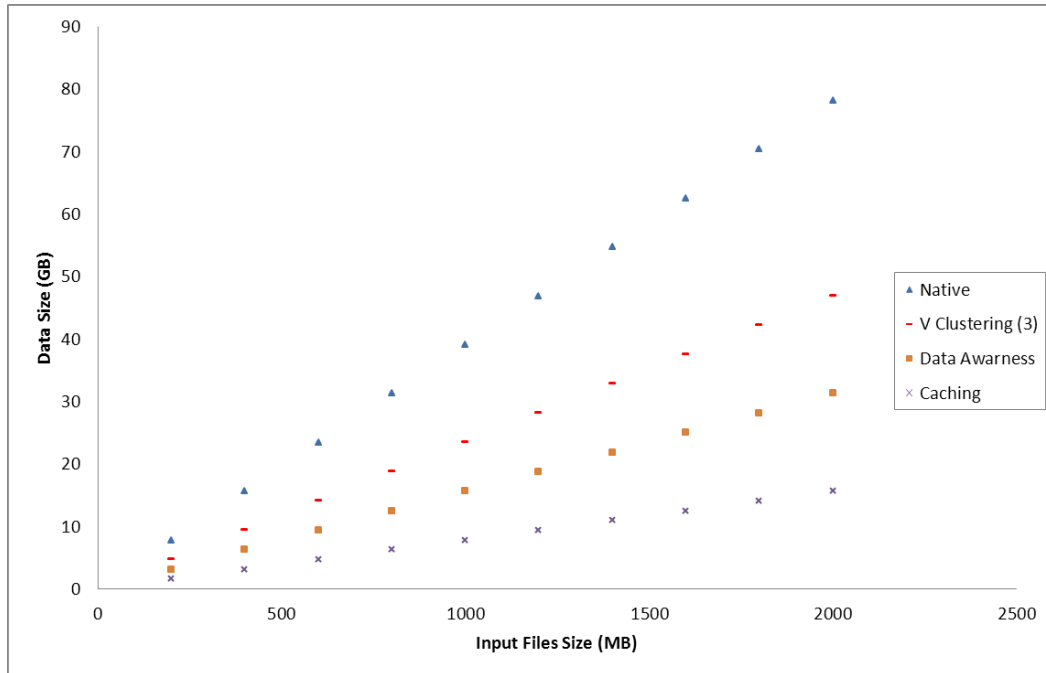


Figure 7.9: Amount of Data Transfer (Data-Intensive Experiment)

- Shutting down the workflow’s **main scheduler** during the execution.
- Shutting down a random Grid node.

To sum up, SWIMS can achieve scalable and reliable distributed execution of scientific workflows through exploiting several WMS instances. The execution of every concrete service is fully controlled by the *ER* instance deployed on the service’s node. The *ER* instance is also capable of handling execution errors without the need of the workflow’s main scheduler through its local *SR* instance. This leads to a distributed fault handling paradigm over the services’ level. The main scheduler of the workflow stores execution checkpoints in the Workflow Catalog that can be used by users either to recover from failures or to achieve smart-rerun. In smart-rerun, users can restart the workflow from any point after modifying it. All these features have been achieved with negligible overhead (up to 7%) when it is compared to their benefits.

7.4 SWIMS’s Workbench Evaluation

In this section, we evaluate SWIMS’s workbench against the system usability challenges discussed in Section 2.4:

- *Ease of use*: SWIMS provides a visual workbench that allows scientists to construct DAG scientific workflow from a set of abstract activities. These

activities are constructed from abstract templates obtained from the Service Catalog. The abstract templates contain the activities' inputs, outputs and parameters that help the system to guide and restrict users during the construction process. The update of available services in the underlying Cyberinfrastructure is done automatically via notifications sent from the *NMR* instances on different available servers to the Service Catalog. In this way, SWIMS shields scientists from the technical complexities of the underlying Cyberinfrastructure. Even more, scientists don't have to bother themselves regarding fault handling or even data transformation between heterogeneous services if mediators have already been constructed for these services.

- *Handling security credentials*: SWIMS tries to hurdle the security issue by applying a double-layer security model as shown in Figure 4.4. The first layer defines the security between a user and the SWIMS system that is based on a simple user name and password mechanism that has to be defined only once. The second layer represents the security between the SWIMS system and different Grid sites (in our case GRIA servers) based on X.509 certificates automatically generated by the WMS instances deployed on the corresponding servers. The security information is stored and become accessible to all other nodes through the Service Catalog. The administrator of every node should be able, through a simple web-based interface, to classify the node's available services to public and private services, and to assign access roles from the list of available users to their private services. A SWIMS user will be able to execute workflows that contain only public services and private services that have been assigned as accessible to him. By using this approach, we attempt to move the security complications from users to server administrators who in most cases have enough IT knowledge to handle them.
- *Workflow monitoring*: During the workflow execution, SWIMS workbench switches to the Monitoring mode where users can follow the progress of the workflow execution process either in text mode through events shown in the logging view or in graphical mode by virtue of small icons added to the workflow's activities that reveals the activities' execution states (see Section 6.3.3).
- *Workflow steering*: SWIMS users can pause the execution of a workflow in two ways. First, by using breakpoints in the editing mode before submitting the workflow for execution (the workflow stops before executing the task with the breakpoint). The second way is to execute the pause command during the execution in the monitoring mode; this will pause the workflow after the completion of the currently running tasks. In addition, users can adapt the paused workflow (see Section 6.3.3).
- *Workflow Sharing*: Workflows in SWIMS are shared among users through the Workflow Catalog that forms a shared workspace. Through this workspace,

users can access workflows already executed by other scientists, their execution snapshots and provenance/error reports (see Section 4.3). Through the "Workflow Catalog View" in the SWIMS workbench, users can explore the contents of the Workflow Catalog in order to re-use previously designed workflows as a template for creating new ones, or to re-run an experiment from a certain checkpoint, or even to analyze the workflow's provenance information or error logs for better understanding of its final results (see Section 6.3.2).

- *Expose workflows as services*: After a successful run of a workflow, users can request the workbench to deploy the workflow as a Grid service (see Section 6.3.3).
- *Workflow validation*: In SWIMS, workflows are validated automatically whenever it is opened or before submission for execution. Meanwhile, users can perform manual validation. The validation process ensures the following: i) all required inputs and parameters are available, ii) all workflow's activities are online and accessible to the current user, iii) no errors or mismatches exist in the workflow's data transitions (see Section 6.4).

Summary

In order to evaluate the usability, and the performance of the SWIMS's WMS, we have conducted a "test deployment" over a small Testbed of six nodes. The evaluation has been done through two different experiments. The first experiment involved ten Montage workflows to test the SWIMS's capability of handling large scale (up to 1084 tasks) workflows. The second experiment has been designed to emphasize SWIMS's ability to handle a large amount of data. It encounters ten simple workflows that exchange a huge amount of data (up to 80,000 MB).

The experiments' results showed that the SWIMS naive execution provides a quite acceptable performance that is nearly equal to the execution of the same workflows on a local engine (Taverna) with an overhead less than 7% of the total execution time. Data-aware scheduling and vertical clustering can improve the execution performance, especially for data intensive workflows (up to 38% and 18% respectively). Horizontal clustering is more suitable for workflows with a large number of tasks in the same level could achieve performance improvement up to 24% for the large scale Montage workflows. Data caching can support an extremely fast smart re-run. We could re-execute a non-modified workflow up to 94% faster than the first run. Another important remark here is that vertical clustering, data-aware scheduling and data caching can help to reduce the amount of data transferred during the execution process. For example, in the second experiment, these features were able to decrease the amount of transferred data up to 40%, 60%, and 80% respectively.

In addition, the SWIMS workbench has been evaluated against the system usability challenges discussed in Chapter 3.

From the previous results, we can derive a set of scenarios where utilizing of SWIMS can provide a great advantage. First, SWIMS can achieve a scalable and reliable execution for complex workflows, with a large number of parallel tasks, or for data intensive workflows. Second, smart-rerun capability in SWIMS is very helpful for repetitive scientific experiments that need to be executed several times with minor modifications of their tasks' parameters. Last but not least, SWIMS is a perfect choice for a set of organizations from the same domain which would like to cooperate with each other. These organizations can deploy SWIMS with a shared Global catalogs. This will allow the users of every organization to access the services offered by other organizations; they can also explore and re-use scientific experiments developed by others, including their provenance and error reports.

Conclusion and Future Trends

Contents

8.1 Summary of the Thesis	106
8.2 Accomplishments of the Thesis	108
8.3 Future Work	112

8.1 Summary of the Thesis

In this thesis, I have proposed, designed, implemented, and evaluated a framework (SWIMS) for scientific workflow management and execution. SWIMS provides a new execution paradigm that utilizes several workflow management systems (SWIMS's WMS instances). These *WfMSs* cooperate with each other to coordinate and execute scientific workflows in a distributed and reliable manner. In addition, SWIMS offers a client workbench with a high level of abstraction and extensive sharing capabilities in order to hide complex technical details from its users and to enrich their work environment.

I started my work by a practical evaluation of a set of selected *SWfMSs* and a review of literature on existing scientific workflow and management systems and their challenges (see *Chapter 2*, and *Chapter 3*). My study established that existing systems still face key challenges in the different stages of the workflow management lifecycle (summarized in Tables 3.1 and 3.2). Accordingly, I have proposed the SWIMS framework that employs Web services technology to provide a new execution paradigm that utilizes several workflow management systems. SWIMS introduces a complete scientific workflow management environment in the sense that it fills in all the gaps discovered in the different phases of the workflow management lifecycle.

The SWIMS architecture (see *Chapter 4*) consists of a client workbench, and a set of Workflow Management System (WMS) deployed on nodes wishing to participate in the workflow management and execution process. In addition, there are three global catalogs: First, the Mediator Catalog stores mediators necessary for data transformations between heterogeneous services. Second, the Workflow Catalog acts as a shared workspace to hold all workflows-related information, including specifications, execution checkpoints, provenance reports, error logs, and cached outputs. Finally, the Service Catalog keeps track of all available concrete services (Web services and GRIA services).

To simplify the implementation of the SWIMS's WMS (see *Chapter 5*), I have divided it into four main components: data management (*DMR*), node management (*NMR*), scheduler (*SR*) and execution (*ER*) resources. With the help of these components, I could provide solutions to the system efficiency and reliability challenges discussed in Section 2.4 as shown in Table 8.1. First, the management and execution of scientific workflows in SWIMS are done by several *SR* (in case of failures) and *ER* instances that ensure the reliability and extendability of the SWIMS system. Second, all planning algorithms, applied by the *SR*, consider the load balancing between different execution nodes. Third, the *DMR* is responsible for reference-based data movement, and automatic data transformation between heterogeneous services based on the mediators stored in the Mediator Catalog. Fourth, fault handling in SWIMS is completely transparent to its users. Every *ER* instance is responsible for handling errors that occur during the execution of its node's concrete services. The *ER* instance can retry the service or use its node's local *SR* instance to find an alternative Grid node to execute the failed task. In addition, as the *ER* instance is located at the concrete service's node, it can generate low-level error reports, including faults occurred at the OS-level. This helps service owners to quickly recover the causes of failures. The workflow's main scheduler (*SR*) ensures that all active *ER* instances are working smoothly; otherwise, it re-schedules the tasks of any failed *ER* instance. Even if the workflow's main scheduler itself has failed, another *SR* instance is elected transparently to continue the execution of the workflow. Fifth, the workflow's main scheduler submits an execution checkpoint to the Workflow Catalog after the execution of every task. These checkpoints can be used either to recover from errors or to re-run the workflow from any selected point after modifying its specifications. Finally, the *ER* instance caches the output of every successfully executed task globally in the Workflow Catalog; this helps the SWIMS environment to avoid re-running of a time consuming service if it has been already executed on the same given input.

The SWIMS workbench (see *Chapter 6*) enables scientists to construct, execute, monitor, steer and share abstract scientific workflows. The SWIMS workbench tries to overcome the system usability challenges highlighted in Section 2.4 as indicated in Table 8.2. First, the workbench utilizes a high level of abstraction to shield scientists from technical complexities; users compose their workflows from a set of abstract activities without considering the type of these activities or where they have been deployed. Second, authentication between users and SWIMS is done through a simple user name and password approach, while X.509 certificates are used for transparent authentication between SWIMS and different available Grid servers. Third, SWIMS users can monitor the execution of their workflows in both text and graphical mode. Fourth, users can steer workflows during execution by pausing, and editing them. Fifth, the SWIMS workbench helps to enrich scientists' work by allowing them to share and re-use other scientists' experiments stored in the Workflow Catalog; they can also explore the provenance and error reports generated during the execution for better understanding of the final results. Even more, they can tailor the behavior of an existing GRIA service and deploy it as

a new service. Sixth, SWIMS's workflows can be deployed as standalone services that can be used as atomic activities within more complex workflows. Last but not least, abstract workflows are validated before submission to ensure that workflows' mandatory inputs and parameters are available, workflows' activities are online and accessible to the current user, and there are no errors or mismatches in workflows' data dependencies. The only limitation of the SWIMS workbench is that it is bounded to DAG workflows as it does not support iterative constructs.

At the end, I have evaluated the usability, and the performance of the SWIMS's WMS through conducting a "test deployment" over a small Testbed of six nodes (see *Chapter 7*). The evaluation has been done through two different experiments. The first experiment involved ten Montage workflows to test the SWIMS's capability of handling large scale (up to 1084 tasks) workflows. The second experiment has been designed to emphasize SWIMS's ability to handle a large amount of data. It encounters ten workflows that exchange huge amount of data (up to 80,000 MB). The experiments' results showed that the SWIMS naive execution provides a quite acceptable performance that is nearly equal to the execution of the same workflows on a local engine (Taverna) with a total overhead less than 7% of the overall execution time. Data-aware scheduling and vertical clustering can improve the execution performance, especially for data intensive Workflows (up to 38% and 18% respectively). Horizontal clustering is more suitable for workflows with a large number of tasks in the same level and could achieve performance improvement up to 24% for the large scale Montage workflows. Data caching can support an extremely fast smart re-run. In my experiments, a workflow run based on cached outputs was up to 94% faster than the first run.

8.2 Accomplishments of the Thesis

The significance of this research is that it tackles existing problems in the workflow management lifecycle. This leads to a set of improvements in the process of scientific workflows management and execution from two different points of views, system efficiency and system usability. This research combines several existing technologies to provide a real scientific workflow execution environment (SWIMS) that exploits several workflow management systems to better reflect the distributed nature of scientific workflows. Moreover, it considers ease of use and extensive sharing aspects to support users with limited IT knowledge.

The main outcomes of this research regarding system efficiency are as follows:

- Even though the new execution paradigm exploits several execution engines and schedulers, the coordination of a workflow is done by a single scheduler for better monitoring and tracking of execution information. However, SWIMS is still highly reliable as it keeps the execution status of running workflows in a global shared space (Workflow Catalog). In case of a failure of a workflow's coordinator, a new one is elected transparently to continue the workflow's execution.

- Having a complete workflow management system on every node improves the overall system openness. Workflow participants can easily join and leave the execution process at their own initiative at any time without affecting the current status of the whole system.
- SWIMS is highly scalable. Every new joining node provides a scheduler for coordinating more workflows and an execution engine for running more tasks.
- Exploiting several execution engines helps SWIMS to provide distributed fault handling on the service level. In addition, every execution engine has access to low-level error reports generated during the execution process. This helps to give a better insight of occurred faults.
- SWIMS offers a low cost global data caching based on an SHA-2 fingerprint of workflows' inputs and the workflows' output references. This helps to avoid the rerunning of time consuming scientific tasks.
- As shown in Figures 7.3 and 7.9, vertical clustering, data-aware scheduling, and data caching features helps to reduce the amount of data transferred during the execution process.
- Data transformation mediators, created by a user, are shared in a global Catalog. SWIMS uses these mediators to achieve automatic data transformation between heterogeneous tasks.

Moving to the system usability, our research contributes the following:

- Ease of use is one of the main goals of SWIMS's client side. It isolates its users from any technical details through high level of abstractions. It provides also a simple authentication mechanism based on user names and passwords. Users don't have to take care of fault handling that is achieved transparently by the environment itself. They don't have also to consider data transformations between heterogeneous services if data transformation mediators for these services already exist in the Mediator Catalog.
- SWIMS provides a completely open and sharing environment that enriches the working surroundings of scientists. Assume a scenario that several scientific organizations from the same domain deployed SWIMS as their workflow management and execution environment. The users of every organization will be able easily to make use of the public services or to request access permission to secured services offered by other organizations. Users can also explore and re-use scientific experiments developed by others, including their provenance and error reports.
- SWIMS offers adaptation and tailoring capabilities to its users. Users can update their workflows while running, update others' workflows and re-execute them, or even tailor the behavior of an atomic service by modifying its script / interface and redeploying it as a new service.

Table 8.2: Comparison of Workflow Management Systems (SUC) - Recalled

SUC	Taverna 2.x	Kepler	Unicore	Pegasus	SWIFT	ASKALON	SWIMS
Ease of Use	- Structure: DAG - Model: concrete - Composition: graph-based	- Structure: Non-DAG - Model: concrete - Composition: graph-based	- Structure: Non-DAG - Model: abstract/ concrete - Composition: graph-based	- Structure: DAG - Model: abstract - Composition: language-based	- Structure: Non-DAG - Model: abstract - Composition: language-based	- Structure: Non-DAG - Model: abstract - Composition: graph-based (UML)	- Structure: DAG - Model: abstract - Composition: graph-based
Handling Security Credentials	X.509 certificates	X.509 certificates	X.509 certificates	Globus My-Proxy	X.509 certificates	Globus My-Proxy	Simple user name and password
Workflow Monitoring	-Graphical Mode	-Graphical Mode (SDF Directory)	-Graphical Mode	-Text Mode	-Graphical Mode -Text Mode	-Graphical Mode	-Graphical Mode -Text Mode
Workflow Steering	supported	Workflow parameters	N/A	N/A	N/A	N/A	supported
Workflow Sharing	myExperiment -Workflow Source	Kepler repository -Workflow Source	N/A	N/A	N/A	N/A	Workflow Catalog -Workflow Source -Checkpoints -Provenance
Expose Workflows as Services	N/A	KAR archives	N/A	N/A	N/A	N/A	GRID service
Workflow Validation	-Data types -Scripts -Services availability	N/A	N/A	N/A	N/A	N/A	-Missing inputs, parameters -Services: -availability -accessibility -Data transitions

8.3 Future Work

This dissertation represents a step in ongoing research efforts to realize the vision of distributed, reliable, extendible, and easy to use *SWfMS* for managing scientific workflows over Grid environments. There are still several issues worthy of future investigations. We have already started a *large scale deployment* of SWIMS over a set of available clusters in the Astronomy department in Bonn University for a better evaluation of the system performance. We are also planning to exploit SWIMS for more *use cases from other domains* (e.g., Bioinformatics). Another future plan is to investigate the capability and the cost of *deploying SWIMS over a Cloud environment* (e.g., Amazon EC2).

Thanks to the separation of functionalities and the autonomic nature of the SWIMS's different components, each of them can be improved separately by adding further functionalities to it. In the following subsections, we are going to explore some of these possible extensions.

1. WMS's Scheduler Resource

- **Iteration support:** The partitioning algorithm needs to be extended in order to be able to interpret and handle iteration constructs. This is a mandatory requirement for supporting non-DAG workflows in SWIMS.
- **Other scheduling algorithms:** Other scheduling algorithms such as algorithms based on performance predictors (e.g., [Blythe 2005, Wiczorek 2005]) can be applied and evaluated for the planning phase.

2. WMS's Task Resource

- **Support other services:** Currently, SWIMS supports Web services and GRIA services. It is possible to add support to new type of services (e.g. Globus, Unicore services) by simply adding new plug-ins to the Freefluo engine that allows the engine to manage and execute them. We will need also to extend the *NMR* to allow the detection of the deployment / undeployment of the new services and to generate abstract templates from their descriptions.

3. WMS's Node Management Resource

- **Code Movement:** As indicated in Section 5.6.1, a single scientific application can produce a huge amount of data. In this case, moving the service itself to the data location will be more cost-effective than moving the data. We have already run an attempt to move GRIA service from one node to another. Nevertheless, we faced several obstacles (see Section 5.6.1). The proposed idea needs to be refined through a further deep research to form stable and consistent solutions for the existing problems.

4. SWIMS Workbench

- *Visualization Plug-ins:* We are planning to add a set of visualization plug-ins for better visualization and analysis of different data items. The added plug-ins will depend on the use cases that will be involved by SWIMS in the future.

GRIA Job Description File

An GRIA application description file is an XML file containing metadata about the application. For example, the following XML describes the Swirl application:

```
<?xml version="1.0" encoding="UTF-8" ?>

<GriaApplicationDescription
  xmlns="http://www.it-innovation.soton.ac.uk/2007/grid/application">
  <JobServiceMinVersion>5.2</JobServiceMinVersion>
  <Application>
    <Description>Application to swirl an image</Description>
    <ApplicationName>
      http://it-innovation.soton.ac.uk/grid/imagemagick/swirl
    </ApplicationName>
    <ApplicationVersion>2.0-1</ApplicationVersion>
    <Group>graphics</Group>
  </Application>

  <DataStagers>
  <DataStager type="input" name="inputImage">
    <Description>Input image to be swirled</Description>
    <MimeType>image</MimeType>
  </DataStager>

  <DataStager type="output" name="outputImage">
    <Description>Swirled image</Description>
    <MimeType>image</MimeType>
  </DataStager>
  </DataStagers>
</GriaApplicationDescription>
```

Every application provided by the GRIA Job Service must be given a unique ***ApplicationName***. To ensure uniqueness, a URI is used. Inputs and outputs are defined as ***DataStager*** elements, with the type attribute set to "input" or "output", as appropriate. Note that you can add as many inputs/outputs as necessary, according to your application. An application might require arrays of inputs, whose exact sizes are specified by the user when creating the job. This is supported by GRIA using the ***minOccurs***, ***maxOccurs*** and ***defaultSize*** attributes on DataStager elements. For example, if your application took between two and eight images as input, you might use the following XML:

```
<DataStager type="input" name="inputImage" minOccurs="2" maxOccurs="8"
  defaultSize="2">
  <Description>Input image</Description>
  <MimeType>image</MimeType>
```

```
</DataStager>
```

Optional inputs are described much like arrays, except the minOccurs attribute is zero and the maxOccurs attribute is one. If your application accepts a set of command line configuration arguments, you can force the Job Service to validate them, before being passed to the application wrappers, by providing a description of them in the metadata file. For example:

```
<Parameters>
  <Parameter name="string" qualifier="--string" type="string"
    minOccurs="0" maxOccurs="1" />
  <Parameter name="bool" qualifier="--bool" type="boolean"
    minOccurs="0" maxOccurs="1" />
  <Parameter name="data" qualifier="" type="string"
    minOccurs="1" maxOccurs="1">
    <allowed>one</allowed>
    <allowed>two</allowed>
    <allowed>three</allowed>
  </Parameter>
</Parameters>
```

This will allow the following command lines:

```
--string "This is a string" one
--bool three
```

The *mimeType* sub-element in the DataStager element defines the type of the presented input/output. SWIMS use a set of user-defined mime types in order to provide a better understanding between the SWIMS client and the remote application. The following table shows different categories of mime types used in SWIMS.

Standard Mime Types e.g. image, text	the well known mime types which reflects a single file of this type
"zip" + Standard Mime Types e.g. zipImage, zipText	reflects a single file from the specified type which needs to be zipped before used/produced by the application
"archive" + Standard Mime Types e.g. archiveImage, archiveText	reflects a list of files from the specified type which needs to be zipped before used/produced by the application

GRIA Job To Abstract Template XSL Transformation

```

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:b="http://www.it-innovation.soton.ac.uk/2007/grid/application"
  exclude-result-prefixes="b">
  <xsl:output method="xml" indent="yes" omit-xml-declaration="yes" />

  <xsl:template match="b:GriaApplicationDescription">
    <ActivityDescription>
      <xsl:attribute name="type">gria</xsl:attribute>
      <xsl:attribute name="name">
        <xsl:value-of
          select="substring-after(b:Application/b:ApplicationName, '/')"/>
        </xsl:attribute>
      <xsl:attribute name="secured">true</xsl:attribute>
      <Description>
        <xsl:value-of select="b:Application/b:Description" />
        </Description>
      <xsl:apply-templates />
    </ActivityDescription>
  </xsl:template>

  <xsl:template match="b:JobServiceMinVersion">
  </xsl:template>

  <xsl:template match="b:Application">
  </xsl:template>

  <xsl:template match="b:Parameters">
    <xsl:element name="{local-name()}">
      <xsl:apply-templates select="@* | node()" />
    </xsl:element>
  </xsl:template>

  <xsl:template match="b:Description">
    <xsl:element name="{local-name()}">
      <xsl:apply-templates select="@* | node()" />
    </xsl:element>
  </xsl:template>

  <xsl:template match="b:Parameter">
    <xsl:element name="{local-name()}">
      <xsl:apply-templates select="@* | node()" />
    </xsl:element>
  </xsl:template>

```

```

</xsl:template>

<xsl:template match="b:Allowed">
  <xsl:element name="{local-name()}">
    <xsl:apply-templates select="@* | node()" />
  </xsl:element>
</xsl:template>

<xsl:template match="@ name | @qualifier | @type">
  <xsl:attribute name="{local-name()}">
    <xsl:value-of select="." />
  </xsl:attribute>
</xsl:template>

<xsl:template match="@ minOccurs | @maxOccurs">
</xsl:template>

<xsl:template match="b:ApplicationName">
</xsl:template>

<xsl:template match="b:DataStagers">
  <Inputs>
    <xsl:apply-templates
      select="b:DataStager[@type='input']" />
  </Inputs>
  <Outputs>
    <xsl:apply-templates
      select="b:DataStager[@type='output']" />
  </Outputs>
</xsl:template>

<xsl:template match="b:DataStager[@type='input']">
  <Input>
    <xsl:attribute name="name">
      <xsl:value-of select="@name" />
    </xsl:attribute>
    <xsl:attribute name="type">
      <xsl:choose>
        <xsl:when test="contains(b:MimeType, 'archive')">
          list
        </xsl:when>
        <xsl:otherwise>file</xsl:otherwise>
      </xsl:choose>
    </xsl:attribute>
    <xsl:attribute name="zip">
      <xsl:choose>
        <xsl:when test="contains(b:MimeType, 'archive')
          or contains(b:MimeType, 'zip')">
          true
        </xsl:when>
        <xsl:otherwise>>false</xsl:otherwise>
      </xsl:choose>
    </xsl:attribute>
  <xsl:if test="@ minOccurs and @maxOccurs">

```

118 Appendix B. GRIA Job To Abstract Template XSL Transformation

```
<xsl:attribute name="array">true</xsl:attribute>
<xsl:attribute name="index">0</xsl:attribute>
</xsl:if>
<MimeType>
  <xsl:choose>
    <xsl:when test="contains(b:MimeType, 'archive ')">
      <xsl:value-of
        select="substring-after(b:MimeType, 'archive ')" />
    </xsl:when>
    <xsl:when test="contains(b:MimeType, 'zip ')">
      <xsl:value-of
        select="substring-after(b:MimeType, 'zip ')" />
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="b:MimeType" />
    </xsl:otherwise>
  </xsl:choose>
</MimeType>
<xsl:apply-templates />
</Input>
</xsl:template>

<xsl:template match="b:DataStager [@type='output ']">
  <Output>
    <xsl:attribute name="name">
      <xsl:value-of select="@name" />
    </xsl:attribute>
    <xsl:attribute name="type">
      <xsl:choose>
        <xsl:when test="contains(b:MimeType, 'archive ')">
          list
        </xsl:when>
        <xsl:otherwise>file</xsl:otherwise>
      </xsl:choose>
    </xsl:attribute>
    <xsl:attribute name="zip">
      <xsl:choose>
        <xsl:when test="contains(b:MimeType, 'archive ')
          or contains(b:MimeType, 'zip ')">
          true
        </xsl:when>
        <xsl:otherwise>>false</xsl:otherwise>
      </xsl:choose>
    </xsl:attribute>
    <MimeType>
      <xsl:choose>
        <xsl:when test="contains(b:MimeType, 'archive ')">
          <xsl:value-of
            select="substring-after(b:MimeType, 'archive ')" />
        </xsl:when>
        <xsl:when test="contains(b:MimeType, 'zip ')">
          <xsl:value-of
            select="substring-after(b:MimeType, 'zip ')" />
        </xsl:when>

```

```
        <xsl:otherwise>
          <xsl:value-of select="b:MimeType" />
        </xsl:otherwise>
      </xsl:choose>
    </MimeType>
    <xsl:apply-templates />
  </Output>
</xsl:template>

<xsl:template match="b:MimeType">
</xsl:template>
</xsl:stylesheet>
```

List of Abbreviations

AGWL	Abstract Grid Workflow Language
B-IT	Bonn-Aachen International Center for Information Technology
BPEL	Business Process Execution Language
DAG	Directed Acyclic Graph
DEE	Distributed Enactment Engine
DMR	Data Management WS-Resource
EBI	European Bioinformatics Institute
EECS	Electrical Engineering and Computer Sciences
EML	Ecological Metadata Language
EPCC	El Paso Community College
ER	Execution WS-Resource
FITS	Flexible Image Transport System
GGF	Globus Grid Forum
GSi	Grid Security Infrastructure
HEFT	Heterogeneous Earliest Finish Time
IPAC	Infrared Processing and Analysis Center
IRSA	Infrared Science Archive
ISI	Information Science Institute
JSC	Jülich Supercomputing Centre
JSDL	Job Submission Description Language
LIGO	The Laser Interferometer Gravitational Wave Observatory
MDS	Monitoring and Discovery Service
MoML	Modeling Markup Language
MOWS	Management Of Web Services
MUWS	Management Using Web Services

NFT	Number of Failed Tasks
NMR	Node Management WS-Resource
NSET	Number of Successfully Executed Tasks
NSF	National Science Foundation
NSIS	Nullsoft Scriptable Install System
NVO	National Virtual Observatory
OGF	Open Grid Forum
OGSA	Open Grid Services Architecture
OGSA-DAI	Open Grid Service Architecture-Data Access and Integration
Pegasus	Planning for Execution in Grids
RCP	Rich Client Platform
RLS	Replica Location Service
SCUFL	Simple Conceptual Unified Flow Language
SEIS	Semantically Enriched Integration System
SERC	System Efficiency and Reliability Challenges
SR	Scheduler WS-Resource
SRB	Storage Resource Broker
SRF	Server Reliability Factor
SUC	System Usability Challenges
SWfMS	Scientific Workflow Management System
SWIMS	Scientific Workflow Integration and Management System
UCC	UNICORE command line client
UDDI	Universal Description Discovery and Integration
UML	Unified Modeling Language
Unicore	UNiform Interface to COmpute REsources
VO	Virtual Organization
WCS	World Coordinate System

WfMC	Workflow Management Coalition
WMS	Workflow Management System
WSCE	Web Service Crawler Engine
WSDL	Web Services Description Language
WSDM	Web Service Distributed Management
WSN	Web Services Notification
WSRF	WS-Resource Framework

Bibliography

- [Abramson 2005] D. Abramson, J. Kommineni and I. Altintas. *Flexible IO services in the Kepler Grid Workflow system*. In Proceedings of the International Conference on eScience and Grid Technology, 2005. 26
- [Al-Masri 2008] E. Al-Masri and Q. H. Mahmoud. *Investigating web services on the world wide web*. In Proceeding of the 17th international conference on World Wide Web, WWW '08, pages 795–804, New York, NY, USA, 2008. ACM. 40
- [Alda 2007] S. Alda, J. Kuck and A. B. Cremers. *Tailorability of personalized BPEL-based Workflow Compositions*. In Proceedings of the 2007 IEEE Congress on Services, pages 245–252, July 2007. 66, 84
- [Anjomshoaa 2005] A. Anjomshoaa, F. Brisard, M. Drescher, D. Fellows, A. Ly, S. Mcgough, D. Pulsipher and A. Savva. *Job Submission Description Language (JSDL) Specification, Version 1.0 - Technical Report*, 2005. 21
- [Antonioletti 2007] M. Antonioletti, N. P. Chue Hong, A. C. Hume, M. Jackson, K. Karasavvas, A. Krause, J. M. Schopf, M. P. Atkinson, B. Dobrzelecki, M. Illingworth, N. McDonnell, M. Parsons and E. Theocharopoulos. *OGSA-DAI 3.0 - The Whats and the Whys*. In UK e-Science All Hands Meeting, pages 158–165, 2007. 49
- [Apache 2007] Apache. *Muse 2.2.0*. <http://ws.apache.org/muse/>, March 2007. 48
- [Berriman 2004] G. B. Berriman, E. Deelman, J. C. Good, J. C. Jacob, D. S. Katz, C. Kesselman, A. C. Laity, T. A. Prince, G. Singh and M. Su. *Montage: a grid-enabled engine for delivering custom science-grade mosaics on demand*. Proceedings of SPIE, vol. 5493, pages 221–232, 2004. 11
- [Bhagwanani 2005] S. R. Bhagwanani. An evaluation of end-user interfaces of scientific workflow management systems. Master's thesis, North Carolina State University, 2005. 16
- [Bharathi 2008] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M. Su and K. Vahi. *Characterization of scientific workflows*. In Proceedings of the 3rd Workshop on Workflows in Support of Large-Scale Science, pages 1–10, 2008. 11, 56
- [Blythe 2005] J. Blythe, S. Jain, E. Deelman, Y. Gil, K. Vahi, A. Mandal and K. Kennedy. *Task scheduling strategies for workflow-based applications in grids*. In Proceedings of the 5th IEEE International Symposium on Cluster Computing and the Grid, volume 2 of *CCGRID '05*, pages 759–767, Washington, DC, USA, 2005. IEEE Computer Society. 22, 112

- [Braun 2008] U. Braun, A. Shinnar and M. Seltzer. *Securing Provenance*. In Proceedings of the 3rd USENIX Workshop on Hot Topics in Security, USENIX HotSec, pages 1–5, Berkeley, CA, USA, July 2008. USENIX Association. 62
- [Brooke 1992] A. Brooke, D. Kendrick and A. Meeraus. *GAMS: A User's Guide*. Release 2.25. The Scientific Press, 1992. 2
- [Brown 2007] D. A. Brown, P. R. Brady, A. Dietz, J. Cao, B. Johnson and J. McNabb. *A Case Study on the Use of Workflow Technologies for Scientific Analysis: Gravitational Wave Data Analysis*. In Ian J. Taylor, Ewa Deelman, Dennis B. Gannon and Matthew Shields, editors, *Workflows for e-Science*, pages 39–59. Springer London, 2007. 11
- [Carzaniga 2007] A. Carzaniga, G. P. Picco and G. Vigna. *Is Code Still Moving Around? Looking Back at a Decade of Code Mobility*. In Proceedings of the 29th International Conference on Software Engineering, ICSE COMPANION '07, pages 9–20, Washington, DC, USA, 2007. IEEE Computer Society. 64
- [Chervenak 2002] A. Chervenak, E. Deelman, I. Foster, L. Guy, W. Hoschek, A. Iamnitchi, C. Kesselman, P. Kunszt, M. Ripeanu, B. Schwartzkopf, H. Stockinger, K. Stockinger and B. Tierney. *Giggle: a framework for constructing scalable replica location services*. In Proceedings of the 2002 ACM/IEEE conference on Supercomputing, Supercomputing '02, pages 1–17, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press. 22
- [Cruz 2009] S. Cruz, M. Campos and M. Mattoso. *Towards a Taxonomy of Provenance in Scientific Workflow Management Systems*. In Proceedings of the 2009 Congress on Services - I, pages 259–266, Washington, DC, USA, 2009. IEEE Computer Society. 62
- [Deelman 2001] E. Deelman and C. Kesselman. *Transformation Catalog Design for GriPhyN - Technical Report*. http://www.isi.edu/deelman/griphyn_2001_17.pdf, 2001. 22
- [Deelman 2005] E. Deelman, G. Singh, M. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob and D. S. Katz. *Pegasus: A framework for mapping complex scientific workflows onto distributed systems*. *Sci. Program.*, vol. 13, pages 219–237, July 2005. 21
- [Deelman 2008] E. Deelman and A. Chervenak. *Data Management Challenges of Data-Intensive Scientific Workflows*. In Proceedings of the 2008 8th IEEE International Symposium on Cluster Computing and the Grid, pages 687–692, Washington, DC, USA, 2008. IEEE Computer Society. 16

- [Deelman 2009] E. Deelman, D. Gannon, M. Shields and I. Taylor. *Workflows and e-Science: An overview of workflow system features and capabilities*. Future Generation Computer Systems, vol. 25, no. 5, pages 528 – 540, 2009. 16
- [Duan 2005] R. Duan, R. Prodan and T. Fahringer. *DEE: A Distributed Fault Tolerant Workflow Enactment Engine for Grid Computing*. In Proceedings of the 2005 International Conference on High Performance Computing and Communications, HPCC-05, pages 704–716. Springer-Verlag, 2005. 22
- [El-Gayyar 2008] M. El-Gayyar, S. Alda and A. B. Cremers. *Towards a user-oriented environment for web services composition*. In Proceedings of the 4th international workshop on End-user software engineering, WEUSE '08, pages 81–85, New York, NY, USA, 2008. ACM. 66, 84
- [El-Gayyar 2009] M. El-Gayyar, Y. Leng and A. B. Cremers. *New Execution Paradigm for Data-Intensive Scientific Workflows*. In Proceedings on the 2009 World Conference on Services - I, pages 334 –339, 2009. 4, 32
- [El-Gayyar 2010] M. El-Gayyar, Y. Leng and A. B. Cremers. *Distributed Management of Scientific Workflows in SWIMS*. In Proceedings of the 2010 9th International Symposium on Distributed Computing and Applications to Business Engineering and Science (DCABES), pages 327 –331, August 2010. 4, 32
- [Equinox 2011] Equinox. *The Eclipse Equinox Project*. <http://www.eclipse.org/equinox/>, 2011. 75
- [Fahringer 2005a] T. Fahringer, R. Prodan, R. Duan, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H. Truong, A. Villazon and M. Wiczorek. *ASKALON: a Grid application development and computing environment*. In Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing, 2005., pages 676–685, 2005. 22
- [Fahringer 2005b] T. Fahringer, J. Qin and S. Hainzer. *Specification of Grid Workflow Applications with AGWL: An Abstract Grid Workflow Language*. In Proceedings of the 2005 IEEE International Symposium on Cluster Computing and the Grid, Cardiff, UK, 2005. IEEE Computer Society Press. 22
- [Fahringer 2007] T. Fahringer, R. Prodan, R. Duan, J. Hofer, F. Nadeem, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H. Truong, A. Villazon and M. Wiczorek. *ASKALON: A Development and Grid Computing Environment for Scientific Workflows*. In Workflows for e-Science, pages 450–471. Springer London, 2007. 22
- [Fahringer 2011] T. Fahringer. *ASKALON: Grid Application Development and Computing Environment*. <http://www.dps.uibk.ac.at/projects/askalon/>, 2011. 19

- [Fegraus 2005] E. Fegraus, S. J. Andelman, M. B. Jones and M. Schildhauer. *Maximizing the Value of Ecological Data with Structured Metadata: An Introduction to Ecological Metadata Language (EML) and Principles for Metadata Creation*. Bulletin of the Ecological Society of America, vol. 86, no. 3, pages 158–168, 2005. 20
- [Fitzgerald 2001] S. Fitzgerald. *Grid Information Services for Distributed Resource Sharing*. In Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing, pages 181–, Washington, DC, USA, 2001. IEEE Computer Society. 22
- [Foster 2001] I. Foster, C. Kesselman and S. Tuecke. *The Anatomy of the Grid: Enabling Scalable Virtual Organizations*. High Perform. Comput. App., vol. 15, pages 200–222, 2001. 8
- [Foster 2002] I. Foster, C. Kesselman, J. M. Nick and S. Tuecke. *The Physiology of the Grid An Open Grid Services Architecture for Distributed Systems Integration*. Globus Project, 2002. 8
- [Foster 2003] I. Foster and C. Kesselman. The grid 2: Blueprint for a new computing infrastructure. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003. 2, 8
- [Foster 2006a] I. Foster. *Globus Toolkit Version 4: Software for Service-Oriented Systems*. Journal of Computer Science and Technology, vol. 21, no. 4, pages 513–520, July 2006. 8
- [Foster 2006b] I. Foster, H. Kishimoto, A. Savva, D. Berry, A. Djaoui, A. Grimshaw, B. Horn, H. F. Siebenlist and R. Subramaniam. *The Open Grid Services Architecture, Version 1.5*. <http://www.ogf.org/documents/GFD.80.pdf>, 2006. 8
- [Foster 2008] I. Foster, Y. Zhao, I. Raicu and S. Lu. *Cloud Computing and Grid Computing 360-Degree Compared*. In Proceedings of the Grid Computing Environments Workshop, 2008. GCE '08, pages 1–10, 2008. 9
- [Freefluo 2009] Freefluo. *Workflow Enactor*. <http://freefluo.sourceforge.net/>, July 2009. 10, 20, 60
- [Frey 2002] J. Frey, T. Tannenbaum, M. Livny, I. Foster and S. Tuecke. *Condor-G: A Computation Management Agent for Multi-Institutional Grids*. Cluster Computing, vol. 5, pages 237–246, 2002. 22
- [Gil 2007] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau and J. Myers. *Examining the Challenges of Scientific Workflows*. Computer, vol. 40, no. 12, pages 24–32, 2007. 16
- [Globus 2011] Globus. *The Globus Toolkit*. <http://www.globus.org/toolkit/>, 2011. 23, 26, 28, 29

- [Graves 2011] R. Graves, T. H. Jordan, S. Callaghan, E. Deelman, E. Field, G. Juve, C. Kesselman, P. Maechling, G. Mehta, K. Milner, D. Okaya, P. Small and K. Vahi. *CyberShake: A Physics-Based Seismic Hazard Model for Southern California*. Pure and Applied Geophysics, vol. 168, pages 367–381, 2011. 11
- [GRIA 2009] GRIA. *A Service-Oriented Grid Middleware*. www.gria.org, 2009. iii, 5, 8, 9, 26
- [GridFTP 2011] GridFTP. *Universal Data Transfer for the Grid*. <http://www.globus.org/toolkit/data/gridftp/>, 2011. 20, 23
- [Hollingsworth 2004] D. Hollingsworth. *The Workflow Reference Model: 10 Years On*. In Fujitsu Services, UK; Technical Committee Chair of WfMC, pages 295–312, 2004. 13
- [Hull 2006] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. Pocock, P. Li and T. Oinn. *Taverna: a tool for building and running workflows of services*. Nucleic Acids Research, vol. 34, pages 729–732, 2006. 20
- [IRSA 2011] IRSA. *NASA/IPAC Infrared Science Archive*. <http://irsa.ipac.caltech.edu/>, 2011. 94
- [Jacob 2009] J. C. Jacob, D. S. Katz, G. B. Berriman, J. C. Good, A. C. Laity, E. Deelman, C. Kesselman, G. Singh, M. Su, T. A. Prince and R. Williams. *Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking*. Comput. Sci. Eng., vol. 4, pages 73–87, July 2009. 11
- [Katz 2005] D. S. Katz, J. C. Jacob, G. B. Berriman, J. Good, A. C. Laity, E. Deelman, C. Kesselman and G. Singh. *A Comparison of Two Methods for Building Astronomical Image Mosaics on a Grid*. In Proceedings of the 2005 International Conference on Parallel Processing - Workshops, ICPPW '05, pages 85–94, Washington, DC, USA, 2005. IEEE Computer Society. 11
- [Kepler 2011] Kepler. *Workflow Management System*. <https://kepler-project.org/>, 2011. 19
- [Kreger 2005] H. Kreger. *A little wisdom about WSDM*. <http://www.ibm.com/developerworks/webservices/library/ws-wisdom/>, March 2005. 47
- [Lee 2000] E. A. Lee and S. Neuendorffer. *MOML - A Modeling Markup Language in XML-Version 0.4*. <http://www.eecs.berkeley.edu/Pubs/TechRpts/2000/3818.html>, 2000. 20
- [Lee 2008] K. Lee, N. W. Paton, R. Sakellariou, E. Deelman, A. Fernandes and G. Mehta. *Adaptive Workflow Processing and Execution in Pegasus*. In Proceedings of the 3rd International Conference on Grid and Pervasive

- Computing - Workshops, pages 99–106, Washington, DC, USA, 2008. IEEE Computer Society. 21
- [Leng 2009] Y. Leng, M. El-Gayyar and S. Shumilov. *Semantically Enriched Integration System for Heterogeneous Web Services*. In Proceedings of the IADIS International Conference on WWW/Internet, 2009. 4, 33, 37, 61
- [Leng 2010] Y. Leng, M. El-Gayyar and A. B. Cremers. *Semantics Enhanced Composition Planner for Distributed Resources*. In Proceedings of the 2010 9th International Symposium on Distributed Computing and Applications to Business Engineering and Science (DCABES), pages 61–65, August 2010. 37
- [Lin 2009] C. Lin, S. Lu, X. Fei, A. Chebotko, D. Pai, Z. Lai, F. Fotouhi and J. Hua. *A Reference Architecture for Scientific Workflow Management Systems and the VIEW SOA Solution*. IEEE Trans. Serv. Comput., vol. 2, pages 79–92, January 2009. iii, iv, 13, 14, 89
- [Livny 2008] J. Livny, H. Teonadi, M. Livny and M. K. Waldor. *High-Throughput, Kingdom-Wide Prediction and Annotation of Bacterial Non-Coding RNAs*. PLoS ONE, vol. 3, no. 9, 2008. 11
- [Ludaescher 2006] B. Ludaescher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao and Y. Zhao. *Scientific workflow management and the Kepler system: Research Articles*. Concurr. Comput. : Pract. Exper., vol. 18, pages 1039–1065, August 2006. 1, 20
- [Markgraefe 2010] F. Markgraefe. Index webservice fuer ein grid. Master's thesis, Rheinische Friedrich Wilhelms Universitaet Bonn, 2010. 40
- [Mcaffer 2005] J. Mcaffer and J. Lemieux. Eclipse rich client platform: Designing, coding, and packaging java(tm) applications. Addison-Wesley Professional, 2005. 75
- [Meier 2010] W. Meier. *eXist-db: Open Source Native XML Database*. <http://exist.sourceforge.net/>, 2010. 49
- [Microsoft 2011] Microsoft. *Registry Functions (Windows)*. [http://msdn.microsoft.com/en\\$-\\$us/library/](http://msdn.microsoft.com/en$-$us/library/), February 2011. 65
- [Miller 2010] S. P. Miller. *Dependency Walker Version 2.2*. <http://www.dependencywalker.com/>, August 2010. 65
- [Missier 2010] P. Missier, S. Soiland-Reyes, S. Owen, W. Tan, A. Nenadic, I. Dunlop, A. Williams, T. Oinn and C. Goble. *Taverna, Reloaded*. In Scientific and Statistical Database Management, volume 6187 of *Lecture Notes in Computer Science*, chapitre 33, pages 471–481. Springer Berlin Heidelberg, 2010. 20, 24

- [MM5 2003] MM5. *The 5th generation NCAR / Penn State Mesoscale Model*. <http://www.mmm.ucar.edu/mm5/>, January 2003. 2
- [Montage 2011] Montage. *An Astronomical Image Mosaic Engine*. <http://montage.ipac.caltech.edu/applications.html>, 2011. 5, 11, 96
- [Moreau 2005] L. Moreau, Y. Zhao, I. Foster, J. Voeckler and M. Wilde. *XDTM: The XML Data Type and Mapping for Specifying Datasets*. In *Advances in Grid Computing*, volume 3470 of *EGC 2005*, pages 495–505. Springer Berlin Heidelberg, 2005. 29
- [Morgan 2005] M. Morgan. *ByteIO Specification Version 1.0*. http://www.gridforum.org/Public_Comment_Docs/Documents/Feb-2006/draft-byteio-rec-doc-v1-1.pdf, 2005. 21
- [Mouallem 2010] P. Mouallem, D. Crawl, I. Altintas, M. Vouk and U. Yildiz. *A Fault-Tolerance Architecture for Kepler-Based Distributed Scientific Workflows*. In *Scientific and Statistical Database Management*, volume 6187 of *Lecture Notes in Computer Science*, pages 452–460. Springer Berlin Heidelberg, 2010. 21
- [MOWS 2006] MOWS. *Management Of Web Services Version 1.1*. <http://www.oasis-open.org/committees/download.php/20574/wsdm-mows-1.1-spec-os-01.pdf>, August 2006. 47
- [MUWS 2006a] MUWS. *Management Using Web Services Version 1.1, Part I*. <http://www.oasis-open.org/committees/download.php/20576/wsdm-muws1-1.1-spec-os-01.pdf>, August 2006. 47
- [MUWS 2006b] MUWS. *Management Using Web Services Version 1.1, Part II*. <http://www.oasis-open.org/committees/download.php/20575/wsdm-muws2-1.1-spec-os-01.pdf>, August 2006. 47
- [myExperiment 2011] myExperiment. *The myExperiment Project*. <http://www.myexperiment.org/>, 2011. 26
- [Novotny 2001] J. Novotny, S. Tuecke and V. Welch. *An Online Credential Repository for the Grid: MyProxy*. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing*, pages 104–, Washington, DC, USA, 2001. IEEE Computer Society. 28, 29
- [NSIS 2011] NSIS. *Nullsoft Scriptable Install System*. <http://nsis.sourceforge.net/>, February 2011. 65
- [OASIS 2005] OASIS. *X.509 Certificate Token Profile Version 1.1*. <http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-os-x509TokenProfile.pdf>, June 2005. 27, 42
- [OGF 2006] OGF. *Open Grid Forum*. <http://www.ogf.org/>, 2006. 8

- [OGSA-DAI 2010] OGSA-DAI. *An innovative solution for distributed data access and management*. <http://www.ogsadai.org.uk/>, 2010. 49
- [Oinn 2006] T. Oinn, M. Greenwood, M. Addis, N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M. Pocock, M. Senger, R. Stevens, A. Wipat and C. Wroe. *Taverna: lessons in creating a workflow environment for the life sciences*. *Concurr. Comput. : Pract. Exper.*, vol. 18, no. 10, pages 1067–1100, August 2006. 20
- [OSGi 2011] OSGi. *The Dynamic Module System for Java*. <http://www.osgi.org/>, 2011. 76
- [Pegasus 2011] Pegasus. *Workflow Management System*. <http://pegasus.isi.edu/>, 2011. 19
- [Ptolemy 2004] Ptolemy. *PtolemyII: project and system*. <http://ptolemy.eecs.berkeley.edu/ptolemyII/>, 2004. 20
- [Qin 2008] J. Qin and T. Fahringer. *A novel domain oriented approach for scientific grid workflow composition*. In Proceedings of the 2008 ACM/IEEE conference on Supercomputing, SC '08, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press. 23
- [Radetzki 2004] U. Radetzki and A. B. Cremers. *IRIS: a framework for mediator-based composition of service-oriented software*. In Proceedings of the 2004 IEEE International Conference on Web Services, pages 752 – 756, July 2004. 37
- [Radetzki 2006] U. Radetzki and A. B. Cremers. *Automatic Discovery and Composition of Services with IRIS*. In Proceedings of the 22nd International Conference on Data Engineering, page 39, 2006. 37
- [Raicu 2007] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster and M. Wilde. *Falkon: a Fast and Light-weight task executiON framework*. In Proceedings of the 2007 ACM/IEEE conference on Supercomputing, SC '07, New York, NY, USA, 2007. ACM. 23
- [Rambadt 2008] M. Rambadt, A. Vanni and R. Niederberger. *Integration of GridFTP as an Alternative File Transfer in UNICORE for the DEISA Infrastructure*. In Proceedings of the 2008 4th IEEE International Conference on eScience, pages 516–523, Washington, DC, USA, 2008. IEEE Computer Society. 21, 22
- [RCP 2010] RCP. *Eclipse Rich Client Platform*. http://wiki.eclipse.org/index.php/Rich_Client_Platform, December 2010. 75
- [Russinovich 2011] M. Russinovich and B. Cogswell. *Process Monitor v2.94*. <http://technet.microsoft.com/en-au/sysinternals/bb896645>, January 2011. 65

- [Scherp 2010] G. Scherp, A. Hoeing, S. Gudenkauf, W. Hasselbring and O. Kao. *Using UNICORE and WS-BPEL for scientific workflow execution in grid environments*. In Proceedings of the 2009 international conference on Parallel processing, Euro-Par'09, pages 335–344, Berlin, Heidelberg, 2010. Springer-Verlag. 21
- [Schulla 1999] J. Schulla and K. Jasper. *Model description WaSiM-ETH*. IAC ETH Zuerich, page 166, 1999. 2
- [SHA-2 Standard 2002] SHA-2 Standard. *National Institute of Standards and Technology (NSIT), Secured Hash Standard, FIPS PUB 180-2*, 2002. 51, 70
- [Shumilov 2006] S. Shumilov, T. Erdenberger, A. B. Cremers, L. Bharati, M. Plotnikova and C. Rodger. *First Steps Towards an Integrated Decision Support System for Water Management*. In Arno Scharl (Eds.) Klaus Tochtermann, editeur, Shaker Verlag, pages 327–334. Shaker Verlag, 2006. ISBN 978-3-8322-5321-9. 1
- [Shumilov 2008] S. Shumilov, Y. Leng, M. El-Gayyar and A. B. Cremers. *Distributed Scientific Workflow Management for Data-Intensive Applications*. In Proceedings of the 2008 12th IEEE International Workshop on Future Trends of Distributed Computing Systems, pages 65–73, Washington, DC, USA, 2008. IEEE Computer Society. 4, 32
- [Siddiqui 2005] M. Siddiqui and T. Fahringer. *GridARM: Askalon's Grid Resource Management System*. In Advances in Grid Computing - EGC 2005, pages 122–131, 2005. 22
- [Singh 2008] G. Singh, M. Su, K. Vahi, E. Deelman, B. Berriman, J. Good, D. S. Katz and G. Mehta. *Workflow task clustering for best effort systems with Pegasus*. In Proceedings of the 15th ACM Mardi Gras conference, MG '08, pages 9:1–9:8, New York, NY, USA, 2008. ACM. 67
- [Sotomayor 2006] B. Sotomayor and L. Childers. *Globus toolkit 4: Programming java services*. The Elsevier Series in Grid Computing. Morgan Kaufmann, 2006. 8
- [SRB 2011] SRB. *The DICE Storage Resource Broker*. http://www.sdsc.edu/srb/index.php/Main_Page, 2011. 20
- [Staples 2006] G. Staples. *TORQUE resource manager*. In Proceedings of the 2006 ACM/IEEE conference on Supercomputing, SC '06, New York, NY, USA, 2006. ACM. 10
- [Streit 2010] A. Streit, P. Bala, A. Beck-Ratzka, K. Benedyczak, S. Bergmann, R. Breu, J. Daivandy, B. Demuth, A. Eifer, A. Giesler, B. Hagemeier, S. Holl, V. Huber, N. Lamla, D. Mallmann, A. Memon, M. Memon,

- M. Rambadt, M. Riedel, M. Romberg, B. Schuller, T. Schlauch, A. Schreiber, T. Soddemann and W. Ziegler. *UNICORE 6 - Recent and Future Advancements*. Annals of Telecommunications, vol. 65, pages 757–762, 2010. 8, 21
- [SurrIDGE 2005] M. SurrIDGE, S. Taylor, D. De Roure and E. Zaluska. *Experiences with GRIA: Industrial Applications on a Web Services Grid*. In Proceedings of the 1st International Conference on e-Science and Grid Computing, pages 98–105, Washington, DC, USA, 2005. IEEE Computer Society. 8, 9
- [Swift 2011] Swift. *Workflow Management System*. <http://www.ci.uchicago.edu/swift/main/>, 2011. 19
- [Taverna 2011] Taverna. *Workflow Management System*. <http://www.taverna.org.uk/>, 2011. 10, 19
- [Taylor 2006] I. J. Taylor, E. Deelman, D. B. Gannon and M. Shields. *Workflows for e-science: Scientific workflows for grids*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. 11
- [Thain 2005] D. Thain, T. Tannenbaum and M. Livny. *Distributed computing in practice: the Condor experience*. *Concurr. Comput. : Pract. Exper.*, vol. 17, pages 323–356, 2005. 10, 22
- [Topcuoglu 2002] H. Topcuoglu, S. Hariri and M. Wu. *Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing*. *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, pages 260–274, March 2002. 22
- [Tsalgatidou 2006] A. Tsalgatidou, G. Athanasopoulos, M. Pantazoglou, C. Pautasso, T. Heinis, R. Groenmo, Hjoerdis Hoff, Arne-Joergen Berre, M. Glittum and S. Topouzidou. *Developing scientific workflows from heterogeneous services*. *SIGMOD Rec.*, vol. 35, pages 22–28, June 2006. 11, 13
- [UDDI 2004] UDDI. *Universal Description Discovery and Integration Version 3.0.2*. http://uddi.org/pubs/uddi_v3.htm, October 2004. 40
- [UML 2011] UML. *UML: Unified Modeling Language Version 2.0*. <http://www.uml.org/>, 2011. iii, iv, 28, 53, 61, 90
- [UNICORE 2011] UNICORE. *Uniform Interface to Computing Resources*. <http://www.unicore.eu/>, 2011. 19
- [USC 2011] USC. *USC Epigenome Center*. <http://epigenome.usc.edu/>, 2011. 11
- [Vetter 1996] J. Vetter and K. Schwan. *Models for Computational Steering*. In Proceedings of the 3rd International Conference on Configurable Distributed Systems, Washington, DC, USA, 1996. IEEE Computer Society. 60

- [von Laszewski 2005] G. von Laszewski and M. Hategan. *Workflow Concepts of the Java CoG Kit*. *Journal of Grid Computing*, vol. 3, pages 239–258, 2005. 23
- [Wang 2008] J. Wang, I. Altintas, C. Berkley, L. Gilbert and M. B. Jones. *A High-Level Distributed Execution Framework for Scientific Workflows*. In *Proceedings of the 2008 4th IEEE International Conference on eScience*, pages 634–639, Washington, DC, USA, 2008. IEEE Computer Society. 20
- [Wang 2009] J. Wang, I. Altintas, P. R. Hosseini, D. Barseghian, D. Crawl, C. Berkley and M. B. Jones. *Accelerating Parameter Sweep Workflows by Utilizing Ad-hoc Network Computing Resources: An Ecological Example*. In *Proceedings of the 2009 World Conference on Services - I*, pages 267–274, 2009. 20
- [Wieczorek 2005] M. Wieczorek, R. Prodan and T. Fahringer. *Scheduling of scientific workflows in the ASKALON grid environment*. *SIGMOD Rec.*, vol. 34, pages 56–62, September 2005. 22, 112
- [Wilde 2011] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz and I. Foster. *Swift: A language for distributed parallel scripting*. *Parallel Computing*, vol. 37, no. 9, pages 633–652, 2011. 29
- [WS-Addressing 2006] WS-Addressing. *Web Services Addressing Version 1.0*. <http://www.w3.org/TR/ws-addr-core/>, May 2006. 48
- [WSDL 2001] WSDL. *Web Services Description Language Version 1.1*. <http://www.w3.org/TR/wsdl>, March 2001. 42
- [WSDM 2006] WSDM. *Web Services Distributed Management Version 1.1*. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsdm, August 2006. 46
- [WSI 2006] WSI. *Basic Profile Version 1.1*. <http://www.ws-i.org/profiles/basicprofile-1.1.html>, April 2006. 48
- [WSI 2007] WSI. *Basic Security Profile Version 1.0*. <http://www.ws-i.org/profiles/basicsecurityprofile-1.0.html>, March 2007. 42
- [WSN 2006] WSN. *Web Services Notification Version 1.3*. http://docs.oasis-open.org/wsn/wsn-ws_base_notification-1.3-spec-os.pdf, October 2006. 44
- [WSRF 2006] WSRF. *Web Services Resource Framework Version 1.2*. <http://docs.oasis-open.org/wsrp/wsrp-primer-1.2-primer-cd-02.pdf>, May 2006. 48
- [XPath 1999] XPath. *XML Path Language Version 1.0*. <http://www.w3.org/TR/xpath/>, November 1999. 50

- [XQuery 2010] XQuery. *An XML Query Language Version 1.0*. <http://www.w3.org/TR/xquery/>, December 2010. 63
- [XScufl 2004] XScufl. *Workflow Language Specifications*. <http://www.ebi.ac.uk/~tmo/mygrid/XScuflSpecification.html>, April 2004. 52, 60
- [XSLT 1999] XSLT. *XSL Transformations Version 1.0*. <http://www.w3.org/TR/xslt>, November 1999. 42
- [Yu 2004] J. Yu and R. Buyya. *A Novel Architecture for Realizing Grid Workflow using Tuple Spaces*. In Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, GRID '04, pages 119–128, Washington, DC, USA, 2004. IEEE Computer Society. 2
- [Yu 2005] J. Yu and R. Buyya. *A taxonomy of scientific workflow systems for grid computing*. SIGMOD Rec., vol. 34, pages 44–49, September 2005. 2, 13, 56
- [ZEF 2011] ZEF. *GLOWA Volta Project*. <http://www.Glowa-Volta.de>, 2011. 1
- [Zhao 2007] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun and M. Wilde. *Swift: Fast, Reliable, Loosely Coupled Parallel Computation*. In Proceedings of the 2007 IEEE Congress on Services, pages 199–206, July 2007. 23
- [Zhao 2008a] Y. Zhao, I. Raicu and I. Foster. *Scientific Workflow Systems for 21st Century, New Bottle or New Wine?* In Proceedings of the 2008 IEEE Congress on Services - I, SERVICES '08, pages 467–471, Washington, DC, USA, 2008. IEEE Computer Society. 16
- [Zhao 2008b] Y. Zhao, I. Raicu, I. Foster, M. Hategan, V. Nefedova and M. Wilde. *Realizing Fast, Scalable and Reliable Scientific Computations in Grid Environments*. CoRR, vol. abs/0808.3548, 2008. 23