Ian J. Taylor
Ewa Deelman
Dennis B. Gannon
Matthew Shields (Eds)

Workflows for e-Science

Scientific Workflows for Grids

Springer

Workflows for e-Science

Scientific Workflows for Grids

Editors: Ian J. Taylor, Ewa Deelman,
Dennis Gannon and Matthew S. Shields

To my fan base: Peter, Caroline, Nicholas,
Teresa, Wojtek, Adam and Alicja — Ewa

For Ruth whose support keeps me going, and my father Martin,
who would have enjoyed seeing this book — Matthew

To Adina and the making of ART — Ian

Foreword

This collection of articles on 'Workflows for e-Science' is very timely and important. Increasingly, to attack the next generation of scientific problems, multidisciplinary and distributed teams of scientists need to collaborate to make progress on these new 'Grand Challenges'. Scientists now need to access and exploit computational resources and databases that are geographically distributed through the use of high speed networks. 'Virtual Organizations' or 'VOs' must be established that span multiple administrative domains and/or institutions and which can provide appropriate authentication and authorization services and access controls to collaborating members. Some of these VOs may only have a fleeting existence but the lifetime of others may run into many years. The Grid community is attempting to develop both standards and middleware to enable both scientists and industry to build such VOs routinely and robustly.

This, of course, has been the goal of research in distributed computing for many years; but now these technologies come with a new twist service orientation. By specifying resources in terms of a service description, rather than allowing direct access to the resources, the IT industry believes that such an approach results in the construction of more robust distributed systems. The industry has therefore united around web services as the standard technology to implement such service oriented architectures and to ensure interoperability between different vendor systems.

The Grid community is also now uniting in developing 'Web Service Grids' based on an underlying web service infrastructure. In addition to the security services of VOs, scientists require services that allow them to run jobs on remote computers and to access and query databases remotely. As these data analysis operations become more and more complex and repetitive, there is a need to capture and coordinate the orchestrated operations that access the resources of a VO or Grid.

Scientific workflows have therefore emerged and been adapted from the business world as a means to formalize and structure the data analysis and computations on the distributed resources. Such scientific workflows in fact

now encapsulate scientific intellectual property and enable the sharing of knowledge between researchers.

This is the first book to provide a comprehensive survey of the present state of the art and include descriptions of all the major scientific workflow systems. From these accounts it is clear that there is much overlap in the functionality of the different systems and it is to be hoped that this collection will be a first step on the road to the consolidation of key workflow services. As such this book may well be a landmark collection heralding a step change in the level of abstraction for scientific workflows.

Tony Hey

16th May 2006

Contents

Part I Application and User Perspective

List of Contributors

Martin Alt
Westfälische Wilhelms–Universität
    Münster
Institut für Informatik
Einsteinstr. 62
D-48149 Münster, Germany
`mnalt@uni-muenster.de`

Roger Barga
Microsoft Research
One Microsoft Way
Redmond, WA 98052, USA
`barga@microsoft.com`

G. Bruce Berriman
Infrared Processing and Analysis
    Center
California Institute of Technology
Pasadena, CA 91125, USA
`gbb@ipac.caltech.edu`

Shawn Bowers
UC Davis Genome Center
Department of Computer Science
University of California
Davis, CA 95616, USA
`sbowers@ucdavis.edu`

Patrick R. Brady
Department of Physics
University of Wisconsin–Milwaukee

P.O. Box 413
Milwaukee, WI 53201, USA
`patrick@gravity.phys.uwm.edu`

John Brooke
Manchester Computing
The University of Manchester
Oxford Road
Manchester, M13 9PL, UK
`j.m.brooke@manchester.ac.uk`

Duncan A. Brown
LIGO Laboratory
California Institute of Technology
Pasadena, CA 91125, USA
`dbrown@ligo.caltech.edu`

Ben Butchart
Software Systems Engineering Group
Department of Computer Science
University College London
Gower Street
London, WC1E 6BT, UK
`B.Butchart@cs.ucl.ac.uk`

Scott Callaghan
Southern California Earthquake
    Center
University of Southern California
Los Angeles, CA 90089, USA
`scottcal@usc.edu`

Nick Cameron
Software Systems Engineering Group
Department of Computer Science
University College London
Gower Street
London, WC1E 6BT, UK
N.Cameron@cs.ucl.ac.uk

Junwei Cao
LIGO Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139, USA
jcao@ligo.mit.edu

Paul Carr
Jodrell Bank Observatory
The University of Manchester
Macclesfield
Cheshire SK11 9DL, UK
pcarr@jb.man.ac.uk

Liang Chen
Software Systems Engineering Group
Department of Computer Science
University College London
Gower Street
London, WC1E 6BT, UK
L.Chen@cs.ucl.ac.uk

Jeremy Cohen
London e-Science Centre
Department of Computing
Imperial College
London SW7 2AZ, UK
jhc02@doc.ic.ac.uk

Peter Couvares
University of Wisconsin–Madison
Computer Sciences Department
1210 West Dayton Street
Madison, WI 53706–1685, USA
pfc@cs.wisc.edu

Yifeng Cui
San Diego Supercomputing Center
La Jolla, CA 92093, USA
yfcui@sdsc.edu

John Darlington
London e-Science Centre
Department of Computing
Imperial College
London SW7 2AZ, UK
jd@doc.ic.ac.uk

Ewa Deelman
Information Sciences Institute
University of Southern California
Marina Del Rey, CA 90292, USA
deelman@isi.edu

Alexander Dietz
Department of Physics
Louisiana State University
Baton Rouge, LA 70803, USA
dietz@phys.lsu.edu

Rubing Duan
Institute of Computer Science
University of Innsbruck
Technickerstraße 21a
A-6020 Innsbruck, Austria
rubing@dps.uibk.ac.at

Mark H. Ellisman
National Center for Microscopy and
    Imaging Research
University of California, San Diego
9500 Gilman Drive, BSB 1000
La Jolla, CA 92093-0608, USA
mark@ncmir.ucsd.edu

Wolfgang Emmerich
Software Systems Engineering Group
Department of Computer Science
University College London
Gower Street
London, WC1E 6BT, UK
W.Emmerich@cs.ucl.ac.uk

Thomas Fahringer
Institute of Computer Science
University of Innsbruck
Technickerstraße 21a
A-6020 Innsbruck, Austria
`tf@dps.uibk.ac.at`

Zoltan Farkas
MTA SZTAKI
H-1132 Budapest
Victor Hugo 18-22, Hungary
`zfarkas@sztaki.hu`

Edward Field
US Geological Survey
Pasadena, CA 91106, USA
`field@caltech.edu`

Ian Foster
Computation Institute and
    Department of Computer Science
University of Chicago
Chicago, IL 60637, USA
and
Mathematics and Computer Science
    Division
Argonne National Laboratory
Argonne, IL 60439, USA
`foster@mcs.anl.gov`

Hunter Francoeur
Southern California Earthquake
    Center
University of Southern California
Los Angeles, CA 90089, USA
`francoeu@usc.edu`

Dennis Gannon
Department of Computer Science
Indiana University
Bloomington, IN 47405, USA
`gannon@cs.indiana.edu`

Yolanda Gil
Information Sciences Institute
University of Southern California
Marina Del Rey, CA 90292, USA
`gil@isi.edu`

Tristan Glatard
CNRS, I3S Laboratory
BP121, 06903 Sophia Antipolis
France
`glatard@i3s.unice.fr`

Carole Goble
School of Computer Science
University of Manchester
Manchester M13 9PL, UK
`carole@cs.man.ac.uk`

Antoon Goderis
School of Computer Science
University of Manchester
Manchester M13 9PL, UK
`goderisa@cs.man.ac.uk`

John Good
Infrared Processing and Analysis
    Center
California Institute of Technology
Pasadena, CA 91125, USA
`jcg@ipac.caltech.edu`

Tom Goodale
School of Computer Science
Cardiff University
Queen's Buildings, The Parade
Cardiff CF24 3AA, UK
and
Center for Computation and
    Technology
Louisiana State University
Baton Rouge, LA 70803, USA
`t.r.goodale@cs.cardiff.ac.uk`

Robert Graves
URS Corporation
Pasadena, CA 91101, USA
`robert_graves@urscorp.com`

Mark Greenwood
School of Computer Science
University of Manchester
Manchester M13 9PL, UK
markg@cs.man.ac.uk

Jeffrey S. Grethe
National Center for Microscopy and
    Imaging Research
University of California, San Diego
9500 Gilman Drive, BSB 1000
La Jolla, CA 92093-0608, USA
jgrethe@ncmir.ucsd.edu

Nitin Gupta
Southern California Earthquake
    Center
University of Southern California
Los Angeles, CA 90089, USA
niting@usc.edu

Vipin Gupta
Southern California Earthquake
    Center
University of Southern California
Los Angeles, CA 90089, USA
vgupta@usc.edu

Andrew Harrison
School of Computer Science
Cardiff University
Queen's Buildings, The Parade
Cardiff CF24 3AA, UK
a.harrison@cs.cardiff.ac.uk

Mihael Hategan
University of Chicago
Research Institute, Suite 405
South Ellis Avenue
Chicago, IL 60637, USA
hategan@mcs.anl.gov

Tony Hey
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052, USA
tonyhey@microsoft.com

Dan Higgins
National Center for Ecological
    Analysis and Synthesis (NCEAS)
University of California, Santa
    Barbara
Santa Barbara, CA 93101, USA
higgins@nceas.ucsb.edu

Jürgen Hofer
Institute of Computer Science
University of Innsbruck
Technickerstraße 21a
A-6020 Innsbruck, Austria
juergen@dps.uibk.ac.at

Andreas Hoheisel
Fraunhofer FIRST
Kekulestr. 7
D-12489 Berlin, Germany
andreas.hoheisel@
    first.fraunhofer.de

Duncan Hull
School of Computer Science
University of Manchester
Manchester M13 9PL, UK
duncan.hull@cs.man.ac.uk

Joseph C. Jacob
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109, USA
Joseph.C.Jacob@jpl.nasa.gov

Ben Johnson
LIGO Hanford Observatory
Richland, WA 99352, USA
bjohnson@ligo-wa.caltech.edu

Andrew C. Jones
School of Computer Science
Cardiff University
Queen's Buildings, The Parade
Cardiff CF24 3AA, UK
Andrew.C.Jones@cs.cardiff.ac.uk

Matthew B. Jones
National Center for Ecological
    Analysis and Synthesis (NCEAS)
University of California, Santa
    Barbara
Santa Barbara, CA 93101, USA
jones@nceas.ucsb.edu

Thomas Jordan
Southern California Earthquake
    Center
University of Southern California
Los Angeles, CA 90089, USA
tjordan@usc.edu

Peter Kacsuk
MTA SZTAKI
H-1132 Budapest
Victor Hugo 18-22, Hungary
kacsuk@sztaki.hu

Gopi Kandaswamy
Renaissance Computing Institute
University of North Carolina at
    Chapel Hill
100 Europa Drive Suite 540,
Chapel Hill, NC 27715, USA
gopi@renci.org

Eleftheria Katsiri
London e-Science Centre
Department of Computing
Imperial College
London SW7 2AZ, UK
ek@doc.ic.ac.uk

Daniel S. Katz
Louisiana State University and Jet
    Propulsion Laboratory
California Institute of Technology
Baton Rouge, LA 70803, USA
d.katz@ieee.org

Douglas B. Kell
Bioanalytical Sciences
School of Chemistry
University of Manchester
Manchester M13 9PL, UK
dbk@manchester.ac.uk

Carl Kesselman
Information Sciences Institute
University of Southern California
Marina Del Rey, CA 90292, USA
carl@isi.edu

Deepti Kodeboyina
University of Chicago
Research Institute, Suite 405
South Ellis Avenue
Chicago, IL 60637, USA
dkodeboy@mcs.anl.gov

Tevfik Kosar
Louisiana State University
Department of Computer Science
and
Center for Computation & Techno-
    logy
Baton Rouge, LA 70803, USA
kosar@cct.lsu.edu

Michael Kramer
Jodrell Bank Observatory
The University of Manchester
Macclesfield
Cheshire SK11 9DL, UK
mkramer@jb.man.ac.uk

Anastasia C. Laity
Infrared Processing and Analysis
    Center
California Institute of Technology
Pasadena, CA 91125, USA
laity@ipac.caltech.edu

Gregor von Laszewski
Argonne National Laboratory
Argonne, IL 60430, USA
and
University of Chicago
Research Institute, Suite 405
5640 South Ellis Avenue
Chicago, IL 60637, USA
gregor@mcs.anl.gov

William Lee
London e-Science Centre
Department of Computing
Imperial College
London SW7 2AZ, UK
wwhl@doc.ic.ac.uk

Peter Li
Bioanalytical Sciences
School of Chemistry
University of Manchester
Manchester M13 9PL, UK
Peter.Li@manchester.ac.uk

Abel W. Lin
National Center for Microscopy and
    Imaging Research
University of California, San Diego
9500 Gilman Drive, BSB 1000
La Jolla, CA 92093-0608, USA
awlin@ncmir.ucsd.edu

Bertram Ludäscher
UC Davis Genome Center
Department of Computer Science
University of California
Davis, CA 95616, USA
ludaesch@ucdavis.edu

Philip Maechling
Southern California Earthquake
    Center
University of Southern California
Los Angeles, CA 90089, USA
maechlin@usc.edu

Suresh Marru
Department of Computer Science
Indiana University
Bloomington, IN, USA
smarru@cs.indiana.edu

Andrew Stephen McGough
London e-Science Centre
Department of Computing
Imperial College
London SW7 2AZ, UK
asm@doc.ic.ac.uk

John McNabb
The Pennsylvania State University
University Park, PA 16802, USA
mcnabb@gravity.psu.edu

John Mehringer
Southern California Earthquake
    Center
University of Southern California
Los Angeles, CA 90089, USA
jmehring@usc.edu

Gaurang Mehta
Information Sciences Institute
University of Southern California
Marina Del Rey, CA 90292, USA
gmehta@isi.edu

Johan Montagnat
CNRS, I3S Laboratory
BP121, 06903 Sophia Antipolis
France
johan@i3s.unice.fr

Farrukh Nadeem
Institute of Computer Science
University of Innsbruck
Technickerstraße 21a
A-6020 Innsbruck, Austria
farrukh@dps.uibk.ac.at

Francesco Nerieri
Institute of Computer Science
University of Innsbruck
Technickerstraße 21a
A-6020 Innsbruck, Austria
nero@dps.uibk.ac.at

Tom Oinn
EMBL European Bioinformatics
    Institute
Hinxton, Cambridge CB10 1SD, UK
tmo@ebi.ac.uk

David Okaya
Southern California Earthquake
    Center
University of Southern California
Los Angeles, CA 90089, USA
okaya@usc.edu

Savas Parastatidis
School of Computing Science
University of Newcastle upon Tyne
Newcastle upon Tyne, NE1 7RU,
    UK
savas@parastatidis.name

Jignesh Patel
Software Systems Engineering Group
Department of Computer Science
University College London
Gower Street
London, WC1E 6BT, UK
J.Patel@cs.ucl.ac.uk

Steven T. Peltier
National Center for Microscopy and
    Imaging Research
University of California, San Diego
9500 Gilman Drive, BSB 1000
La Jolla, CA 92093-0608, USA
peltier@ncmir.ucsd.edu

Deana D. Pennington
Long Term Ecological Research
    Network (LTER) Office
University of New Mexico
Albuquerque, NM, 87131, USA
dpennington@LTERnet.edu

A. Townsend Peterson
Natural History Museum and
    Biodiversity Research Center
University of Kansas
Lawrence, KS 66045, USA
town@ku.edu

Stephen Pickles
Manchester Computing
The University of Manchester
Oxford Road
Manchester, M13 9PL, UK
stephen.pickles@manchester.ac.uk

Beth Plale
Department of Computer Science
Indiana University
Bloomington, IN, USA
plale@cs.indiana.edu

Stefan Podlipnig
Institute of Computer Science
University of Innsbruck
Technickerstraße 21a
A-6020 Innsbruck, Austria
spodlipn@dps.uibk.ac.at

Thomas A. Prince
Division of Physics, Mathematics,
    and Astronomy,
California Institute of Technology
Pasadena, CA 91125, USA
prince@caltech.edu

Radu Prodan
Institute of Computer Science
University of Innsbruck
Technickerstraße 21a
A-6020 Innsbruck, Austria
radu@dps.uibk.ac.at

Jun Qin
Institute of Computer Science
University of Innsbruck
Technickerstraße 21a
A-6020 Innsbruck, Austria
jerry@dps.uibk.ac.at

Alain Roy
University of Wisconsin–Madison
Computer Sciences Department
1210 West Dayton Street
Madison, WI 53706–1685, USA
roy@cs.wisc.edu

Matthew Shields
School of Physics and Astronomy
Cardiff University
Queens Buildings, The Parade
Cardiff CF24 3AA, UK
m.s.shields@cs.cardiff.ac.uk

Satoshi Shirasuna
Department of Computer Science
Indiana University
Bloomington, IN, USA
sshirasu@cs.indiana.edu

Mumtaz Siddiqui
Institute of Computer Science
University of Innsbruck
Technickerstraße 21a
A-6020 Innsbruck, Austria
mumtaz@dps.uibk.ac.at

Yogesh Simmhan
Department of Computer Science
Indiana University
Bloomington, IN, USA
ysimmhan@cs.indiana.edu

Gurmeet Singh
Information Sciences Institute
University of Southern California
Marina Del Rey, CA 90292, USA
gurmeet@isi.edu

Gergely Sipos
MTA SZTAKI
H-1132 Budapest
Victor Hugo 18-22, Hungary
sipos@sztaki.hu

Aleksander Slominski
Department of Computer Science
Indiana University
Bloomington, IN 47405, USA
aslom@cs.indiana.edu

Robert Stevens
School of Computer Science
University of Manchester
Manchester M13 9PL, UK
robert.stevens@manchester.ac.uk

Mei-Hui Su
Information Sciences Institute
University of Southern California
Marina Del Rey, CA 90292, USA
mei@isi.edu

Ian Taylor
School of Computer Science
Cardiff University
Queen's Buildings, The Parade
Cardiff CF24 3AA, UK
and
Center for Computation and
    Technology
Louisiana State University
Baton Rouge, LA 70803, USA
Ian.J.Taylor@cs.cardiff.ac.uk

Hong-Linh Truong
Institute of Computer Science
University of Innsbruck
Technickerstraße 21a
A-6020 Innsbruck, Austria
truong@dps.uibk.ac.at

Daniele Turi
School of Computer Science
University of Manchester
Manchester M13 9PL, UK
dturi@cs.man.ac.uk

Karan Vahi
Information Sciences Institute
University of Southern California
Marina Del Rey, CA 90292, USA
vahi@isi.edu

Alex Villazon
Institute of Computer Science
University of Innsbruck
Technickerstraße 21a
A-6020 Innsbruck, Austria
avt@dps.uibk.ac.at

Ian Wang
School of Physics and Astronomy
Cardiff University
Queen's Buildings, The Parade
Cardiff CF24 3AA, UK
`i.n.wang@cs.cardiff.ac.uk`

Bruno Wassermann
Software Systems Engineering Group
Department of Computer Science
University College London
Gower Street
London, WC1E 6BT, UK
`B.Wassermann@cs.ucl.ac.uk`

Jim Webber
Thoughtworks
Level 7
16 O'Connell Street
Sydney, NSW 2000
Australia
`jim@webber.name`

Jeff Weber
University of Wisconsin–Madison
Computer Sciences Department
1210 West Dayton Street
Madison, WI 53706–1685, USA
`weber@cs.wisc.edu`

R. Kent Wenger
University of Wisconsin–Madison
Computer Sciences Department
1210 West Dayton Street
Madison, WI 53706–1685, USA
`wenger@cs.wisc.edu`

Marek Wieczorek
Institute of Computer Science
University of Innsbruck
Technickerstraße 21a
A-6020 Innsbruck, Austria
`marek@dps.uibk.ac.at`

Michael Wilde
Computation Institute
University of Chicago
Chicago, IL 60637, USA
and
Mathematics and Computer Science
    Division
Argonne National Laboratory
Argonne, IL 60439, USA
`wilde@mcs.anl.gov`

Simon Woodman
School of Computing Science
University of Newcastle upon Tyne
Newcastle upon Tyne, NE1 7RU,
    UK
`s.j.woodman@ncl.ac.uk`

Jun Zhao
School of Computer Science
University of Manchester
Manchester M13 9PL, UK
`zhaoj@cs.man.ac.uk`

Li Zhao
Southern California Earthquake
    Center
University of Southern California
Los Angeles, CA 90089, USA
`zhaol@usc.edu`

Yong Zhao
Computation Institute and
    Department of Computer Science
University of Chicago
Chicago, IL 60637, USA
`yongzh@cs.uchicago.edu`

1

Introduction

Dennis Gannon, Ewa Deelman, Matthew Shields, and Ian Taylor

Workflows for e-Science is divided into four parts, which represent four broad but distinct areas of scientific workflows. In the first part, Background, we introduce the concept of scientific workflows and set the scene by describing how they differ from their business workflow counterpart. In Part II, Application and User Perspective, we provide a number of scientific examples that currently use workflows for their e-Science experiments. In Workflow Representation and Common Structure (Part III), we describe core workflow themes, such as control flow or dataflow and the use of components or services. In this part, we also provide overviews for a number of common workflow languages, such as Petri Nets, the Business Process Execution Language (BPEL), and the Virtual Data Language (VDL), along with service interfaces. In Part IV, Frameworks and Tools, we take a look at many of the popular environments that are currently being used for e-Science applications by paying particular attention to their workflow capabilities. The following four sections describe the chapters in each part and therefore provide a comprehensive summary of the book as a whole.

1.1 Background

Over the past 25 years, we have seen a revolution in the way science and engineering has been conducted. Specifically, computation became an established third branch of science alongside theory and experiment. The first phase of this change came with the use of supercomputers to simulate large, physically complex systems modeled by partial differential equations. The adoption of these computational tools soon led to other applications that involved complex data analysis and visualization steps. The task of moving data to a supercomputer for analysis or simulation and then managing the storage of the output results was often repeated many times, and it was left to the persistence and creativity of the user to make sure things were done correctly. At the same time, the business community was also faced with the problem of automating

their business processing steps and the computer industry began supplying tools to help them. What emerged was a primitive science of workflow design. Within the business world, *workflow orchestration* is a term that refers to the activity of defining the sequence of tasks needed to manage a business or computational science or engineering process. A *workflow* is a template for such an orchestration. A *workflow instance* is a specific instantiation of a workflow for a particular problem. Within the scientific and engineering community these terms have a slightly broader meaning, which we will discuss below once we have set more historical context.

The earliest workflows in both business and science were encoded and managed by complex job-control language and shell scripts that were used to stage input data to the computer and then move output results to tapes or workstations. Frequently these scripts involved substantial preprocessing to bring data into a form for the analysis tools and postprocessing to put them into a form appropriate for human understanding. The scripting approach became more sophisticated as the processes became more demanding. However, two additional major changes in the computing landscape drove a fundamental shift in the evolution of workflow technology.

The second major change in computing came with the use of computational resources that were distributed over a network. Simple scripts could not control the execution and coordination of task execution on machines elsewhere on the network. This job required distributed computing technology to solve problem such as synchronization between remote concurrent tasks, fault recovery, distributed logging, and remote data management. To deal with this problem, workflow systems had to evolve beyond simple scripting into systems built around remote procedure calls, distributed object technology, and distributed file systems and databases. These approaches to distributed systems have now evolved into Grid technology and Web-service-oriented architectures. Workflow tools that operate in this domain are described extensively throughout this book.

The third change that has influenced the way the scientific community has approached workflow is use of component-based tools to program large systems. Some of this work evolved from Petri Net models, while other work came from dataflow concepts. As a model for workflow, it was first seen in early visualization tools, such as AVS [266]. In computer graphics and visualization, where it is not uncommon for a single rendering job to require many separate transformation steps to produce the final image, a dataflow model provides an excellent means to automate the schedule of tasks. These tools provided a compositional programming model based on a graphical layout tool where tasks are boxes and arrows between boxes indicate the control and data dependencies between them. This programming metaphor has proven to be extremely popular and is a common component of most scientific workflow systems described here.

We conclude this chapter with an overview of the contents of the remainder of this volume.

1.2 Application and User Perspective

As science today becomes more complex and relies on the analysis of large-scale data sets, it is becoming necessary to manage the data and the data processing in an automated and scalable way. Workflows have recently emerged as a way of formalizing and structuring the data analysis in a way that makes it easy to define the analysis, execute the necessary computations on distributed resources, collect information about the derived data products, and if necessary repeat the analysis. Worfklows also enable the definition and sharing of the analysis definitions within scientific collaborations. In the Application and User Perspective section of this book, we have compiled a set of cutting-edge applications that rely on workflow technologies to advance the state of the art in a variety of fields from astronomy, gravitational wave science, ecology, meteorology, earthquake science, and neuroscience.

Chapter 3 describes the use of workflow technologies in generating large-scale image mosaics of the sky. The authors describe how the workflows describing the process of mosaic generation can be used in a variety of applications depending on the data sources used in the mosaic construction. The chapter also describes the technologies used in managing the workflows, such as Pegasus (Chapter 23) and DAGMan (Chapter 22) and contrast them with implementations based on the Message Passing Interface (MPI) standard.

Two chapters (Chapter 4 and Chapter 5) deal with issues of supporting gravitational wave science using workflow technologies. Chapter 4 focuses on providing scientists with tools that allow for easy workflow construction and leveraging workflow management tools to schedule the workflows in Grid environments. Chapter 5 focuses on issues of obtaining good overall workflow performance by optimizing critical workflow portions.

There are also two chapters (Chapter 6 and Chapter 7) that address issues of providing ecologists with a means of easily defining complex workflows. The authors of both chapters recognize the need to provide an interface that enables the users to describe the workflows using high-level, scientifically meaningful concepts without exposing details of the workflow management and execution. Chapter 6 discusses the use of Triana (Chapter 20) in workflow design and management, whereas the authors of Chapter 7 use the Kepler [19] system to provide that functionality.

Neuroscientists impose similar requirements on the worfklow tools, requiring ease of use and operation at high levels of abstraction. The authors of Chapter 8 describe how portals can be used to provide custom interfaces for a broad community of users. Behind the portal, they use technologies such as Pegasus and Condor to manage the workflows.

Chapter 9 describes how workflows are used in simulations of the weather events such as tornadoes and hurricanes. The chapter addresses issues of workflow adaptivity, where the analysis adapts to the changes in the physical environment (in this case the weather), to the simulation results, and to the changes in the computational environment.

Workflows have also been a useful tool for earthquake scientists who need to analyze Terabytes of data in an automated manner. Chapter 10 describes how workflow technologies can be used to manage the large-scale computations with hundreds of thousands of individual tasks and leverage a number of distributed resources.

The applications described in Part I rely on a variety of workflow technologies, such as Kepler, Triana, Pegasus, Condor, and others, some of which are described in the Frameworks Part (Part III).

1.3 Workflow Representation and Common Structure

In this Part, we examine some of the common elements and ideas that occur in scientific workflow languages and environments. Although the tools and frameworks described in this book are all very different, there are often concepts and techniques that get repeated or reused. Business workflow methods have been in use for far longer than scientific workflows, and consequently many of the ideas have migrated from the business domain to the scientific domain. In some cases, BPEL (Chapter 14), the business domain workflow language, is being used directly for scientific workflows. In others, it is merely concepts such as dependencies, data or not, that are borrowed from the earlier field. This chapter compares some very different formalisms for workflow representation from the fairly typical *graphs* through *Petri Nets* (Chapter 13) to π-calculus and the Soap Service Description Language (SSDL) (Chapter 15). It also includes a chapter on the use of *semantics* in scientific workflows (Chapter 16) and the use of a *virtual data language* (Chapter 17) to separate physical representations from logical typing.

The argument of control flow versus dataflow representations is outlined in Chapter 11. Control flow, with its history in scripting languages, and dataflow, with its history in the data-processing arenas of image and signal processing, are both widely used within the tools and frameworks described in this book.

Chapter 12 considers the impact of reusable software components and component architectures on scientific workflows as we move from object-oriented component systems to service-based workflows. There are several different representations for workflows: Many of the tools in this book use graph representations, typically either *directed acyclic graphs* (DAGs) or *directed cyclic graphs* (DCGs) depending upon whether or not loop connections are allowed.

Petri Nets are a formalism for describing distributed processes by extending state machines with concurrency. Chapter 13 covers a brief introduction to Petri Net theory and then explains how this can be applied to the choreography, orchestration, and enactment of scientific workflows. Issues such as synchronization, persistence, transactional safety, and fault management are examined within this workflow formalism.

BPEL is a well-known leading workflow language for composing business domain Web services. In Chapter 14 the author examines how the language

meets the needs for a scientific workflow language in a Grid environment. Some of the dynamic aspects of scientific workflows that are not common in business workflows are used to show how BPEL can be adapted to this use.

Chapter 15 describes SSDL, an interesting approach to workflow representation based upon the "service" and "message" abstraction. Workflows are described using the interaction of Simple Object Access Protocol (SOAP) messages between services, from simple request-response to multiservice exchanges. One of the SSDL protocols, the *Sequential Constraints* protocol, is introduced, which can be used to describe multiservice, multimessage exchanges between Web services using notations based upon the π-calculus. The formal model basis for this protocol allows the authors to make assertions about certain properties of the composed workflows.

Semantics is the study of meaning. In Chapter 16, the author explains how semantic representations can be used to automate and assist in workflow composition and to manage complex scientific processes. The chapter discusses separating levels of abstraction in workflow descriptions, using semantic representations of workflows and their components, and supporting flexible automation through reuse and automatic completion of user specifications for partial workflows.

The final chapter in this Part, Chapter 17, also covers the use of abstraction in workflow representations. The tasks of describing, composing, and executing workflows are often complicated by heterogeneous storage formats and ad hoc file system structures. The authors show how these difficulties can be overcome via a typed, compositional *virtual data language* (VDL), where issues of physical representation are cleanly separated from logical typing. Logical types are represented as Extensible Markup Language (XML) schema, and the relationship between logical and physical types is specified as type-specific mapping operations, with workflows defined as compositions of calls to logically typed programs or services.

1.4 Frameworks and Tools: Workflow Generation, Refinement and Execution

The general theme of this Part is *workflow generation, refinement, and execution*, which reflects the broad stages of how workflows are represented and converted into an executable format, and how such workflows are executed through the use of an execution engine or enactment subsystem. The various frameworks within this section take different approaches to these stages and, furthermore, these terms mean different things to different frameworks. For example, in the Virtual Data System (see Chapter 23), refinement might involve using their *Virtual Data Catalog* to transform the requested files into workflows that can be used to generate them. This process involves modifying the workflow by inserting subworkflows that generate the various data dependencies. In contrast, however, refinement within the Triana workflow

system (see Chapter 20) generally involves dynamic switching at runtime of the Grid services that are used to execute the specific parts of the workflow. Triana uses the Grid Application Toolkit (GAT) interface, which can switch between different low-level Grid functionalities. This results in *refinements* being made based on the current execution environment. These themes therefore reflect a"look and feel" for the chapters so that each framework can organize its content with a format familiar to the reader. Each chapter therefore is part of a series rather than a disconnected set of papers, which as editors we tried hard to avoid.

In Chapter 18, the authors distinguish between two different techniques for managing job submissions: task-based and service-based. They argue that in complex control flows for considering data and computationally intensive scientific applications, these techniques exhibit significant differences for representing data flows, parametric input data, and efficient exploitation of the distributed infrastructures. They introduce a service-based workflow manager called MOTEUR, and discuss its integration with both the P-GRADE portal and DAGMan workflow manager, and show how these can represent and execute parametric data intensive applications.

The Taverna workbench discussed in Chapter 19 was developed for ^{my}Grid to support *in silico* experiments in biology and to provide scientists with user-friendly access to underlying services that they wish to use. Taverna is based on Web services and uses the ^{my}Grid Simple Conceptual Unified Flow Language (SCUFL) for workflow choreography. Taverna enables users to construct, share, and enact workflows using a customized fault-tolerant enactment engine for execution.

Triana (Chapter 20) is a graphical workflow environment that consists of a simple drag-and-drop style Graphical User Interface (GUI) for composing applications and an underlying subsystem for workflow enactment across P2P, service-based, and Grid environments. Components can be grouped to create aggregate or compound components (called *Group Units* in Triana) for simplifying the visualization of complex workflows and groups can contain groups for recursive representation of the workflow. Triana employs the use of two generic interfaces, called the Grid Application Prototype (GAP) and GAT, which can interact with services or Grid tools, respectively, for interaction with JXTA, P2PS, Web services, WS-RF services, or Grid tools like the Globus Resource Allocation Manager (GRAM), Grid File Transfer Protocol (GridFTP), and Grid Resource Management and Brokering Service (GRMS), etc. The authors discuss these bindings and provide use cases showing how the various stages are accomplished.

The Java CoG Kit, discussed in Chapter 21, focuses on workflow solutions in the Karajan workflow framework. Karajan can specify workflows using XML, and can support hierarchical workflows based on DAGs with control structures and parallel constructs. Workflows can be visualized and tracked through an engine and modified at runtime through interaction with a workflow repository or schedulers for dynamic association of resources to tasks.

Karajan has been demonstrated to scale to hundreds of thousands of jobs due to its efficient scalability-oriented threading mechanisms.

Condor (Chapter 22) began in 1988 and focused on reliable access to computing over long periods of time instead of highly tuned, high-performance computing over short periods. This chapter discuss two components: DAGMan, for submission and management of complex workflows; and Stork, a batch scheduler for data placement. Job dependencies are specified as arbitrary directed acyclic graphs (DAGs), and DAGMan supports a rich array of features, including pre- and postscripting, throttling, fault tolerance, and can scale up to 100,000 nodes. Stork implements techniques for queuing, scheduling, and the optimization of data placement, and supports a number of data transport protocols (FTP, GridFTP, HTTP, and DiskRouter) and data storage systems (SRB, UniTree, NeST, dCache, and CASTOR).

Pegasus (Chapter 23) can map large-scale workflows onto Grid resources and along with VDL (see Chapter 17) forms part of the Virtual Data System (VDS) released with the Virtual Data Toolkit. Pegasus supports a wide range of functionality, including catalog interfacing, workflow reduction, resource selection (based on the available resources, characteristics, and location of data), task clustering (to cluster jobs at the same resource), executable staging (at the remote site), pre- and poststaging and interfacing with an execution subsystem's workflow languages, (e.g., DAG for DAGMan). For execution, Pegasus supports failure recovery, optimization of workflow performance, and debugging capabilities, and it has been used in scientific domains ranging from bioinformatics to high-energy physics.

The Imperial College e-Science Networked Infrastructure (ICENI) system (Chapter 24) is a service-based software architecture to enable end-to-end, high-level workflow processing in a heterogeneous Grid environment. The authors distinguish between an *e-Scientists conceptual workflow* to describe tasks to be performed with dependencies and a *middleware workflow* for execution on the Grid. The architecture of ICENI supports deployment, performance, reliability, and charging for resource use. The current ICENI architecture is derived from previous work and experiences with e-Science projects, such as the Grid Enabled Integrated Earth system model (GENIE), e-Protein, and RealityGrid, which are described in this chapter.

Cactus, discussed in Chapter 25, is a framework designed for tightly coupled, high-performance simulations. This chapter provides a brief introduction to the framework and its component model, with an emphasis on the workflow aspects, and provides some illustrative examples. The chapter then examines current and future work to use Cactus for high-throughput distributed simulations and the use of Cactus within other component architectures.

The Sedna environment in Chapter 26 works on BPEL, which being standardized has strong industrial support, and many tools and middleware exist. However, being primarily targeted at business workflows, it does not necessarily provide abstractions that are suitable for use in scientific workflows. Sedna creates domain-independent as well as domain-specific language abstractions

that are more suitable for use by application scientists, while achieving compliance with the standard BPEL specification. ASKALON (27), on the other hand, supports workflow composition and modeling using the Unified Modeling Language (UML) standard and provides an XML-based Abstract Grid Workflow Language (AGWL) for application developers to use. The AGWL is given to a WSRF-based runtime system for scheduling and execution. ASKALON contains a resource manager (GridARM) that provides resource discovery, advanced reservation and virtual organization-wide authorization along with a dynamic registration framework for activity types and activity deployments.

2

Scientific versus Business Workflows

Roger Barga and Dennis Gannon

The formal concept of a workflow has existed in the business world for a long time. An entire industry of tools and technology devoted to workflow management has been developed and marketed to meet the needs of commercial enterprises. The Workflow Management Coalition (WfMC) has existed for over ten years and has developed a large set of reference models, documents, and standards. Why has the scientific community not adopted these existing standards? While it is not uncommon for the scientific community to reinvent technology rather than purchase existing solutions, there are issues involved in the technical applications that are unique to science, and we will attempt to characterize some of these here. There are, however, many core concepts that have been developed in the business workflow community that directly relate to science, and we will outline them below.

In 1996, the WfMC defined workflow as "the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules." [478] While this definition predates the currently evolving models of workflow based on service oriented architectures, it does provide a window on the original workflow concepts, which are based on Business Process Management (BPM). The book, *"Production Workflows: Concepts and Techniques"* by Leymann and Roller [255] provides an excellent overview of the entire field. A business process is an instance of any well-defined task that is often repeated as part of a standard enterprise task. For example, it may be the steps required to complete a purchase order, or it may be related to internal business tasks such as internal audits or corporate database management. Those parts of a business process that relate to the computer automation of business processes are the domain of workflow management.

Leyman and Roller [255] characterize four basic types of workflows encountered in business, and most have direct counterparts in science and engineering. They define collaborative workflows as those that have high business value to the company and involve a single large project and possibly many individuals. For example, the production, promotion, documentation, and re-

lease of a major product fall into this category. The workflow is usually specific to the particular project, but it may follow a standard pattern used by the company. Within the engineering disciplines, this corresponds to the tracking of tasks and subsystem integration required to design and release a new microprocessor. Within the scientific community, it can refer to the management of data produced and distributed on behalf of a large scientific experiment such as those encountered in high-energy physics. Another example may be the end-to-end tracking of the steps required by a biotech enterprise to produce and release a new drug.

The second type of workflow they describe is ad hoc. These activities are less formal in both structure and required response; for example, a notification that a business practice or policy has changed that is broadcast to the entire workforce. Any required action is up to the individual receiving the notification. Within science, notification-driven workflows are common. A good example is an agent process that looks at the output of an instrument. Based on events detected by the instrument, different actions may be required and subworkflow instances may need to be created to deal with them. The third type of workflow is administrative, which refers to enterprise activities such as internal bookkeeping, database management, and maintenance scheduling, that must be done frequently but are not tied directly to the core business of the company. On the other hand, the fourth type of workflow, referred to as production workflow, is involved with those business processes that define core business activities. For example, the steps involved with loan processing are one of the central business processes of a bank. These are tasks that are repeated frequently, and many such workflows may be concurrently processed. Both the administrative and production forms of workflow have obvious counterparts in science and engineering. For example, the routine tasks of managing data coming from instrument streams or verifying that critical monitoring services are running are administrative in nature. Production workflows are those that are run as standard data analyses and simulations by users on a daily basis. For example, doing a severe storm prediction based on current weather conditions within a specific domain or conducting a standard datamining experiment on a new, large data sample are all central to e-Science workflow practice.

There are however, areas where business workflows seem, at first glance, to be substantially different from their scientific counterparts. For example, a central concern about business workflows is the security and integrity of a sequence of actions. Paying customers demand that when they pay for a service, that service must be guaranteed complete and the results exactly as advertised. Customers demand service. They do not conduct experiments that may or may not succeed. This concept of the integrity of a sequence of actions in embodied in the concept of transaction and is central to understanding workflows in business.

An import class of transactions are those that are long running. Among these long running transactions are those that satisfy the ACID test. An

ACID transaction represents a logical unit of work that is composed of a set of operations. It is an activity that is completed in its entirety or not at all. ACID is an acronym where

- A stands for atomicity, which is an "all or nothing" execution guarantee
- C refers to the fact that the database is always in a *consistent* state
- I means the actions of each transaction are *isolated*, i.e. they are not seen and do not effect other operations that are not part of the transaction
- D is for durability. Once a transaction completes, its effect will survive even if the entire system crashes

The important point of an ACID transaction is that if some subtask fails, the entire transaction can be rolled back so that the entire state of the world is as it was prior to the start of the transaction. And the effect of the transaction is not visible until all subtasks have completed and the entire set of operations is committed. The application of this concept is clear. It is essential that any workflow that carries out the terms of a contract shall either complete the contract or the entire activity is aborted, and that fact is clear to all parties. For example, customers of a bank want to know when they have transferred funds from one account to another that the money was not lost along the way.

Unfortunately, not every workflow can be characterized as an ACID transaction. A long-running workflow is one that may involve many subworkflows each of which is an ACID transaction, but it may not be possible to completely rollback the entire workflow with a single rollback operation. Parts of the workflow may have already completed, and the state of the world may have been altered in various ways. In this case, a failure is something that requires a sequence of new workflows that involve compensating transactions. A typical example of a long-running workflow may involve multiple businesses engaged in a long-running collaboration to produce a product. One company may have been contracted to supply parts to another company producing the final product. The specific details of the interaction with the subcontractor may be governed by one subworkflow. But suppose the subcontractor is unable to deliver the goods. A compensating subworkflow may be to void the original contract and search for a secondary supplier and engage in a negotiation for a replacement service.

Both ACID and long-running workflows have their counterparts in e-Science. The concept of the ACID workflow is essential for any activity that involves building and maintaining a database. and increasingly databases are becoming an essential tool for scientific data analysis. Databases store our collective knowledge in areas such as biological and chemical informatics. Any workflow that could potentially corrupt such a database is one that must be ACID in nature. Long-running workflows also play a role in scientific workflows. A scientist may divide up the overall task into smaller subtasks, each of which can be considered an individual step in the experiment. The results obtained from each such step are either analyzed and/or stored for dissemination to other sites or individuals, used as an input to the next step in an

experiment or exploration, or both. If the scientist later decides an experiment step was faulty, he or she can compensate the subtask, possibly deleting the result and notifying others. Such a together chaining smaller tasks to achieve a desired result from an experiment or exploration, using various data and analysis services, is easily captured as a long-running transaction.

The business workflow industry has had to deal with the increasing complexity of the business processes that have come about because of the distributed nature of enterprises. The corporate information technology landscape has become very heterogeneous. This is a result of many factors, including corporate mergers and piecemeal software and hardware upgrades to different divisions of the company. In addition, there is an increasing need to improve efficiency across the entire organization, and this implies different parts of the organization must work in close alignment. The corporate workflows have to become more corporation-wide.

To address these problems, the workflow industry has been aggressive in its pursuit of technology that improves the time to completion of a workflow design process, reliability of the result, and interoperability across a wide range of platforms. Object-oriented technology has been widely adopted within the industry, and distributed object systems such as the Common Object Request Broker Architecture (CORBA) were a major step forward. The concept of programming by scripting the composition of software components is central to many workflow tools. Leymann and Roller note that to be used as an effective workflow tool, scripts must obey a strict set of rules. For example, it must be possible to interrupt a script at any point and resume its execution later. This implies that the script's state must be saved in a persistent store. Likewise, scripts must be recoverable. If something goes wrong, we should be able to stop the script and roll back any ACID subworkflows and replay the script from a point prior to the failure. It is assumed that the script is orchestrating remotely deployed and executing components and that these components may run in parallel if there are no dependencies preventing it. An important property of any component system is that the implementation technology of the individual components is not exposed. The only thing the script and other components see are interfaces. Leyman and Roller observe that the exploitation of components requires data flow facilities; for example, the input parameters of a component are constructed from the output of several preceding components.

Businesses are also under competitive pressure to rapidly integrate existing applications and business processes to react to changing business conditions. Process integration has always been a challenge and is only complicated further by the fact that business processes today often span multiple internal and external systems. Historically, custom integration solutions have addressed point-to-point integration, in which integration comes at a great cost. The most recent response to the integration challenge is service-oriented architectures (SOAs) [135] and Web service technologies. The promise of SOA is that application components can be assembled with little effort into a network of

loosely coupled services to create a business process that spans organizations and computing platforms. SOA is supported by a range of emerging standards that make it possible to define, implement, and deliver a service in a uniform way so it can be reused in different contexts. The dominant set of standards are those known as WS-*. Included in this set of standards are the Web Service Description Language (WSDL) for service description), the Universal Description, Discovery and Integration (UDDI) protocol for service discovery, the Simple Object Access Protocol (SOAP) for service communication, and the Web Service Business Process Execution Language (WS-BPEL) for workflow.

The essence of SOA lies in independent services that are interconnected with messaging. Each service is a self-contained chunk of code and data that is private to that service, and can be described, published, discovered, orchestrated, and deployed across networks such as the Internet. Services communicate with each other exclusively through messages. No knowledge of the partner service is shared other than the message formats and the sequences of messages that are expected. The bottom-up view of the SOA is that different applications expose their functionalities through Web services. Thus, programmers can access different functionalities of different legacy and newly developed applications in a standard way through Web services.

However, Web services by themselves do not address the need to compose and coordinate a process. WS-BPEL, or BPEL for short, is the de facto standard for the combination and orchestration of Web services. Orchestration, and therefore BPEL, enables a user to specify how existing services should be chained together in various ways to design an executable workflow. The new workflow can then be presented as a new service, which is why BPEL is often described as a language for recursive composition.

BPEL offers a rich language for orchestrating both business and scientific workflows. A BPEL process specifies the exact order in which participating services should be invoked. This can be done sequentially or in parallel. A programmer can express conditional behavior; for example, a Web service invocation can depend on the value of a previous invocation. One can also construct loops, declare variables, copy and assign values, define fault handlers, and so on. By combining all these constructs, the programmer can define a complex scientific experiment in an algorithmic manner. BPEL also provides support for both ACID and long running transactions. Most BPEL implementations can cause the state of a process instance to persist, allowing a user to interrupt a running workflow and reactivate it later when necessary. Moreover, workflows specified in BPEL are fully executable and portable across BPEL-conformant environments, which is an important step toward workflow reuse and exchange.

Today, scientists face many of the same challenges found in enterprise computing, namely integrating distributed and heterogeneous resources. Scientists no longer use just a single machine, or even a single cluster of machines, or a single source of data. Research collaborations are becoming more and more

geographically dispersed and often exploit heterogeneous tools, compare data from different sources, and use machines distributed across several institutions throughout the world. And as the number of scientific resources available on the Internet increases, scientists will increasingly rely on Web technology to perform *in silico* experiments. However, the task of running and coordinating a scientific application across several administrative domains remains extremely complex.

One reason BPEL is an attractive candidate for orchestrating scientific workflows is its strong support for Web services. With scientific resources now available as Web and Grid services, scientists can transition from copying and pasting data through a sequence of Web pages offering those resources to the creation and use of a workflow for experiment design, data analysis, and discovery. Many types of *in silico* genomics analyses, such as promoter identification, start with an initial set of data, perhaps acquired in a more mechanical way such as through fast sequencing equipment or from a microarray chip. This is followed by an ordered sequence of database queries, data transformations, and complex functional, statistical, and other analyses. Such work may require computing power ranging from a desktop computer to a remote supercomputer but is relatively loosely coupled and in many instances asynchronous. By defining a workflow to automatically invoke and analyze more routine parts of the process, multiple data sets can be processed in parallel without requiring a significant amount of additional effort from the scientist and can considerably increase productivity. With the proper tools, scientists with limited programming skills can use BPEL to construct a workflow that carries out an experiment or that retrieves data from remote data services.

There are other advantages to be gained from adapting BPEL for scientific workflows. Since BPEL workflows are designed to act as a Web service, a workflow can be published as a Web service and easily combined with other Web services. Capturing an *in silico* experiment or data transformation as a reusable workflow that can be defined, published, and easily reused is essential in sharing scientific best practice.

Using BPEL to orchestrate an experiment also enables fault tolerance. Because scientists are allowed to select and employ services from a UDDI registry into the workflow, they also have the ability to use an alternative service with similar functionality from the registry in case the original service fails. This ensures that no experiment terminates unexpectedly because of the failure of one particular service in the flow.

Furthermore, a BPEL workflow is specified in terms of service invocations. This allows all aspects of the workflow, such as service execution, message flow, data and process management, fault handling, etc., to be specified as a single integrated process rather than handled separately. The result is a workflow in which each step is explicit, no longer buried in Java or C code. Since the workflow is described in a unified manner, it is much easier to comprehend, providing the opportunity to verify or modify an experiment.

There is a clear case for the role of workflow technology in e-Science; however, there are technical issues unique to science. Business workflows are typically less dynamic and evolving in nature. Scientific workflows tend to change more frequently and may involve very voluminous data translations. In addition, while business workflows tend to be constructed by professional software and business flow engineers, scientific workflows are often constructed by scientists themselves. While they are experts in their domains, they are not necessarily experts in information technology, the software, or the networking in which the tools and workflows operate. Therefore, the two cases may require considerably different interfaces and end-user robustness both during the construction stage of the workflows and during their execution.

In composing a workflow, scientists often incorporate portions of existing workflows, making changes where necessary. Business workflow systems do not currently provide support for storing workflows in a repository and then later searching this repository during workflow composition.

The degree of flexibility that scientists have in their work is usually much higher than in the business domain, where business processes are usually pre-defined and executed in a routine fashion. Scientific research is exploratory in nature. Scientists carry out experiments, often in a trial-and-error manner wherein they modify the steps of the task to be performed as the experiment proceeds. A scientist may decide to filter a data set coming from a measuring device. Even if such filtering was not originally planned, that is a perfectly acceptable option. The ability to run, pause, revise, and resume a workflow is not exposed in most business workflow systems.

Finally, the control flow found in business workflows may not be expressive enough for highly concurrent workflows and data pipelines found in leading-edge simulation studies. Current BPEL implementations, and indeed most business workflow languages, require the programmer to enumerate all concurrent flows. Scientific workflows may require a new control flow operator to succinctly capture concurrent execution and data flow.

Over the last 20 years, there has been a great deal of interest in both research and industry in systematically defining, reasoning about, and enacting processes and workflows. With so many driving forces at work, it is clear that workflow systems are here to stay and will have a major role to play in the future IT strategies of business and scientific organizations, both large and small. The current focus is on the use of Web services and a move toward a new paradigm of service oriented architecture in which many loosely-coupled Web services are composed and coordinated to carry out a process, and orchestrated using an execution language such as BPEL.

It is genuinely hard to build a robust and scalable orchestration engine and associated authoring tools, and few groups have succeeded in doing so. The emergence of BPEL as the de facto industry standard for Web service orchestration is significant because it means that a number of commercial-grade BPEL engines will be readily available.

The strength of BPEL for orchestrating scientific workflows is its strong support for seamless access to remote resources through Web services. As scientific applications and curated data collections are published as Web services, as will increasingly be the case with the emergence of service-based Grid infrastructures, commercial BPEL engines will be an attractive execution environment for scientific workflows.

Part I

Application and User Perspective

3

Generating Complex Astronomy Workflows

G. Bruce Berriman, Ewa Deelman, John Good, Joseph C. Jacob, Daniel S. Katz, Anastasia C. Laity, Thomas A. Prince,
Gurmeet Singh, and Mei-Hui Su

3.1 Introduction

Astronomy has a rich heritage of discovery using image data sets that cover the full range of the electromagnetic spectrum. Image data sets in one frequency range have often been studied in isolation from those in other frequency ranges. This is mostly a consequence of the diverse properties of the data collections themselves. Images are delivered in different coordinate systems, map projections, spatial samplings, and image sizes, and the pixels themselves are rarely co-registered on the sky. Moreover, the spatial extent of many astronomically important structures, such as clusters of galaxies and star formation regions, is often substantially greater than that of individual images.

Astronomy thus has a need for image mosaic software that delivers mosaics that meet end users' image parameters (size, coordinates, spatial sampling, projection, rotation) while preserving the astrometric and photometric integrity of the original images. The Montage [299] software package[1] has been designed to meet this need. A driver in the design of Montage has been the requirement that Montage be usable without modification on end users' desktops, clusters, computational Grids, and supercomputers. This design goal has been achieved by delivering Montage as a toolkit in which the processing tasks in computing a mosaic are performed in independent modules that can be controlled through simple executables. The processing is easily performed in parallel computing environments with the processing of images performed on as many processors as are available. This approach has been successfully demonstrated with two instances of parallel technology—MPI (Message Passing Interface) [389] and Pegasus (Chapter 23). An on-demand image mosaic service has been built on the TeraGrid [412] and is currently under evaluation by astronomers, who simply submit a request for a mosaic using a Web form; the TeraGrid architecture is hidden from them. Montage can be

---

[1] `http://montage.ipac.caltech.edu`.

considered an enabling technology in that the mosaics it generates will widen avenues of astronomical research, including deep source detection by combining data over multiple wavelengths and studying the wavelength-dependent structure of extended sources, and image differencing to detect faint features.

The execution of complex workflows that produce image mosaics requires an understanding of the design philosophy of Montage and the algorithms implemented in it. Therefore we preface the discussion of parallelization and workflows with this topic.

3.2 The Architecture of Montage

3.2.1 Architectural Components

Montage employs the following four steps to compute a mosaic:

- Reprojection of input images to a common spatial scale, coordinate system, and World Coordinate System (WCS) projection
- Modeling of background radiation in images to achieve common flux scales and background levels by minimizing the interimage differences
- Rectification of images to a common flux scale and background level
- Co-addition of reprojected, background-corrected images into a final mosaic

Montage accomplishes these computing tasks in independent modules written in ANSI C for portability; they are listed in Table 3.2.1 and shown as a parallelized workflow in Figure 3.1. This "toolkit" approach controls testing and maintenance costs and provides considerable flexibility to users. They can, for example, use Montage simply to reproject sets of images and co-register them on the sky, or implement a custom background-removal algorithm without impact on the other steps, or define a specific processing flow through custom scripts.

3.2.2 A General Reprojection Algorithm

To support the broadest range of applications, the basic Montage reprojection and image-flux redistribution algorithm works on the surface of the celestial sphere. All pixel vertices from both input and output images are projected onto this sphere; if necessary, a coordinate system transform is applied to the input pixel vertices to put their sky coordinates in the same frame as the output. Then, for overlapping pixels, the area of overlap (in steradians) is determined. This overlap, as a fraction of the input pixel area, is used to redistribute the input pixel "energy" to the output pixels. In this way, total energy is conserved for those input pixels that do not extend beyond the bounds of the output image area. Even when a pixel has "undefined" vertices, such as at the boundaries of an Aitoff All-sky projection, the same process can

Component	Description
Mosaic Engine Components	
mImgtbl	Extracts the FITS header geometry information from a set of files and creates an ASCII image metadata table from it used by several of the other programs.
mProject	Reprojects a single image to the scale defined in a pseudo-FITS header template file. Produces a pair of images: the reprojected image and an "area" image consisting of the fractional input pixel sky area that went into each output pixel.
mProjExec	A simple executable that runs mProject for each image in an image metadata table.
mAdd	Coadds the reprojected images using the same FITS header template and working from the same image metadata table.
Background Rectification Components	
mOverlaps	Analyzes an image metadata table to determine a list of overlapping images.
mDiff	Performs a simple image difference between a single pair of overlapping images. This is meant for use on reprojected images where the pixels already line up exactly.
mDiffExec	Runs mDiff on all the pairs identified by mOverlaps.
mFitplane	Fits a plane (excluding outlier pixels) to an image. Meant for use on the difference images generated above.
mFitExec	Runs mFitplane on all the mOverlaps pairs. Creates a table of image-to-image difference parameters.
mBgModel	Modeling/fitting program that uses the image-to-image difference parameter table to interactively determine a set of corrections to apply to each image to achieve a "best" global fit.
mBackground	Removes a background from a single image (planar has proven to be adequate for the images we have dealt with).
mBgExec	Runs mBackground on all the images in the metadata table

Table 3.1: The design components of Montage.

be applied by determining an edge pixel's outline on the sky, described in the general case as a spherical polygon. The co-addition engine then creates the final mosaic by reading the reprojected images from memory and weighting each pixel's flux by the total input area [48].

This approach is completely general and preserves the fidelity of the input images. A comparison of sources extracted from the mosaics with those extracted from the original images shows that, in general, Montage preserves photometric accuracy to better than 0.1% and astrometric accuracy to better than 0.1 pixels [301]. Generality in reprojection is achieved at the expense of processing speed. For example, reprojection of a 512×1024 pixel Two Micron All Sky Survey (2MASS) [387] image takes 100 seconds on a machine equipped
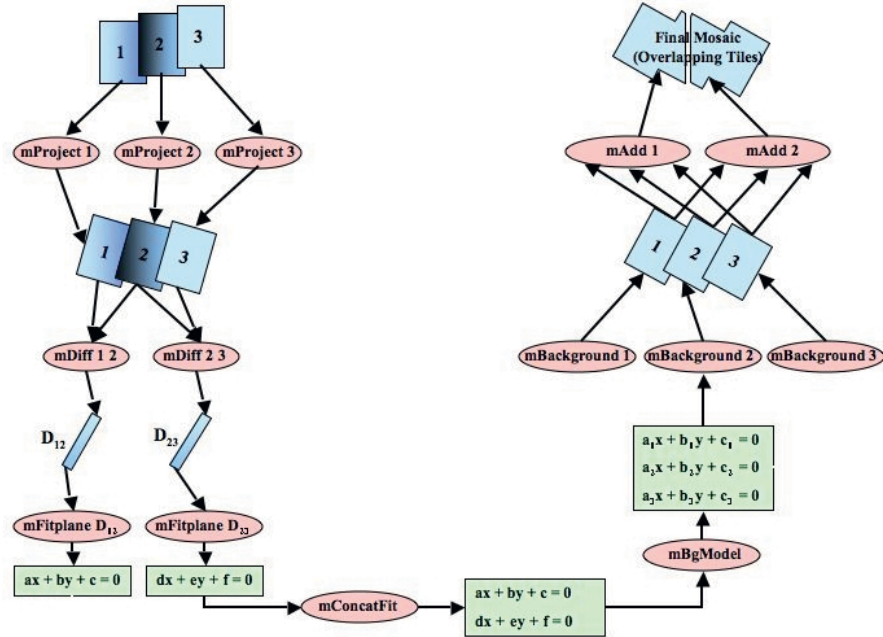
Figure 3.1: Montage workflow.

with a 2.26 GHz Intel processor and 1 GB memory running Red Hat Linux 8.0.

The algorithm described above was deployed in the first public distribution of the Montage software, version 1.7 [300]. Two further drawbacks inherent in this distribution are that the maximum image-mosaic size is limited by the available computer memory, and co-addition of flux in the reprojected pixels only supports weighting by area coverage.

The Montage team has taken advantage of the software's modular design to address these limitations in subsequent distributions. These improvements have taken the following forms:

- A general co-addition algorithm (Section 3.2.3)
- Custom, fast reprojection algorithms applicable to commonly used astronomical projections that bypass projection of pixels onto a sphere and transform input pixel flux directly into output pixel space (Section 3.2.4)
- Exploitation of the parallelization inherent in the design—many of the steps needed to compute a mosaic can be performed in parallel (Section 3.3)

The following sections describe these optimizations in more detail.

3.2.3 A General Co-addition Algorithm for Montage

The limitations of the available memory on the processing machine have been simply overcome by reading the reprojected images a single line at a time from files that reside on disk. Assuming that a single row of the output file does not fill the memory, the only limitation on file size is imposed by the file system. Images of up to 30 GB have been built with the new software. The algorithm has also been developed further to support quite general co-addition methods. For each output line, the co-addition module determines which input files will be contributing pixel values and opens only those files. Each contributing pixel value is read from the flux and area coverage files, and the value of each of these pixels is stored in an array until all contributing pixels have been read for the corresponding output row. This array constitutes a "stack" of input pixel values; a corresponding stack of area coverage values is also preserved. The contents of the output row are then calculated one output pixel (i.e., one input stack) at a time by averaging the flux values from the stack. Different algorithms to perform this average can be trivially inserted at this point in the program. The greater flexibility of the new software comes at the modest expense of 30% in speed.

Currently, Montage supports mean and median co-addition, with or without weighting by area. The mean algorithm (default) accumulates flux values contributing to each output pixel and then scales them by the total area coverage for that pixel. The median algorithm ignores any pixels whose area coverage falls below a specific threshold and then calculates the median flux value from the remainder of the stack. This median input pixel is scaled by its corresponding area coverage and written as the output pixel. If there are no area files, then the algorithm gives equal weight to all pixels. This is valuable for science data sets where the images are already projected into the same pixel space. An obvious extension of the algorithm is to support outlier rejection, and this is planned for a future release as an enhancement.

3.2.4 Performance Improvements through Custom Reprojection Algorithms

In its general form, the Montage reprojection algorithm transforms pixel coordinates in the input image to coordinates on the sky and then transforms that location to output-image pixel space. Under certain circumstances, this can be replaced by a much faster algorithm that uses a set of linear equations (though not a linear transform) to transform directly from input pixel coordinates to output pixel coordinates. This alternative approach is limited to cases where both the input and output projections are "tangent plane" (Gnomonic, orthographic, etc.), but since these projections are by far the most commonly used in astronomy, it is appropriate to treat them as a special case.

This "plane-to-plane" approach is based on a library developed at the Spitzer Science Center [302]. When both images are tangent plane, the geometry of the system can be viewed as in Figure 3.2, where a pair of Gnomonic

projection planes intersects the coordinate sphere. A single line connects the center of the sphere, the projected point on the first plane, and the projected point on the second plane. This geometric relationship results in transformation equations between the two planar coordinate systems that require no trigonometry or extended polynomial terms. As a consequence, the transform is a factor of 30 or more faster than using the normal spherical projection.

A bonus to the plane-to-plane approach is that the computation of pixel overlap is much easier, involving only clipping constraints of the projected input pixel polygon in the output pixel space.
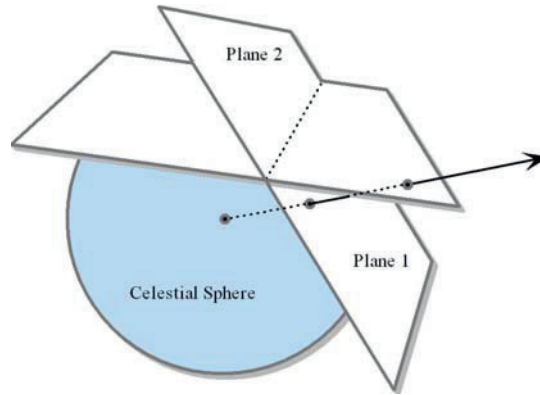
Figure 3.2: The principle of plane-to-plane reprojection.

This approach excludes many commonly used projections such as "Cartesian" and "zenithal equidistant" and is essentially limited to small areas of a few square degrees. Processing of all-sky images, as is almost always the case with projections such as Aitoff, generally requires the slower plane-to-sky-to-plane approach.

There is, however, a technique that can be used for images of high resolution and relatively small extent (up to a few degrees on the sky). Rather than use the given image projection, we can often approximate it to a very high degree of accuracy with a "distorted" Gnomonic projection. A distorted space is one in which the pixel locations are slightly offset from the locations on the plane used by the projection formulas, as happens when detectors are slightly misshapen, for instance. This distortion is modeled by pixel-space polynomial correction terms that are stored as parameters in the image FITS (Flexible Image Transport System) [142] header.

While this approach was developed to deal with physical distortions caused by telescope and instrumental effects, it is applicable to Montage in augmenting the plane-to-plane reprojection. Over a small, well-behaved region, most projections can be approximated by a Gnomonic (TAN) projection with small

distortions. For instance, in terms of how pixel coordinates map to sky co-ordinates, a two-degree "Cartesian" (CAR) projection is identical to a TAN projection with a fourth-order distortion term to within about 1% of a pixel width. Figure 3.3 shows this in exaggerated form for clarity, with the arrows showing the sense of the distortion.
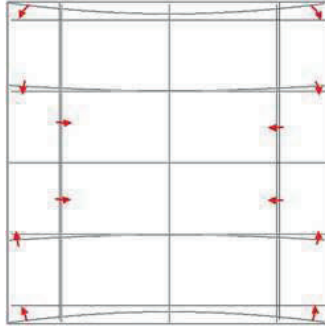

Figure 3.3: Representation of a WCS projection as a distorted Gnomonic (TAN) projection, exaggerated for clarity. The arrows indicate the sense of the distortions.

In this example, the curved coordinate Grid is an undistorted TAN and the rectangular Grid is both a CAR and the equivalent distorted TAN. This polynomial "correction" plus the plane-to-plane transform is still much faster than the normal reprojection. While this still does not cover all the possible transformations, it does include all those used for very large data collections.

3.3 Grid-Enabled Montage

3.3.1 Parallelism in Montage

Because the Montage components can require a large amount of time to complete a mosaic of reasonable size, various means of speeding up the calculations were examined. The slow speed of the calculations is due to three factors: the CPU speed, the compilers and memory systems limit how much of the CPU's peak performance can be obtained; and the I/O system limits how fast input images can be loaded from remote archives and also how the local disk stores intermediate and final results. Each of these limitations can be addressed, but addressing each adds complexity. The rest of this section will discuss potential solutions.

The simple solution to the limit of a CPU's performance is to use multiple CPUs. Currently, the market and commodity choice is to use multiple PC CPUs in a cluster, where each CPU was designed as an individual machine,

and to use some collective software and hardware, including networking, to make the system appear to be a single system in many ways.

To deal with the limits on what fraction of the peak performance of the CPUs Montage can exploit, it uses standard libraries where possible, as the libraries can be optimized by their developers better than standard code can be optimized by a compiler. Montage uses simple C code rather than C++ code around the libraries, as C can often be compiled into better-performing code than C++. C remains more portable than C++, though this is not a factor with modern hardware and compilers.

Finally, on the question of I/O limits to performance, the Montage design is kept as flexible as possible so that it can take best advantage of the network and disk systems that are available. In particular, Montage will benefit from parallel file systems where they exist.

Given a system of hardware consisting of multiple individual systems (nodes) that sometimes appear as a single system, C code and standard libraries, and a lack of dependence on the choice of I/O system, the question that is left to be answered is how to make all of these choices work together to solve the problem for which Montage was intended, construction of astronomical image mosaics, where parallelism is inherent, as seen in Figure 3.1.

The design of a set of simple applications connected by scripts lets us take advantage of a number of processing environments, including a single processor; a cluster of multiple processors with a shared file system; multiple clusters, each with a shared file system; a set of processors, each with its own file system; or any Grid-enabled hardware. For the single processor, the simple executables and scripts are sufficient. For the other cases, two different solutions have been implemented: an MPI approach and a Grid approach.

3.3.2 MPI Approach

MPI, the Message Passing Interface [389], is a standard that defines how various processes can work together to solve a single problem through exchanging messages. Messages can include data or can be used for synchronization. Two common programming paradigms are used in MPI programs: single program multiple data (SPMD) and master–worker. The Montage design provides a master–worker-like structure for many of the modules in the form of executables (such as mProjExec and mProject), and so the generation of MPI master–worker code would have been quite simple. Nevertheless, the SPMD model was adopted because master–worker applications scale with the number of workers, not the number of processors, and scaling with the number of processors was an explicit requirement from the sponsor. In general, the structure of the executables is similar in that each has some initialization that involves determining a list of files on which a worker module will be run, a loop in which the worker is called for each file, and some finalization work that includes reporting on the results of the worker runs. The executables are parallelized very simply in the SPMD paradigm, with all processes of a given

executable being identical to all the other processes of that executable. All the initialization is duplicated by all processors. A line is added at the start of the main loop, so that each processor only calls a worker module on its own processor if the remainder of the loop count divided by the number of processors equals the MPI rank. All processors then participate in global sums to find the total statistics of how many worker modules succeeded, failed, etc., as each processor initially keeps track of only its own statistics. After the global sums, only the processor with rank 0 prints out the global statistics.

mAdd, however, is different, as it writes to the output mosaic a single line at a time, reading from its input files as needed. The sequential mAdd writes the FITS header information into the output file before starting the loop on output lines. In the parallel mAdd, only the processor with rank 0 writes the FITS header information; then it closes the file. Now, each processor can carefully seek to the correct part of the output file and then write data, without danger of overwriting another processor's work. While the other executables were written to divide the main loop operations in a round-robin fashion, it makes more sense to parallelize the main mAdd loop by blocks since it is likely that a given row of the output file will depend on the same input files as the previous row, and this can reduce the amount of I/O for a given process.

Note that Montage includes two modules that can be used to build the final output mosaic, mAdd (to write a single output file) and mAddExec (to write tiled output files), and both can be parallelized as discussed in the previous two paragraphs. Currently, Montage runs one or the other, but it would be possible to combine them in a single run.

Some parts of the MPI-based Montage code, such as mImgtbl, will only use one processor, and other parts, such as mProjExecMPI, will use all the processors. Overall, most of the processors are in use most of the time. There is a small amount of overhead in launching multiple MPI jobs on the same set of processors. One might change the shell script into a parallel program, perhaps written in C or Python, to avoid this overhead, but this has not been done for Montage.

The timing results of the MPI version of Montage are shown in Figure 3.4. The total times shown in this figure include both the parallel modules (the times for which are also shown in the figure) and the sequential modules (the times for which are not shown in the figure but are relatively small).

MPI parallelization reduces the one-processor time of 453 minutes down to 23.5 minutes on 64 processors for a speedup of 19 times. Note that with the exception of some small initialization and finalization code, all of the parallel code is nonsequential. The main reason the parallel modules fail to scale linearly as the number of processors is increased is I/O. On a system with better parallel I/O, one would expect to obtain better speedups; the situation where the amount of work is too small for the number of processors has not been reached, nor has the Amdahl's law limit.

Note that there is certainly some variability inherent in these timings due to the activity of other users on the cluster. For example, the time to run
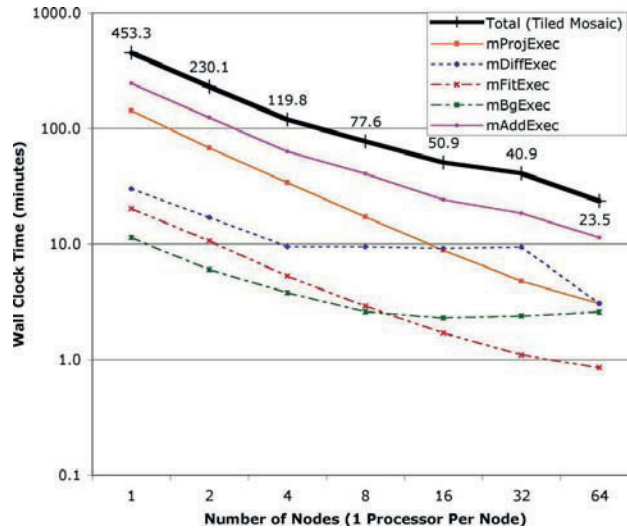
Figure 3.4: Performance of the MPI version of Montage building a 6×6 degree mosaic on the "Phase 2" TeraGrid [412] cluster at NCSA using dual 1.5 GHz Itanium-2 processors with at least 4 GB of memory.

mImgtbl should be the same in all cases since it is always run on a single processor. However, the measured results vary from 0.7 to 1.4 minutes. Also, the time for mDiffExec on 64 processors is fairly different from that on 16 and 32 processors. This appears to be caused by I/O load from other jobs running simultaneously with Montage. Additionally, since some of the modules' timings are increasing as the number of processors is increased, one would actually choose the fastest timing and run the module on only the number of processors that were used for that timing. For example, mBgExec on this machine should only be run on 16 processors, no matter how many are used for the other modules.

These MPI timings are probably close to the best that can be achieved on a single cluster and can be thought of as a lower bound on any parallel implementation. The MPI approach is suitable for a set of processors that share a file system, as there is implicit communication from one module to another through files, and these files must be visible to all processors. Additionally, if any processor fails in the MPI run, the entire MPI job and any remaining part of the Montage job will also fail. A more general solution to the problem of making use of multiple processors is to use a Grid approach.

3.3.3 Grid Approach

In the Grid approach, we create a workflow that describes the process of creating a mosaic using the Montage modules and use Pegasus (Chapter 23) for

executing the workflow over the Grid resources. Pegasus [110–112,116], which stands for Planning for Execution in Grids, is a framework that enables the mapping of complex workflows onto distributed resources such as the Grid. Pegasus maps an abstract workflow to a form that can be executed on a variety of computational platforms, from single hosts, to Condor pools [262], to compute clusters, to the TeraGrid. While the MPI-based approach focuses on running the Montage computations on a set of processors on a particular resource, such as a TeraGrid cluster, Pegasus takes a more generic approach by including mechanisms for resource and data discovery, mapping of the computations to the appropriate resources, orchestration of data transfers between the computations as needed, publication of the results in Grid catalogs, and other runtime optimizations in order to improve the execution efficiency.

In order to use the Pegasus approach, an abstract workflow is generated that describes the various tasks and the order in which they should be executed in order to generate the mosaic. The abstract workflow for Montage consists of the various application components as shown in Figure 3.1. The tasks in the abstract workflow represent the logical transformations such as mProject, mDiff, and others. The edges of the workflow represent the data dependencies between the transformations. For example, mConcatFit requires all the files generated by all the previous mFitplane steps. The rationale for choosing this particular workflow structure was to exploit the inherent parallelism in the Montage modules. Other workflow structures for Montage are also possible, such as the one consisting of Montage executables (e.g., mProjExec, etc). Even the previous MPI-based version of Montage could be represented as a workflow.

Pegasus queries Grid information services to find the location of compute and storage resources and to locate the physical replicas of the required input data for the workflow. It then maps each task in the abstract workflow to a compute resource based on a scheduling policy such as round-robin, random, etc. The MPI approach used a shared file system for sharing data between the Montage modules. In addition, Pegasus can transfer data using GridFTP [9] between the various tasks based on the dependencies in the workflow, where such shared file systems are not available. It transfers the input data required by the tasks to the compute resources and then transfers the created mosaic to a predefined location. These transfers are orchestrated by adding data transfer tasks to the workflow at the appropriate places. This results in the creation of a concrete workflow that can be executed using the Condor DAGMan (Chapter 22) [97] workflow engine. DAGMan submits tasks to the remote resources, monitors their execution, maintains the dependencies in the workflow, and retries in case of failures.

Pegasus can be used to generate concrete workflows that can execute on Grid resources that present a Globus Resource Allocation Manager (GRAM) [102] interface or on a local Condor [262] pool. The Condor pool can consist of dedicated or opportunistically shared resources. It can be constructed from remote Grid resources using a Condor feature known as glide-in [96]. Glide-in

can temporarily allocate a certain number of processors from a resource such as the TeraGrid, create a Condor pool from these allocated processors, and execute the workflow on this pool. Note that in both of these approaches, the resources used can be local or remote, dedicated or shared. The key difference is in the protocol used for submitting tasks to the resources and monitoring their execution.

There are overheads associated with execution of workflows on Grids due to the distributed nature of the resources, heterogeneity of the software components that need to interact, the scale and structure of the workflows, etc. These overheads are absent or minimal in the case of the MPI-based approach, and hence the mosaic creation time using the MPI-based approach can be considered to be the lower bound on the time taken to create the mosaic using the Grid approach. We have created a set of optimizations that reduce the overheads and improve the execution efficiency of the workflow. These optimizations include task clustering techniques that increase the computational granularity of the workflow and hence reduce the impact of the execution overhead on the workflow runtime. Experiments done using these optimizations have shown that the mosaic creation time using the Grid approach compares favorably with the MPI approach when the number of processors allocated is less than 64 (Figure 3.5) [234].
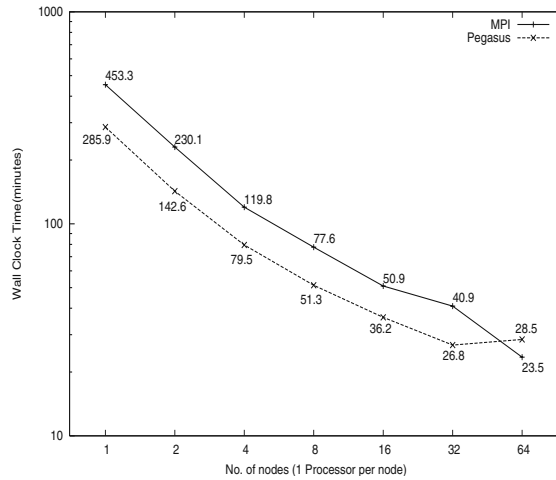

Figure 3.5: Times for building and executing the concrete workflow for creating a 6×6 degree mosaic of the M16 region.

3.4 Supporting a Community of Users

The Montage software is available[1] through a free "clickwrap" license issued by the California Institute of Technology. Users of the software fall into two groups: those who wish to order a mosaic from a third party and download the resulting mosaic, and those who download and install the software on desktops or incorporate it into processing environments. Section 3.4.1 describes the architecture and operation of a portal for users who wish to request mosaics online, and Section 3.4.2 describes examples of how Montage is being used within processing environments to generate science and education and public outreach products.

3.4.1 A Grid Portal for Montage

This section illustrates how to combine application-specific services and Grid-based services to provide users with a Montage portal. An advanced prototype of the architecture described below has been developed. When fully deployed, this portal will be publicly accessible and will operate on a 24/7 basis. The service is likely to process roughly 20,000 requests per month, based on similar requests for 2MASS images at the NASA/IPAC Infrared Science Archive (IRSA).

The Montage TeraGrid portal has a distributed architecture, as illustrated in Figure 3.6. The portal is comprised of the following five main components, each having a client and server: (i) User Portal, (ii) Abstract Workflow service, (iii) 2MASS Image List service, (iv) Grid Scheduling and Execution service, and (v) User Notification service. These components are described in more detail below.

User Interface

Users on the Internet submit mosaic requests by filling in a simple Web form with parameters that describe the mosaic to be constructed, including an object name or location, mosaic size, coordinate system, projection, and spatial sampling. After request submission, the remainder of the data access and mosaic processing is fully automated, with no user intervention. The server side of the user portal includes a CGI program that receives the user input via the Web server, checks that all values are valid, and stores the validated requests to disk for later processing. A separate daemon program with no direct connection to the Web server runs continuously to process incoming mosaic requests. The processing for a request is done in two main steps:

1. Call the Abstract Workflow service client code
2. Call the Grid Scheduling and Execution service client code and pass to it the output from the Abstract Workflow service client code
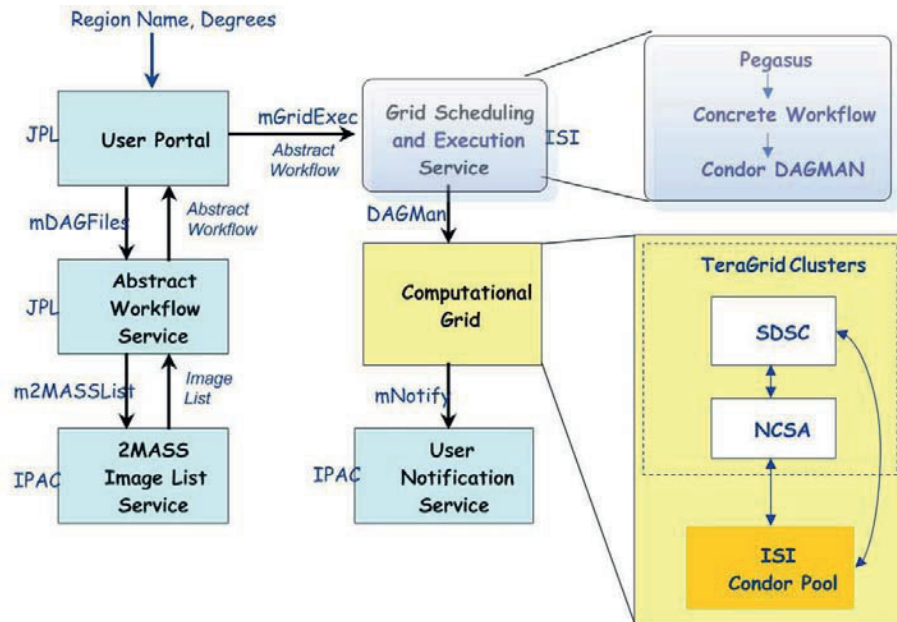
---

[1] `http://montage.ipac.caltech.edu/docs/download.html.`

Figure 3.6: The distributed architecture of the Montage TeraGrid portal.

Abstract Workflow service

The Abstract Workflow service takes as input a celestial object name or location on the sky and a mosaic size and returns a ziped archive file containing the abstract workflow as a directed acyclic graph (DAG) in XML and a number of input files needed at various stages of the Montage mosaic processing. The abstract workflow specifies the jobs and files to be encountered during the mosaic processing and the dependencies between the jobs.

Image List service

The Image List service takes as input a data-set identifier, celestial object name or location on the sky (which must be specified as a single argument string), and a mosaic size. The astronomical images from the specified data-set (e.g., 2MASS) that intersect the specified location on the sky are returned in a table, with columns that include the filenames and other attributes associated with the images.

Grid Scheduling and Execution service

The Grid Scheduling and Execution service takes as input the ziped archive generated by the Abstract Workflow service, which contains the abstract workflow and all of the input files needed to construct the mosaic. The service authenticates users, schedules the job on the Grid using Pegasus, and then executes the job using Condor's DAGMan.

Users are authenticated on the TeraGrid using their Grid security credentials. Users first need to save their proxy credential on the MyProxy server [314]. MyProxy is a credential repository for the Grid that allows a trusted server (such as our Grid Scheduling and Execution service) to access Grid credentials on the user's behalf. This allows the appropriate credentials to be retrieved by the portal using the user's username and password.

Once authentication is completed, Pegasus schedules the Montage workflow onto the TeraGrid or other clusters managed by PBS and Condor. Upon completion, the final mosaic is delivered to a user-specified location and the User Notification service, described below, is contacted.

User Notification service

The last step in Grid processing is to notify the user of the URL where the mosaic may be downloaded. This notification is done by a remote User Notification service so that a new notification mechanism can be used later without having to modify the Grid Scheduling and Execution service. Currently the user notification is done with a simple email, but a later version could provide more sophisticated job monitoring, query, and notification capabilities.

Our design exploits the parallelization inherent in the Montage architecture. The Montage Grid portal is flexible enough to run a mosaic job on a number of different cluster and Grid computing environments, including Condor pools and TeraGrid clusters. We have demonstrated processing on both a single cluster configuration and on multiple clusters at different sites having no shared disk storage.

3.4.2 Applications of Montage in Dedicated Processing Environments

One application of Montage is as a general reprojection engine to derive large-scale or full-sky images. Figure 3.7 shows an image of the 100 μm map of the sky by Schlegel, Finkbeiner, and Davis [379] that aggregates the sky maps produced by the Diffuse Infrared Background Experiment (DIRBE) aboard the Cosmic Background Explorer (COBE) and the Infrared Astronomical Satellite (IRAS), shown transformed from the Zenithal Equal Area projection to the Cartesian projection. This map is a science product that can be made accessible to astronomers online either as a single file for download or through a cutout Web service, which will deliver image subsets of arbitrary size centered

on a target position. The NASA/Infrared Processing and Analysis Center (IPAC) Infrared Science Archive (IRSA) is including this image as part of a broader science service that is required by the *Herschel* mission for observation planning. It will return estimates of the dust emission galactic emission and extinction along a line of sight, and when fully developed will return fluxes extrapolated to other wavelengths. The Spitzer/IPAC E/PO group is planning to deliver E/PO products made from such mosaics, including fold-out icosahedrons of the sky that will be distributed online.

Figure 3.7: The 100 μm sky represented in Cartesian projection, computed by Montage from composite DIRBE and IRAS skymaps of Schlegel, Finkbeiner, and Davis [379].
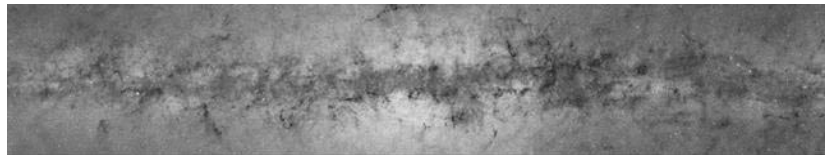

Figure 3.8: A section of the Galactic plane, 44 ° by 8 °, measured by 2MASS in the *J*-band and shown in Cartesian projection. The full-resolution image contains 4800 Megapixels.

A second application is generation of large-scale image mosaics, which can also be served as products either for download or through spatial subsetting services. Figure 3.8 shows a mosaic of a section of the Galactic plane in the 2MASS *J*-band [1], 44 ° long and 8 ° wide, centered on the Galactic Center and shown in Cartesian projection. The production of this mosaic was intended as a pilot project to provide resource estimates for generation of a full-sky 2MASS mosaic to be computed on the San Diego Supercomputer Center's

IBM DataStar supercomputer when fully commissioned. The mosaic was produced on a cluster of four 1.4 GHz Linux processors that processed the input images in parallel. By taking advantage of the algorithmic improvements described in Sections 2.3 and 2.4, the map was generated in 4 hours wall-clock time from 16,000 2MASS images in sine projection and containing 512×1024 pixels each.

Montage has found particular application to the *Spitzer Space Telescope*, and this is described in the remainder of this section.
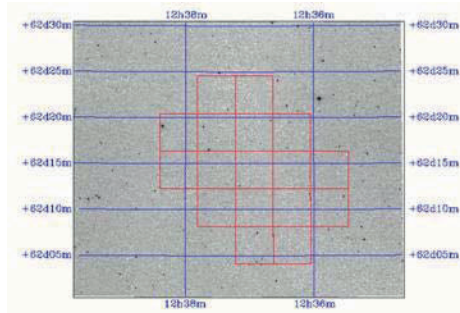

Figure 3.9: Footprints of *Hubble Space Telescope* (HST) Advanced Camera System (ACS) images in the Hubble Deep Field North supporting the Great Observatories Origins Deep Survey (GOODS), shown on a backdrop 2MASS K-band mosaic computed with Montage. The GOODS data query service is accessible at http://irsa.ipac.caltech.edu/data/GOODS.

Backdrops for Rendering Spatial Coverage of Spitzer Ancillary Observations. On behalf of the Spitzer Science Center, IRSA is serving ancillary, ground-based data supporting the Spitzer First Look Survey and Legacy projects. The data generally consist of collections of images, spectra, and source catalogs covering target areas that are generally several degrees on a side. These targets include the Lockman Hole, the ELAIS Fields, and the Hubble Deep Fields. Montage has been used to create mosaics, generally from 2MASS all-sky images, that act as background images that render the sky coverage of results of spatial searches for data. An example is shown in Figure 3.9.

Support for Data Production and Validation by Spitzer Space Telescope Legacy Teams. Two Spitzer Legacy teams, the Spitzer Wide-area InfraRed Extragalactic (SWIRE) survey [264] and the Galactic Legacy Infrared Mid-Plane Survey Extraordinaire (GLIMPSE) [163], are using Montage to support their processing pipelines, quality assurance, and mission planning. SWIRE is using the Infrared Array Camera (IRAC) and the MIPS (Millions of Operations Per Second) to trace the evolution of extragalactic populations at moderate redshifts. GLIMPSE is generating a four-color multiwavelength infrared

atlas of the Galactic plane with IRAC. Both projects are actively delivering scientific data products to the Spitzer Science Center (SSC).

SWIRE has been using Montage on Solaris platforms as a fast reprojection and co-addition engine to build sky simulations at a common spatial sampling that model the expected behavior of the sky, including galaxies, stars, and cirrus. These simulations have been used to validate the processing pipeline and source extraction. Predictions of the expected source populations and appearance of the sky have been used to plan the observing strategy. Following the launch of Spitzer, SWIRE is using Montage as an engine for co-registering images from different instruments, delivered with differing sampling frequencies, coordinate systems, and map projections, on a common spatial sampling scale and with common instrument parameters, and placing the backgrounds of each set of images on a common level. Figure 3.10 shows part of a 2.5 GB mosaic generated from images obtained with IRAC; the bright galaxy left of center is the Tadpole Galaxy. Montage was used here as a background rectification and co-addition engine applied to mosaic images generated as part of the Spitzer pipeline. The SWIRE project[1] has compiled a list of over 30 (as of March 2006) scientific publications that exploit SWIRE data products.

Figure 3.10: Part of a three-color mosaic of Spitzer Infrared Array Camera (IRAC) images. The complete mosaic is 10,000 pixels on a side.

Figure 3.11: Four-color IRAC mosaic of the Galactic star formation region RCW 49 measured at 3.6 μm, 4.5 μm, 5.8 μm, and 8 μm.

The GLIMPSE team has also integrated Montage into their Linux cluster-based pipeline. As part of their quality assurance program, they have used mosaics of the entire GLIMPSE survey region at J, H, K and MSX 8 μm [164]. They provide quick-look comparisons for quality assurance of the IRAC mosaics. An example of the early science data products is shown in Figure 3.11.

[1] http://swire.ipac.caltech.edu/swire/astronomers/publications.html

These data products are leading to a new understanding of the star formation in the plane of the Galaxy, in which star formation is proceeding at a much higher rate than previously thought, and have led to the discovery that the bar in the center of the Galaxy is some 7500 parsecs long, substantially longer than previously thought [44, 92]. The GLIMPSE team Web site (http://www.astro.wisc.edu/sirtf/glimpsepubs.html) has listed over 20 peer-reviewed papers (as of March 2006) that use the GLIMPSE data products.
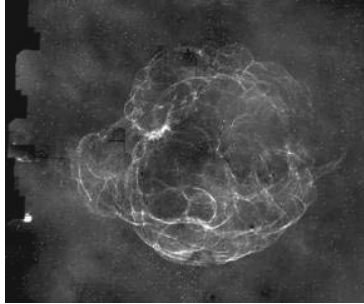

Figure 3.12: Continuum-subtracted Hα image of the supernova remnant S147. This is a very large-scale image, built by computing a mosaic of a large number of overlapping IPHAS fields. The total imaged area is roughly 5×3.5 square degrees.

The INT/WFC Photometric H-alpha Survey (IPHAS) is performing a deep Hα survey of the Southern Galactic Plane in the red (Sloan R and I bands). The project surveys short-lived phases of stellar evolution to significantly advance our knowledge of the extreme phases of stellar evolution, and as part of its operations is generating large-scale mosaics (5×5 square degrees) of regions of the Galactic plane. Figure 3.12 shows a sample image; more can be seen in [216, 217].

Acknowledgments

4

A Case Study on the Use of Workflow Technologies for Scientific Analysis: Gravitational Wave Data Analysis

Duncan A. Brown, Patrick R. Brady, Alexander Dietz, Junwei Cao, Ben Johnson, and John McNabb

4.1 Introduction

Modern scientific experiments acquire large amounts of data that must be analyzed in subtle and complicated ways to extract the best results. The Laser Interferometer Gravitational Wave Observatory (LIGO) is an ambitious effort to detect gravitational waves produced by violent events in the universe, such as the collision of two black holes or the explosion of supernovae [37, 258]. The experiment records approximately 1 TB of data per day, which is analyzed by scientists in a collaboration that spans four continents. LIGO and distributed computing have grown up side by side over the past decade, and the analysis strategies adopted by LIGO scientists have been strongly influenced by the increasing power of tools to manage distributed computing resources and the workflows to run on them. In this chapter, we use LIGO as an application case study in workflow design and implementation. The software architecture outlined here has been used with great efficacy to analyze LIGO data [2–5] using dedicated computing facilities operated by the LIGO Scientific Collaboration, the LIGO Data Grid. It is just the first step, however. Workflow design and implementation lies at the interface between computing and traditional scientific activities. In the conclusion, we outline a few directions for future development and provide some long-term vision for applications related to gravitational wave data analysis.

4.2 Gravitational Waves

Although Einstein predicted the existence of gravitational waves in 1916, the challenge in directly observing them is immense because of the extremely weak

coupling between matter and gravitation. Small amounts of slowly moving electric charge can easily produce detectable radio waves, but the generation of detectable amounts of gravitational radiation requires extremely massive, compact objects, such as black holes, to be moving at speeds close to the speed of light. The technology to detect the waves on Earth only became practical in the last decade of the twentieth century. The detection of gravitational waves will open a new window on the universe and allow us to perform unprecedented tests of general relativity. Almost all of our current knowledge about the distant universe comes from observations of electromagnetic waves, such as light, radio and X-ray. Gravitational waves, unlike electromagnetic waves, travel through matter and dust in the universe unimpeded. They can be used to see deep into the cores of galaxies or probe the moment when space and time came into being in the Big Bang.

Gravitational waves are ripples in the fabric of space-time; their effect on matter is to stretch it in one direction and squeeze it in the perpendicular direction. To detect these waves, LIGO uses three laser interferometers located in the United States. Two interferometers are at the Hanford Observatory in southeastern Washington and one is at the Livingston Observatory in southern Louisiana. The purpose of the multiple detectors is to better discriminate signal from noise, as a gravitational wave signal should be detectable by all three interferometers. Each interferometer consists of a vacuum pipe arranged in the shape of an L with 4 kilometer arms. At the vertex of the L and at the end of each of its arms are mirrors that hang from wires. Laser beams traversing the vacuum pipes accurately measure the distance between the mirrors in the perpendicular arms. By measuring the relative lengths of the two arms, LIGO can measure the effect of gravitational waves. These changes in length are minute, typically 10^{-19} meters over the 4 kilometer arms—much less than the size of a proton. To measure such small distances requires ultrastable lasers and isolation of the mirrors from any environmental disturbances. Any difference in the lengths of the arms, due to detector noise or gravitational waves, is detected as a change in the amount of light falling on a photodetector at the vertex of the L. Figure 4.1 shows a schematic diagram of a LIGO detector. In a perfect detector and in the absence of a gravitational wave, no light would fall on the photodetector. In practice, however, random fluctuations in the interferometer cause some light to fall on the detector. Among other sources, these fluctuations come from seismic noise from ground motion coupling into the mirrors, thermal noise from vibrations in the mirrors and their suspensions, and shot noise due to fluctuations in the photons detected by the photodetector. LIGO data analysis is therefore a classic problem in signal processing: determining if a gravitational wave signal is present in detector noise.

Data from the LIGO detectors are analyzed by the LIGO Scientific Collaboration (LSC), an international collaboration of scientists. The searches for gravitational waves in LIGO data fall broadly into four classes: compact binary inspiral, continuous waves from rotating neutron stars, unmodeled burst
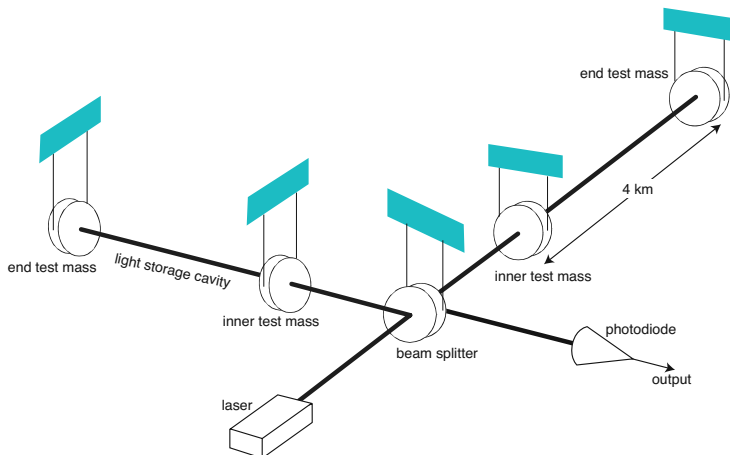
Figure 4.1: Schematic diagram of a LIGO detector. Laser light is incident on a partially reflective mirror or beamsplitter. Half the light is transmitted into one arm of the interferometer and half is reflected into the other arm. The light in each arm resonates between two mirrors that act as test masses and change position in response to a gravitational wave. The light is recombined at the beamsplitter, and the light incident on the photodiode contains information about the position of the mirrors and hence about any gravitational waves incident on the detector.

sources, and stochastic gravitational wave backgrounds. In this chapter we focus on the workflows used in the search for gravitational waves from compact binary inspirals. For details on the other searches, we refer the reader to [37].

The gravitational waves arising from coalescing compact binary systems consisting of binary neutron stars and black holes are one of the best understood sources for gravitational wave detectors such as LIGO [427]. Neutron stars and black holes are the remnants produced by the collapse of massive stars when they reach the end of their lives. If two stars are in a binary system, the compact bodies orbit around each other and lose energy in the form of gravitational waves. The loss of energy causes their orbit to shrink and their velocities to increase. The characteristic "inspiral" signal emitted increases in frequency and amplitude until the bodies finally plunge toward each other and coalesce, terminating the waveform. Figure 4.2 shows a time–frequency spectrogram of a simulated inspiral signal. It is expected that there will be approximately one binary neutron star coalescence every three years in the volume of the universe accessible to LIGO [231].

The shape of the inspiral waveform depends on the masses of the binary components. When both components are below approximately three solar masses, the waveform is well modeled by theoretical calculations and we can use matched filtering to find the signals in detector noise. For higher-mass
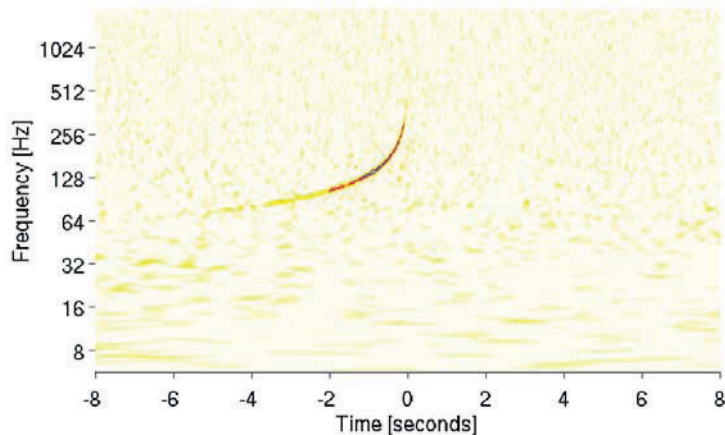
Figure 4.2: A time–frequency spectrogram of a simulated binary inspiral signal. The waveform increases in amplitude and frequency as time increases. The well-defined shape of the waveform makes matched filtering a suitable data-analysis technique.

waveforms, such as black hole binaries, uncertainties in the waveforms grow, but in practice we may continue to use matched filtering, albeit with a modified template family [68,69]. These templates are not exact representations of the signals but are designed to capture the essential features of the waveforms. The first science run of LIGO focused attention on the search for binary neutron stars [2]. The second science run refined the binary neutron star search [3] and extended the analysis to include searches for binary black hole systems with higher masses [5] and subsolar-mass binary black hole systems that may be components of the Milky Way Halo [4].

Analysis of the LIGO data for binary inspirals is performed using the LIGO Data Grid (LDG) [256]. In this chapter, we describe the LDG infrastructure, the software used to construct data analysis workflows for the LDG, and the components and execution of the inspiral analysis pipeline. Finally, we discuss the use of these tools by other gravitational wave searches and the extension of the workflows to other Grids, such as the Open Science Grid (OSG) [328].

4.3 The LIGO Data Grid Infrastructure

LSC scientists conducting gravitational wave data analysis need to analyze many terabytes of data. The scientists have access to a large number of distributed computing resources, including resources external to the collaboration. To fully leverage the distributed resources in an integrated and seamless way, infrastructure and middleware have been deployed to structure the resources as a Grid. The LIGO Data Grid infrastructure includes the LSC Linux

clusters, the networks that interconnect them to each other, Grid services running on the LSC Linux clusters, a system for replicating LIGO data to LSC computing centers, DOE Grids certificate authority authentication [120], and a package of client tools and libraries that allow LSC scientists to leverage the LIGO Data Grid services.

The LDG hardware consists of Linux clusters for data analysis and Linux and Sun Solaris servers used for data replication and metadata services. The hardware is distributed among the LIGO observatories, the LIGO Laboratories at the California Institute of Technology (Caltech), the Massachusetts Institute of Technology (MIT), and various LSC member institutions, as detailed below. The middleware software that supports Grid services and users is known as the LDG server package. The LDG server package itself is built on top of the Virtual Data Toolkit (VDT) [440] as provided by the international Virtual Data Grid Laboratory (iVDGL) [215] and OSG [328] projects. A subset of the LDG server software is distributed as the LDG client package and contains only the tools needed to access the computing clusters and discover LIGO data across the LDG. The LDG also uses some proprietary software, such as the Sun StorEdge SAM-QFS [401] software and the IBM DB2 [209] database. In this section, we describe the LDG hardware and software infrastructures in more detail.

4.3.1 Management of the Raw Detector Data

The LIGO detectors are sensitive to gravitational waves with frequencies between approximately 40 Hz and 4 kHz. The output signal from each of the three detectors is digitized as a 16 bit signal at a sample rate of 16384 Hz. In addition to the output photodiode signal, many other detector data channels are recorded at various sample rates between 8 Hz and 16384 Hz. These channels monitor the performance of the detector and its environment. The total output data rate of the observatories is 8 MB per second for Hanford and 4 MB per second for Livingston. The many channels are written to a high-performance file system, each individual file or *frame* containing 32 seconds of data. Approximately 10000 frame files are written per day at each observatory.

Distribution of these data is managed by the LIGO Data Replicator (LDR) [257], which provides robust replication and data discovery services. The LDR service is built on top of the Globus Replica Location Service (RLS) [88], Globus GridFTP [9], and a metadata catalog service. Each of these services is deployed separately from the other services in the LDG server package. Together, these services are used for replicating data. Data at the observatories are published into LDR and then replicated to the LIGO Laboratory at Caltech, which is responsible for permanent data storage and archiving of data. Other LDG sites deploy LDR to replicate particular subsets of LIGO data to the local site for data analysis. The subsets of LIGO data that are replicated can be configured by each site's local policy, and each site stores the data in accordance with its own local policies in terms of the directory structure.

Note that the LDR service replicates data in bulk to sites, independently of the demands of any particular data analysis job. In order to execute analysis workflows, LSC scientists need to be able to discover the location of specific LIGO data files across the LIGO Data Grid. The *LSCdataFind* tool included in the LDG client package allows LSC scientists to discover LIGO data based on gravitational wave detector specific metadata rather than discovery based on filenames. Typical metadata attributes used for finding LIGO data include a start and end time describing the epoch of data to be analyzed, the observatory at which the data were collected, and the class of LIGO data files (different classes or frame types contain different sets of data channels from the detectors).

The LSCdataFind tool by default returns a list of physical filenames (PFNs) or URLs for the location of LIGO data files at a particular LDG site. These PFNs can then be used directly by tools building a LIGO workflow, tailoring it for use at that particular site. In order to support the more sophisticated planning of the LIGO workflows detailed below, LSCdataFind also supports returning only the logical filenames (LFNs) of the data files meeting the user's metadata constraints. The LFNs are just the simple filenames and do not contain any location information.

4.3.2 Management of Detector Metadata

In addition to the access to raw detector data, LSC scientists need additional metadata, known as *data quality information*, which describe the state of the interferometers, when the data are suitable for analysis, and records information about periods of unusual behavior. These metadata are stored in the LSC segment database, which allows storage, retrieval, and replication of the data. The segment database uses the IBM DB2 database to provide the underlying relational-database engine. The publication scripts used to publish the data into LDR also publish detector state information into the segment database.

The segment databases at Caltech and the observatories are connected together by low-latency peer-to-peer database replication using the "Q-replication" service provided by DB2. Any metadata inserted at one of the three databases will be replicated to the two other databases with a latency of a few seconds to a couple of minutes. Replication time varies depending on the load on the databases. IBM WebSphere MQ [210] is used as the transport layer for replication between the databases. Message queues are set up between each of the servers that take part in the replication, and these are used by the replication programs to send and receive data and control messages.

Client and server tools written on top of the LDG server middleware allow scientists to connect to the database, query information, and insert new metadata based on detector characterization investigations. Segment discovery services are provided by the *LSCsegFind* server, which runs at each site and responds to user requests for segment and data-quality information. It constructs the SQL needed to service the user's request, executes the query

on the database, and returns the results to the user. The client and server communicate over a Globus GSI [72] authenticated connection. The server runs on the same machine as the DB2 database, and queries can be issued by remote clients, which are distributed as part of the LDG client bundle.

Metadata are exchanged in the LSC as XML data, with the LSC-specific schema called LIGO lightweight XML. The Lightweight Database Dumper (LDBD) provides a generic interface between the segment database and LIGO lightweight XML representations of table data in the database. The LDBD server can parse the contents of a LIGO lightweight XML document containing table data and insert them into the database. It can also execute SQL queries from a client and return the results as LIGO lightweight XML data. Data quality information is generated as LIGO lightweight XML by various data-monitoring tools and inserted via the LDBD server. This generic framework allows construction of metadata services specific to the various requirements of gravitational wave data analysis. Again, communication between the client and server is performed over a GSI-authenticated socket connection. The server runs on the same machine as the DB2 database, and queries can be issued by remote clients. The LDBD server is also capable of inserting LFN to PFN maps into an RLS server, if desired, to allow metadata to be associated with specific files.

4.3.3 Computing Resources

LSC scientists have access to a number of computing resources on which to analyze LIGO data. Some resources are dedicated Linux clusters at LSC sites, others are Linux clusters available via LSC partnership in large Grid collaborations such as the international Virtual Data Grid Laboratory (iVDGL) [215] and its successor the Open Science Grid [328], and still other resources are available via more general arrangements with the host institution. The vast majority of available computing resources are Intel [214] or AMD [21] based clusters running some version of the Linux operating system.

LSC Linux Clusters

The LSC itself has available as dedicated computing resources Linux clusters hosted at the LIGO observatories at Hanford and Livingston, at the LIGO host institutions Caltech and MIT [258], and at LSC computing sites hosted at the Pennsylvania State University (PSU) [342] and the University of Wisconsin—Milwaukee (UWM) [434]. In addition there are Linux clusters dedicated for gravitational wave data analysis made available by the British–German GEO 600 [470] gravitational wave detector, which is also a member of the LSC.

Each dedicated LSC Linux cluster and its related data storage hardware is categorized as a Tier 1, 2, or 3 site depending (in a rough way) on the amount of computing power and data storage capacity available at the site. The LIGO Caltech Linux cluster, with over 1.2 teraflops (TFlop) of CPU

and 1500 terabytes (TB) of data storage, serves as the Tier 1 site for the collaboration. All LIGO data are archived and available at the Tier 1 site. The detector sites at Hanford and Livingston, although the LIGO data originate there, are considered to be Tier 2 sites. The Hanford site has available 750 gigaflops (GFlop) of CPU and 160 TB of data storage, while the Livingston site has available 400 GFlops of CPU and 150 TB of data storage. The LIGO MIT site is also considered a Tier 2 site, with 250 GFlops of CPU and 20 TB of data storage. The PSU and UWM sites are operated as Tier 2 sites. The PSU site includes 1 TFlop of CPU and 35 TB of storage. The UWM site has operated in the past with 300 GFlops of CPU and 60 TB of storage, although it is currently being upgraded to 3 TFlops and 350 TB of storage.

Each of the Linux clusters within the LIGO Data Grid deploys a set of standard Grid services, including Globus GRAM [147] for submitting jobs and resource management, a Globus GridFTP server for access to storage, and a GSI-enabled OpenSSH server [182] for login access and local job submission. All of these services authenticate via digital certificate credentials. The middleware software that supports these and other Grid services is deployed using the LDG server package.

Other Computing Resources

Through LSC membership in large Grid computing projects and organizations, LSC scientists have access to a large number of computing resources outside of the dedicated LSC computing resources. The LSC was a founding contributor to iVDGL, and much of the development and prototyping of the effort described here was done as part of an effort to allow LSC scientists to leverage iVDGL resources not owned by the LSC. In particular, the initial prototyping of the LIGO inspiral workflow management that leverages the use of Condor DAGMan (see Chapter 22, and reference [97]) and Pegasus (see Chapter 23 and references [111], [112], [116]) was driven by the desire to leverage the Grid3+ [172] resources made available by the iVDGL collaboration. The more recent work done to run LIGO inspiral workflows on non-LSC resources is targeted at running on the Open Science Grid. In addition, LSC scientists (in particular those running inspiral workflows) have access to the large computing resources from the Center for Computation and Technology at Louisiana State University [85].

4.3.4 Batch Processing

All of the LSC Linux clusters, with the exception of the cluster at PSU, use Condor (see Chapter 22) as the local batch scheduler. As discussed in detail below, this has allowed LSC scientists to begin developing complex workflows that run on a single cluster and are managed by Condor DAGMan. To run workflows across LSC clusters running Condor and leverage geographically

distinct resources as part of a single workflow, the LSC has investigated using Condor-only solutions such as Condor Flocking [133].

The Linux clusters at PSU and LSU, however, use the Portable Batch System (PBS) [339] for managing batch jobs, and since these resources represent a significant fraction of the resources available to LSC scientists, it is important that the workflows also be able to leverage those resources. In addition, a majority of the resources available outside the LDG use a tool other than Condor for managing compute jobs. While recent development work from the Condor group involves providing access to non-Condor managed resources directly from a Condor-only environment, the workflow management work described here has focused on using a blended approach that involves tools beyond Condor and Condor DAGMan.

4.3.5 LIGO Data Grid Client Package

LIGO Data Grid users install the LDG client package on their workstations. The LDG client package is also built on top of the VDT but only includes a subset of the client tools and libraries. No Grid services are deployed as part of the client package. In addition to the client tools from the VDT, a number of client tools specifically for use in creating and managing LIGO workflows are included in the client package. The most significant of these are the tools LSCdataFind, used for data discovery, and LSCsegFind, used for data quality information retrieval across the LIGO Data Grid.

4.4 Constructing Workflows with the Grid/LSC User Environment

In the previous section, we described the hardware and middleware infrastructure available to LSC scientists to analyze LIGO data. In this section, we describe the Grid/LSC User Environment (Glue), a toolkit developed to allow construction of gravitational wave data analysis workflows. These workflows can be executed on LSC Linux clusters using the Condor DAGMan workflow execution tool or planned and executed on wider Grids, such as the OSG, using the Pegasus workflow planner, Condor DAGMan, and Globus GRAM.

4.4.1 Overview of LIGO Workflows

LIGO data analysis is often referred to as "embarrassingly parallel," meaning that although huge quantities of data must be analyzed over a vast parameter space of possible signals, parallel analysis does not require interprocess communication. Analysis can be broken down into units that perform specific tasks that are implemented as individual programs, usually written in the C programming language or the Matlab processing language/environment. Workflows may be parallelized by splitting the full parameter space into smaller

blocks or parallelizing over the time intervals being analyzed. The individual units are chained together to form a data analysis pipeline. The pipeline starts with raw data from the detectors, executes all stages of the analysis, and returns the results to the scientist. The key requirements that must be satisfied by the software used to construct and execute the pipelines:

1. Ensure that all data are analyzed and that the various steps of the workflow are executed in the correct sequence
2. Automate the execution of the workflow as much as possible
3. Provide a flexible pipeline construction toolkit for testing and tuning workflows
4. Allow easy, automated construction of complex workflows to analyze large amounts of data
5. Have a simple reusable infrastructure that is easy to debug

In order to satisfy the first two requirements, we implement a data analysis pipeline as a directed acyclic graph (DAG) that describes the workflow (the order in which the programs must be called to perform the analysis from beginning to end). A DAG description of the workflow can then be submitted to a batch processing system on a computing resource or to a workflow planner. The pipeline construction software must maintain an internal representation of the DAG, which can then be written out in the language that a batch processing system or a workflow planner can understand. By abstracting the representation of the workflow internally, the workflow may be written out using different syntaxes, such as a Condor DAGMan input file or the XML syntax (known as DAX) used by the Pegasus workflow planner. To simplify the construction of DAGs for gravitational wave data analysis, the LSC has developed the Grid/LSC User Environment, or Glue, a collection of modules, written in the Python language, developed especially for LSC scientists to help build workflows.

The components of a DAG are its *nodes* and *edges*. The nodes are the individual analysis units and the edges are the relations between the nodes that determine the execution order. Each node is assumed to be an instance of a *job* that performs a specific task in the workflow. Glue contains three basic abstract classes that represent DAGs, jobs, and nodes. The DAG class provides methods to add nodes and write out the workflow in various formats. The job class provides methods to set the name of the executable and any options or arguments common to all instances of this job in the DAG. The node class, which inherits from the job class, provides methods to set arguments specific to a node, such as the start and stop time to be analyzed, or the required input files. The node class also has a method to add parent nodes to itself. The edges of the DAG are constructed by successive calls to `add_parent` for the nodes in the workflow. The executables to be run in the DAG read their arguments from the command line and read and write their input from the directory in which they are executed. This constraint is enforced to allow portability to Grid environments, discussed below. Glue also knows about other LIGO-
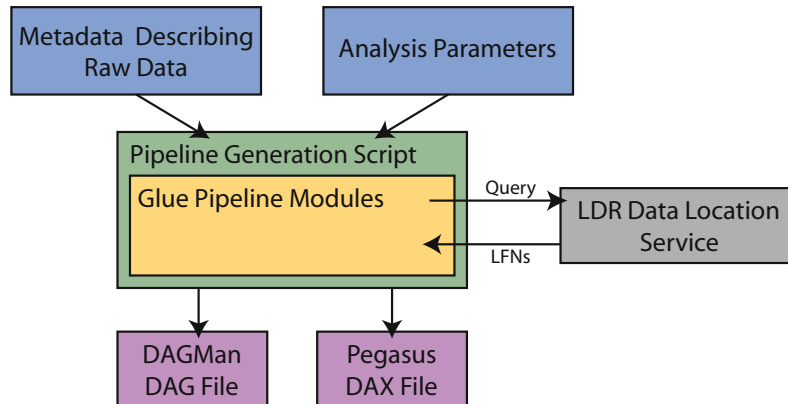
Figure 4.3: The Glue pipeline modules are used by LSC scientists to write pipeline generation scripts. Pipeline scripts take as input analysis parameters and metadata describing the raw data and output workflows as DAGMan DAG files or Pegasus DAX files, which can be used to execute the pipeline. If Glue is generating a Pegasus DAX, the pipeline modules can query the LDR data location service to obtain LFNs for the input data, as described in Section 4.4.4.

specific concepts, such as *science segments* (time epochs of LIGO data suitable for analysis) and the methods that are used to split these segments into *blocks* or subunits of science segments used to parallelize workflows. By providing iterators for these classes, it is simple to loop over segments and blocks in the construction of a workflow.

To address the specific needs of different analysis tasks, the user writes a pair of classes that describe the task to Glue: a job class and a node class that inherit from the base classes. The user may extend or override the base methods to allow the pipeline construction scripts to set options particular to the task being described. In this way, the components of the workflow are abstracted, and it is straightforward to write pipeline scripts that construct complex workflows. The Glue method of constructing data analysis pipelines has been used in the binary inspiral analysis, the search for gravitational wave bursts from cosmic strings, excess power burst analysis, and stochastic gravitational wave background analysis. Figure 4.3 shows how Glue is used in workflow construction, with metadata and analysis parameters taken as input and different workflow styles written as output. Below we give an example of a script to construct a simple workflow, and Section 4.5 describes how this is used in practice for the inspiral analysis pipeline.

```
from glue import pipeline
import gwsearch

data = pipeline.ScienceData()
data.read('segments.txt',2048)
data.make_chunks(2048)
dag = pipeline.CondorDAG('myworkflow')

datafind_job = pipeline.LSCDataFindJob()
datafind_job.add_option('data-type','raw')
previous_df = None

gwsearch_job = analysis.GWSearchJob()

for seg in data:
        df = pipeline.LSCDataFindNode()
        df.set_start(seg.start())
        df.set_end(seg.end())
        for chunk in seg:
                insp = gwsearch.GWSearchNode()
                insp.set_start(chunk.start())
                insp.set_end(chunk.end())
                insp.add_parent(df)
        if previous_df:
                df.add_parent(previous_df)
        previous_df = df

dag.write_dag()
```

Figure 4.4: Example code showing the construction of a workflow using Glue. The input data times are read from the file `segments.txt`. For each interval in the file, an LSCdataFind job is run to discover the data and a sequence of inspiral jobs are also run to analyze the data. The workflow is written to a Condor DAG file called `myworkflow.dag`, which can be executed using DAGMan.

4.4.2 Constructing a Workflow with Glue

In this example, an LSC scientist wishes to analyze data from a single LIGO detector through a program called *GWSearch*, which analyzes data in blocks of 2048 seconds duration. Figure 4.4 shows the Python code necessary to construct this workflow using Glue. The user has written a pair of classes that describe the job and nodes for the GWSearch program, as described in the previous section, and the script imports them along with the pipeline generation module from Glue. The user has requested a list of times from the segment database that are suitable for analysis and stored them in a text file
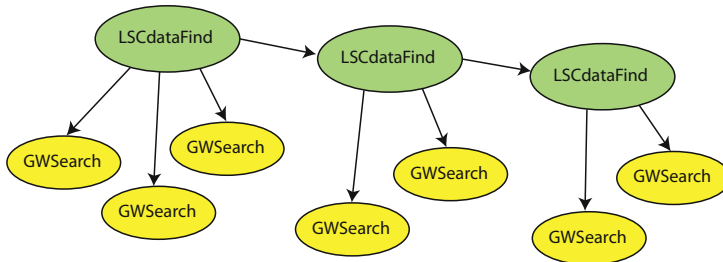
Figure 4.5: The workflow constructed by the sample code shown in Figure 4.4. In this case, there are three segments used as input, the first of which contains three 2048 second blocks and the second and third containing two 2048 second blocks. The resulting workflow has ten nodes.

named `segments.txt`. This file contains a list of start and stop times in GPS seconds, which may vary in length between several seconds and many hours. The user's pipeline script creates a representation of these intervals using the Glue `ScienceData` class. The segments are parsed from the file by the `read` method, which is told to discard any segments shorter than 2048 seconds. The segments are then split into blocks of length 2048 seconds by the `make_chunks` method.

To construct a workflow, the script first creates a representation of the workflow itself using the `CondorDAG` class. Instances of the `LSCDataFindJob` and `GWSearchJob` classes are then created to describe the programs that will be used in the workflow. Next the script iterates over all segments in the `data` class and constructs a node in the workflow that performs an LSCdataFind job to find the data for each segment. There is then a second loop over the 2048 second blocks within each segment and a node to execute the GWSearch program on each block. A dependency is created between the LSCdataFind and the GWSearch jobs by using the `add_parent` method of the GWSearch nodes. This ensures that the GWSearch jobs do not execute until the LSCdataFind job is complete. Finally, a relation is created between the LSCdataFind jobs, so that only one job executes at a time; this is a technique used in real workflows to reduce the load on the server. The final workflow constructed by this example is shown in Figure 4.5 for a segment file that contains segments of lengths 6144, 4192, and 4192 seconds.

4.4.3 Direct Execution Using Condor DAGMan

Once the script to generate an analysis pipeline has been written, the resulting workflow must be executed on an LSC computing cluster. As described previously, most of the LSC clusters run the Condor batch processing system. The `write_dag` method of the Glue DAG class creates a DAG in Condor DAGMan format, as well as the necessary Condor submit files to execute the jobs. DAGs

for LSC data analysis range in size from a few tens of nodes to over 100000 nodes. The DAG written by the pipeline script is submitted to Condor, which ensures that all the nodes are executed in the correct sequence. If any node fails, for example due to transient errors on cluster nodes, a rescue DAG is created containing only the nodes that failed or were unable to execute due to failures. This rescue DAG can be resubmitted to Condor and in this way LSC scientists can ensure that all data have been correctly and completely analyzed.

4.4.4 Planning for Grids with Pegasus

To complete a search for gravitational waves, it is necessary to run many large-scale Monte Carlo simulations with simulated signals added to the data. The results of these simulations are used to measure the efficiency and tune the parameters of the search. This requires a great deal of computing power, and Glue has been extended to write workflows in the *abstract DAG* (DAX) format so they can be planned for grid execution with Pegasus.

When running data on the Grid, it is no longer guaranteed that the LIGO data are present on the computing cluster on which the job will execute. Glue has been modified so that when it is instructed to write a DAX it does not add any requested LSCdataFind nodes to the workflow. Instead it queries the LDR data discovery service to find the logical filenames (LFNs) of the input data needed by each node and adds this information to the DAX. When the workflow is planned by Pegasus on a given list of potential Grid sites, it queries the Globus RLS servers deployed on the LIGO Data Grid to determine the physical filenames or URLs of the input data. Pegasus then adds transfer nodes to the workflow to stage data to sites that do not have the input data and uses local replicas of the data on those sites that already have the necessary input data available. In addition to the LFNs of the input data, Glue also writes the LFNs of all intermediate data products in the DAX so that Pegasus may plan the workflow across multiple sites. One of the key features of Glue is that this is transparent to the user. Once users have written their workflow generation script, they may simply add a command-line switch that calls the `write_dax` method rather than `write_dag`, and Glue will produce a DAX description of the workflow suitable for use with Pegasus.

4.5 The Inspiral Analysis Workflow

In the previous sections, we have described the infrastructure of the LIGO Data Grid and the construction of workflows using Glue. In this section, we describe the use of these tools to implement the search for compact binary inspirals in LIGO data, with practical examples of the workflow.

The signal from a true gravitational wave should be present in all the LIGO detectors. It should occur at the same time in the two detectors at the

Hanford Observatory, and no later than the light-travel time of 10 ms at the Livingston Observatory. The actual time delay between observatories varies, depending on where on the sky the signal originates. Triggers are said to be *coincident* if they have consistent start times. The triggers must also be in the same waveform template and may be required to pass additional tests, such as amplitude consistency. The triggers that survive all coincidence tests are the output of the inspiral analysis pipeline and are known as *event candidates*. Further manual follow-up analysis is used to determine if the triggers are truly due to gravitational waves.

If one detector is more sensitive than the two other detectors, as was the case in the second LIGO science run, we may only wish to analyze data from the less sensitive detectors when there is a trigger in the most sensitive detector. If the detectors are equally sensitive, as is presently the case, we may wish to demand that a trigger from the matched filter be present in all three detectors before computing computationally expensive signal-based vetoes.

4.5.1 Components of the Inspiral Analysis

The inspiral workflow is divided into blocks that perform specific tasks, which are summarized in Table 4.1. Each task is implemented as a separate program written in the C programming language. The core of the workflow, and the most computationally intensive task, is the computation of the matched filter signal-to-noise ratio and a time–frequency test, known as the χ^2 veto [10, 11]. There are several other components of the workflow, however, which we describe briefly here. A detailed description of the components may be found in [65].

Data from the three LIGO detectors must first be discovered and then split into blocks of length 2048 seconds for analysis by the inspiral program. The workflow uses the LSCdataFind program to discover the data and the methods of the Glue pipeline module described above to subdivide the data into blocks. For each block, and for each detector, a *template bank* must be generated for the matched filtering code. The template bank is a discrete subset of the continuous family of waveforms that belong to the parameter space. The placement of the templates in the bank is determined by the *mismatch* of the bank, which is the maximum fractional loss of signal-to-noise ratio that can occur by filtering a true signal with component masses m_1, m_2, with the "nearest" template waveform for a system with component masses m_1', m_2'. The construction of an appropriate template bank is discussed in [329, 330].

The bank is then read in by the inspiral program, which reads in the detector data and computes the output of the matched filter for each template in the bank. In the presence of a binary inspiral, the signal-to-noise ratio ρ of the matched filter will peak, as shown in Figure 4.6. The inspiral program may also compute the χ^2 time–frequency veto, which tests that the signal-to-noise ratio has been accumulated in a manner consistent with an inspiral signal and not as the result of a "glitch" or other transient in the detector data. If the

Table 4.1: The components of the inspiral analysis workflow.

| Component	Description
tmpltbank	Produces a bank of waveform parameters for use by the matched filtering code. The bank is chosen so that the loss of signal-to-noise ratio between a signal anywhere in the desired parameter space and the nearest point in the bank is less than some specified value, which is typically 3%.
inspiral	For each template in a bank, compute the matched filter and χ^2 veto algorithms on a given block of data. Generates a list of inspiral triggers, which are times when the matched filter signal-to-noise ratio and the value of the χ^2 veto exceed user-defined thresholds.
trigbank	Converts a list of triggers coming from the inspiral program into a template bank that is optimized to minimize the computational cost in a follow-up stage.
inca	Performs several tests for consistency between triggers produced by the inspiral program from analyzing data from two detectors.
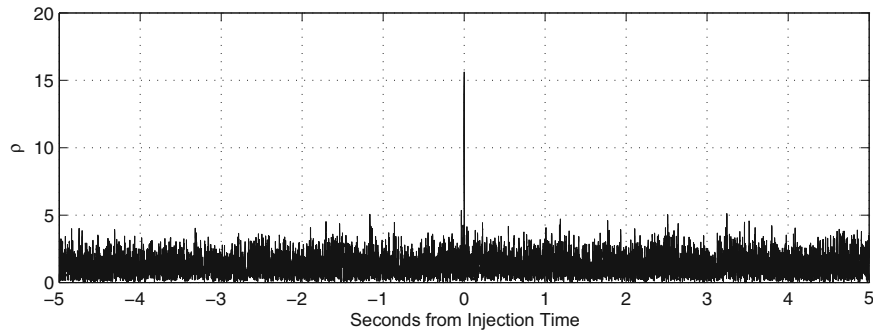
Figure 4.6: The output of the matched filter in the presence of a simulated signal. The signal is injected into the data at time $t = 0$. The signal-to-noise ratio generated by the filter peaks at the time of the injected signal.

value of the signal-to-noise and χ^2 veto pass defined thresholds at any given time, the inspiral code outputs a *trigger* for this time with the parameter of the template and filter output. These triggers must then be confronted with triggers from other detectors to look for coincidences.

The trigbank program can convert a list of triggers from the inspiral program into a template bank that is optimized to minimize the computational cost of a follow-up stage. We describe the optimization in detail in Section 4.5.2. The *in*spiral *c*oincidence *a*nalysis program, or inca, performs several

tests for consistency between triggers produced by inspiral output from analyzing data from two or more detectors and generates event candidates.

4.5.2 Inspiral Workflow Applications

The Second LIGO Science Run

In LIGO's second science run (S2), we performed a *triggered* search for primordial binary black holes and neutron stars [3, 4]. Since we require that a trigger occur simultaneously and consistently in at least two detectors located at different sites in order for it to be considered as a detection candidate, we save computational effort by analyzing data from the Livingston detector (the most sensitive detector at the time) first and then performing follow-up analyses of Hanford data only when specific triggers are found. We describe the tasks and their order of execution in this triggered search as our detection pipeline (workflow).

Figure 4.7 shows the workflow in terms of these basic tasks. Epochs of simultaneous Livingston–Hanford operation are processed differently depending on which interferometer combination is operating. Thus, there are several different sets of data: $L1 \cap (H1 \cup H2)$ is when the Livingston detector L1 is operating simultaneously with either the 4 km Hanford detector H1 or the 2 km Hanford detector H2 (or both)—this is all the data analyzed by the S2 inspiral analysis—while $L1 \cap H1$ is when L1 and H1 are both operating, $L1 \cap (H2 - H1)$ is when L1 and H2 but not H1 are operating, and $L1 \cap H1 \cap H2$ is when all three detectors are operating. A full L1 template bank is generated for the $L1 \cap (H1 \cup H2)$ data, and the L1 data are filtered with inspiral. Triggers resulting from these filter operations are then used to produce triggered banks for follow-up filtering of H1 and/or H2 data. However, if both H1 and H2 are operating, then filtering of H2 is suspended until coincident L1-H1 triggers are identified by inca. The workflow used to execute this pipeline is generated by a script called *inspiral_pipe*, which is written using the Glue library described in the previous section. The script is given the list of times suitable for analysis and generates a Condor DAG that is used to execute the pipeline. Figure 4.8 shows a small subset of the workflow created by the pipeline generation script.

The Fifth LIGO Science Run

As the complexity of the analysis pipeline increases and the amount of data to be analyzed grows, the size of the inspiral workflow increases also. To illustrate this, we give a brief description of the binary neutron star search in the fifth LIGO science run (S5). The S5 run is presently under way (as of April 2006) and will record a year of coincident data from the LIGO detectors. We will not describe the S5 inspiral pipeline in detail here, suffice it to say that the analysis uses a workflow topology different from that of the second
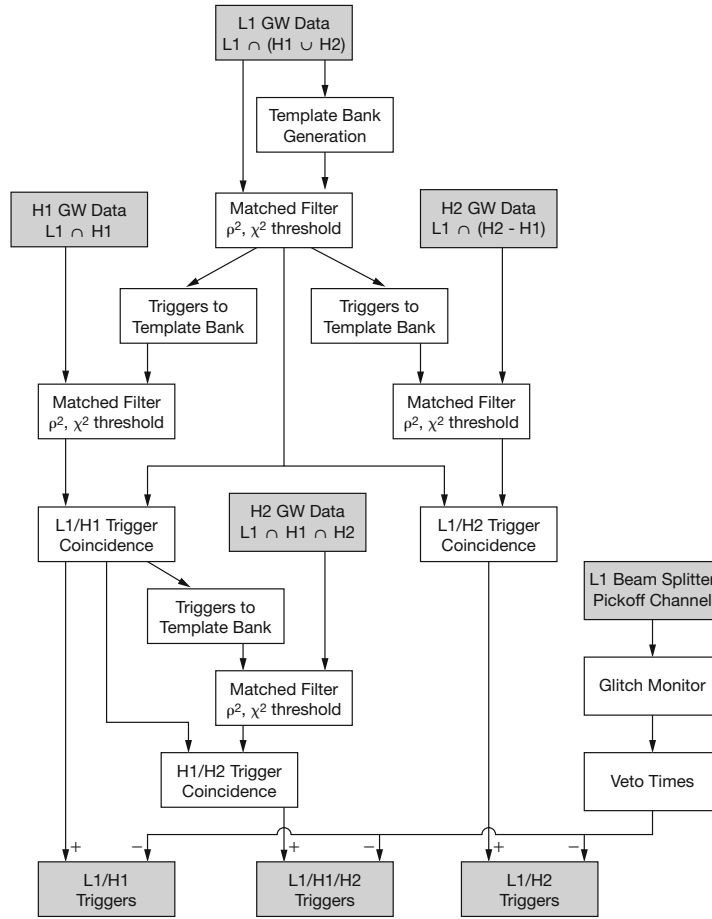
Figure 4.7: Structure of the S2 triggered search pipeline.

science run. To analyze a small subset of S5 consisting of 1564 hours of data for binary neutron star inspirals requires a workflow with 44537 nodes. To execute this workflow required 3000 CPU-days on the LIGO Caltech cluster, which consists of 1000 2.2 GHz dual-core AMD Opteron processors. A complete analysis of these data will require approximately 3–6 additional executions of the workflow.

4.5.3 Using Pegasus to Plan Inspiral Workflows

Since the inspiral pipeline workflows are produced using Glue, it is trivial to create Pegasus abstract DAX descriptions of the workflow (see Chapter 23). To run the inspiral analysis on the Penn State LSC cluster, which uses
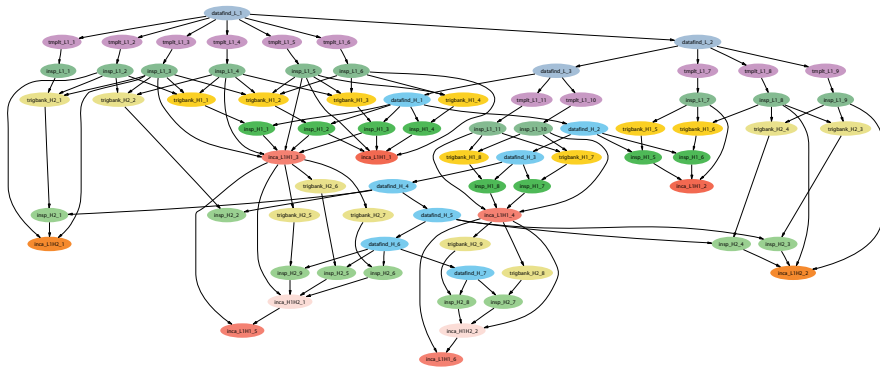
Figure 4.8: A subset of the workflow used to analyze data from the second LIGO science run for binary inspirals. The full workflow has 6986 nodes.

PBS as the scheduler rather than Condor, a DAX is created that describes the workflow. Using this method, we conducted a Monte Carlo based computation that analyzed 10% of the data from the fourth LIGO science run (S4), a total of 62 hours of data. The DAX created by the inspiral pipeline script contained 8040 nodes with 24082 LFNs listed as input files, 7582 LFNs listed as intermediate data products generated by the workflow, and 458 final data products. Once the DAX was planned by Pegasus, the executable concrete DAG used to execute the workflow had 12728 nodes, which included the jobs necessary to stage the input data to the remote cluster and transfer the output back to the user's local system. Execution of the workflow took 31 hours on the PSU cluster, described in Section 4.3.3.

Pegasus has also been used to parallelize inspiral workflows across multiple Grid sites. For a demonstration at the SC 2004 conference a typical LIGO inspiral analysis workflow was planned using Pegasus to run across the LSC Linux clusters at Caltech and UWM as well as a Linux cluster operated by the Center for Compuation and Technology at LSU. The effort demonstrated:

1. Running a LIGO inspiral analysis workflow internally within the LIGO Data Grid
2. Running a LIGO inspiral analysis workflow externally to the LIGO Data Grid on the LSU resource
3. Running across multiple types of cluster batch systems (Condor at Caltech and UWM and PBS at LSU)
4. Running at sites where LIGO data were prestaged using the LIGO Data Replicator (the LSC sites)
5. Running at sites where LIGO data needed to be staged to the compute resource as part of the workflow (the LSU Linux cluster)

All of the work planned by Pegasus and executed across the Grid sites ran to completion, and all of the output was staged back to the machine from which the workflow was launched.

4.6 Concluding Remarks

The workflow tools described in this chapter provide an extensible architecture for rapid workflow development and deployment and continue to be used and extended by the LIGO Scientific Collaboration. There are areas of the current framework that need to be strengthened, however, which we discuss in this section.

A key challenge is better integration of the pipeline development tools and workflow planning middleware. The LSC has successfully used the Pegasus workflow planner to leverage computing power at remote Grid sites, but there is still a substantial burden on the scientific end user to integrate this into the execution of a workflow. There is a need to develop the interfaces between data management, planning, and batch processing tools so that the use of large, distributed Grid computing resources appears to be as simple to the end user as submitting a DAG to a single LDG cluster running Condor.

Gravitational wave detectors generate large data sets that need to be accessed by various elements of the analysis workflows. In order to transparently execute jobs at remote locations, it is important to have seamless management of jobs and data transfer. In the work described above, Pegasus has been used to provide data staging to remote sites using GridFTP. Additional development will be needed to take advantage of Grid storage management technologies, such as dCache [109], and to accommodate any storage constraints that may be placed by non-LDG computing centers.

LIGO workflows also typically consist of a mixture of computationally intensive and short-running jobs. This information is not presently taken into account when planning a workflow. The Glue environment could be extended to provide additional job metadata to the workflow planner to allow it to make better use of available resources. For example, the user may only wish to run long-running jobs on remote Grid sites and execute short follow-up jobs locally. Furthermore, only minimal information about the Grid on which the workflow is to be executed is presently incorporated at the workflow planning stage. Metadata services need to be better integrated into the workflow design and implementation to allow efficient planning and execution.

Finally, the user interfaces to all of these computing resources must be simplified if they are to become truly powerful scientific tools. Users must easily be able to monitor the activity of their jobs using simple tools such as the Unix command `top`, they must be easily able to access their data products or input data sets, and they must be able to prototype and deploy application workflows with ease. From the perspective of the user—an application

scientist—quick and easy access to this information is of paramount importance.

Acknowledgments

5

Workflows in Pulsar Astronomy

John Brooke, Stephen Pickles, Paul Carr, and Michael Kramer

5.1 Introduction

In this chapter, we describe the development of methods that operate on the output of the signal of a radio telescope to detect the characteristic signals of pulsars. These signals are much weaker than the noise in the signal at any given wavelength, and therefore algorithms for combining the signals in different wavelength bands must be applied. This is heavily expensive in terms of CPU power. Early versions of distributed algorithms ran on a distributed network of supercomputers connected by Internet-aware Message Passing Interface (MPI) during the period 1999–2001. Today such techniques are being integrated into workflows that automate the search process and enable sophisticated astronomical knowledge to be captured via the construction of the workflow. In particular, we address issues of parallelism within components of the workflow. Parallelism is necessary due to two constraints on workflow performance. One is the application of the workflow in real time as the signal is being processed to enable very precise measurements to be carried out on known pulsars. The other is the use of the workflow to explore large regions of parameter space in search of previously undetected pulsars. There are very severe restraints on the degree of abstraction that can currently be applied in this work since details of the architecture of the computing resource (parallel cluster or computational Grid) on which the workflows are to be run *cannot* be ignored in the construction of the workflow.

5.2 Pulsars and Their Detection

Pulsars are rapidly rotating, highly magnetized neutron stars, emitting beams of radio waves (in the manner of a lighthouse) that permit the detection of characteristic, regularly spaced "pulse profiles" by radio telescopes on the Earth's surface (Figure 5.1). The fastest pulsars have rotational periods of only a few milliseconds and, as massive, essentially frictionless flywheels, make

superb natural clocks. These millisecond pulsars permit a wide variety of fundamental astrophysical and gravitational experiments. Examples include the study of neutron stars, the interstellar medium, and binary system evolution, and stringent tests of the predictions of general relativity and cosmology (see [274] for an overall description of pulsar astronomy).
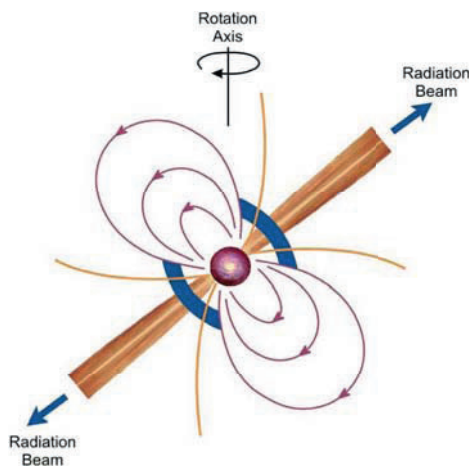

Figure 5.1: The radio beams of a pulsar. The highly condensed neutron star at the center has a powerful magnetic field, $O(10^{12})$ Gauss. Concentrated beams of electromagnetic radiation emerge at the magnetic poles. If the magnetic axis is not aligned with the rotation axis, they rotate like the beam of a lighthouse, giving a pulse of radiation as they cross the line of sight of the pulsar seen from the Earth.

The observed radio signal from pulsars manifests itself at the telescope as a periodic increase in broadband radio noise. In order to observe the pulse with a high signal-to-noise ratio, we need to observe across a wide band of radio frequencies. As the space between the Earth and the pulsar (the interstellar medium) is slightly charged, it is dispersive, and therefore different frequencies propagate at different velocities. The practical result of this effect is that the pulse is detected at the high-frequency end of the band before it arrives at the low-frequency end. If we do not correct for this propagation effect, the pulse profile is not observable, as it is broadened by this "dispersive delay." The amount of broadening a particular pulsar observation will display is directly related to our distance from the pulsar and the charge density along the signal path and is characterized by the "dispersion measure," or DM.

In addition, the sensitivity of radio telescopes means that the signal from the area of sky to which the antenna is pointing is contaminated with signals of terrestrial origin. These frequencies are not affected by the interstellar dispersion described above. Thus they can in principle be detected and eliminated.

The problem is that as more of the radio frequency spectrum is excised, the higher is the risk that signals from pulsars will also be lost. This means that we can have the situation (unusual in astronomy) that older observations may be better than newer ones, at least so far as the detection of new pulsars is concerned. Thus storing and reanalyzing the signal with improved algorithms for detecting pulsar signals and the use of increasing amounts of computing power are major factors in the search for pulsars. This makes the problem of great interest from the point of view of scientific workflows since there are several stages in the cleaning and processing of the recorded signal. For more details of the observational aspects of pulsar astronomy, see [270].

5.3 Workflow for Signal Processing

5.3.1 Astronomical Determinants of the Workflow

We summarize very briefly the important stages of a workflow for processing radio telescope signals used in searching for or observing pulsars. The works quoted in the previous section give fuller details and Jodrell Bank has collected information and software on its Web pages.[1] The signal gathered at the antenna of the telescope when pointed at a given region of the sky is known as a pointing. The pointing contains one or more beams whose signal has a range of frequencies with an upper limit given by the smoothness of its surface in relation to the wavelength of the radio waves (electromagnetic waves). If the surface is rough at the scale of a particular wavelength, it will be scattered rather than focused by the dish. The beam contains radiation from all signals that arrive in such a direction as to be focused at the antenna or are part of the noise intrinsic to the antenna. This needs processing in different ways according to what is observed.

   In Figure 5.2, we show a workflow for analyzing data that have previously been recorded from a radio telescope signal after some cleaning (to remove interference) and after digital sampling of the analog signal. We observe that there is a natural parallelism introduced into the workflow at different stages. Data stored from the radio telescope signal are extracted from a data archive. The data are divided into sections (currently separate files) representing a particular beam of radiation. Then several stages of processing may be applied to each beam, with multiple parallel processes for each stage. In observing distant sources, the effects of the interstellar conducting medium on the electromagnetic signals need to be compensated. These effects cause the signal velocity to depend on frequency, this is known as dispersion and it can be useful in eliminating terrestrial interference since this has zero dispersion. We may not know the dispersion a priori for unknown objects and thus have to apply a trial-and-error process. In Figure 5.2, different dispersion measures (DMs) are

---

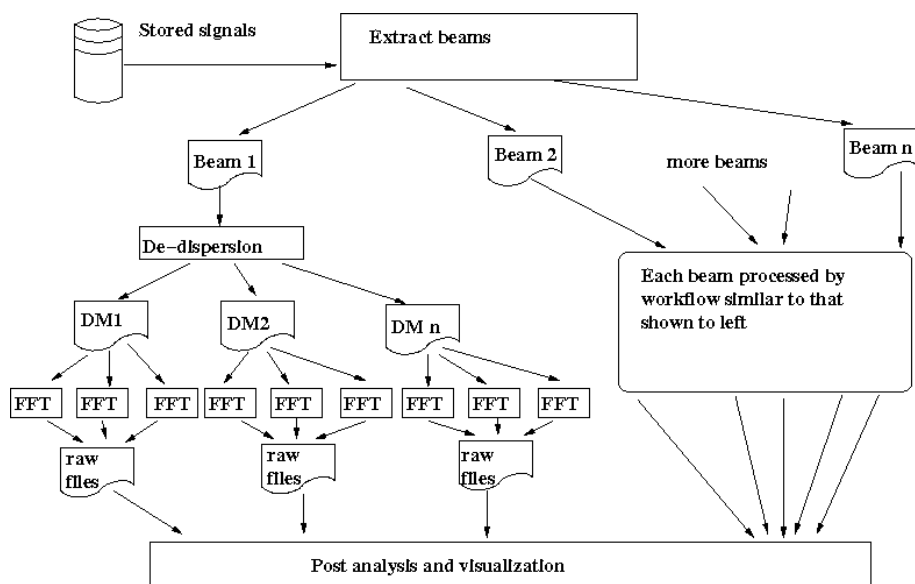[1] `http://www.jb.man.ac.uk/research/pulsar`

Figure 5.2: The inherent parallelism of a pulsar workflow. Data are stored in an archive as a collection of beams, each representing radiation from a particular direction in the sky. Dedispersion processing must be applied to each beam and, for each dedispersed beam, processing via Fourier transforms is carried out for a range of corrections for orbital acceleration. The raw results from each of these corrections for each dedispersed signal are then subject to post-workflow analysis. In the figure, full rectangles refer to stages in processing, and those with curved bottoms represent intermediate data sets.

applied to each beam. Then, for *each* dedispersed beam signal, multiple fast Fourier transforms are applied to represent various search parameters. For example, for pulsars in a binary star system, a correction in Fourier space needs to be applied to correct for the Doppler shifting of frequency caused by the orbital motion. However, since during the pulsar search the desired correction is not known ahead of time, a range of possible corrections need to be applied. To complicate things further, different search methods are applied for different types of orbits; e.g., those where the orbital period is long compared with the pulsar period, those where it is short, and those in between. Thus, without methods for intelligently exploiting the parallelism at different stages of the workflow, the flow of the data through to the eventual postprocessing stage, where potential candidates are examined by interactive and visual methods, can stall.

Our methodology in this chapter will be to examine in detail how parallelism is handled at the dedispersion stage of processing. We have detailed results for this stage and can present a general analysis for methods to estim-

ate the computational resources needed per unit of data. We consider that as workflows are increasingly applied to very large volumes of data, requiring large amounts of data processing, such quantitative analysis will become indispensable in the investigation of methods of workflow construction.

5.3.2 Coherent and Incoherent Methods of Dedispersion

In Figure 5.3, we see how dispersion caused the arrival time of the wavefront to be delayed by progressively longer times at different radio frequencies. In order to get a sufficiently strong signal, either to detect a new pulsar against background noise or else to determine very accurately the timing and shape of the pulse, we must sum across the radio frequencies. The simplest method to compensate for dispersion is to split the frequency band into independent frequency channels and apply appropriate time delays to each channel so that they all arrive at the output of the channel at the same time. In this process, knowledge of the phase of the voltage from the telescope is lost; hence the method is known as *incoherent* phase dispersion. The splitting into channels and the application of the time corrections was formerly done by hardware but now is increasingly being carried out by software on computer processors working in parallel. For very accurate measurements, such as timing of the
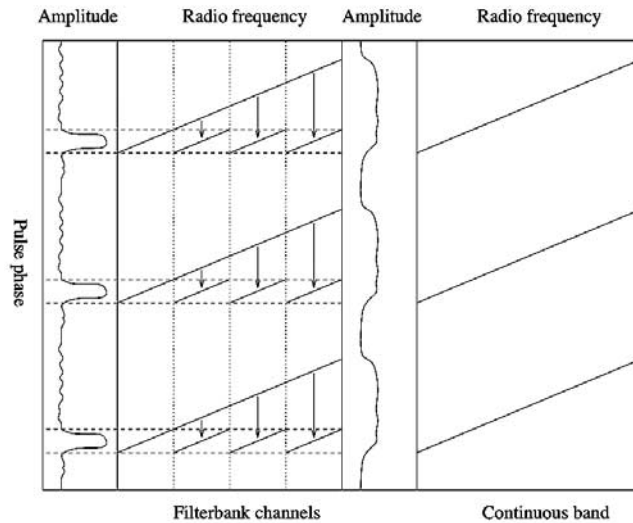

Figure 5.3: Pulse dispersion and the process of dedispersion. The radio frequency band is divided into channels, and time-delay corrections are applied to each channel. The flattened-out pulse of the original signal has a restored profile, and the signal-to-noise ratio is increased.

pulsar signals, incoherent dedispersion is insufficient. The method of *coherent* phase dispersion applies a phase-only filter. This is most simply applied in the frequency domain. Now the signal must undergo a Fourier transform, and then the application of the filter in frequency space before the inverse Fourier transform is applied to give the dedispersed signal in the time domain. This involves far more processing power if it is to be carried out by software. If the dedispersion must be applied in real time, or if a large number of trial dedispersion measures must be performed (in a situation where the real dedispersion is not known), there are severe constraints on the parallelism of the dedispersion component of the workflow.

Two major constraints come into play when considering how workflows should be parallelized. First, in real-time observation, the processing needs to keep pace with the signal capture. Second, in searching for new pulsars, especially those in binary orbits, where we have to search over considerable volumes of a multiparameter space, we need to obtain results in a reasonable amount of time. The methods of parallelism can be different in these two cases. We illustrate this by considering the coherent dedispersion stage in a signal-processing workflow both in the case of a parameter space search and in the real-time processing of a signal.

5.3.3 Workflow of the Search in Parameter Space

The workflow is parallelized across data sections; essentially it is a pipeline workflow. We proceed by first breaking up the data into segments, which have to be at least twice as long as the dispersive delay time across the observing bandwidth. Each segment is then subjected to a Fourier transform and multiplied by the Fourier transform of the inverse filter, and the resultant data length is then transformed back into the time domain and processed to produce a dedispersed time series. The next segment is then processed, and the resultant series are then concatenated together. In practical terms, the input data set only has to be forward transformed once but has to transformed back into the time domain as many times as we have trial DMs. One complication of dividing the time series in this manner and applying a Fourier transform is that we create boundary effects at the beginning and end of each segment, which appear as spurious frequencies in the transform. Thus we have to communicate duplicated parts of the time series at the start and end of each segment. These are called "wings." The size of these wings is given by the range of frequency bandwidth, and thus it is fixed by the observational signal. This means that if we divide the time series into small units for the purposes of parallelism (see Section 5.3.4), we have to communicate increasing proportions of repeated redundant data. In effect, we send more than the original data owing to this redundancy. In Grid applications where bandwidth may be a constraint, this can be a highly important restriction.

The result of the dedispersion is a time series for each trial dispersion measure. These time series are then subject to various analytical methods whose

aim is to determine the presence of periodic signals. This analysis produces a list of candidate periods, which may be pulsars or local radio interference or simply artifacts of the data collection hardware and subsequent software processing. These candidates can then be confirmed or rejected by further observations.

5.3.4 Work Distribution Model

The original intention was to adopt a master–slave model in which one (or more) master processes read the disk and distribute work to slave processes. This approach is highly attractive because it naturally furnishes an effective method of load–balancing. The main drawback is that, in the absence of complicated and time-consuming preventative steps, no one processor would get two contiguous chunks of the global time series. This would make more difficult the process of collecting the distributed data for the next stage in the workflow process, in which algorithms for the detection of periodic pulses are applied to each dedispersed signal. Instead, we adopt a client–server model. This illustrates how considerations of the total workflow affect the parallelism of a stage in the workflow process. A handful of server nodes poll requests for data from clients, and the bulk of the intelligence resides with the clients who do the actual work.

By allocating a single contiguous portion of the global data set to each worker, gaps occur only at a processor boundary, just as in the original non-distributed application. By letting each processor keep track of the portion of the global data set for which it is responsible, the impact on the logic and structure of the code is minimized. We give a diagrammatic representation of the client–server work distribution in Figure 5.4. Here the stages of the workflow run downward from the data-reading stage, and in the horizontal direction we show the parallelism produced by the "chunking" of the total pulsar observational data. The data are read in chunks by the server and sent to the clients as they request them. Since our execution environment is modeled as a small number of clusters with a large number of nodes, we observe that there is a pipeline effect with a start-up cost that is half the sum of the time needed to send data to each requesting processor in the cluster. Each processor then spends a certain amount of time processing before it requests its next chunk of work. The disadvantage of the client–server model adopted here is that we are compelled to tackle the load-balancing problem statically, taking into account *estimates* of the MPI bandwidth, processor speed, disk performance, and so on. The algorithm used is described below in Section 5.4. In general terms, it permits the analysis to be extended to clusters of differing numbers and types of host machines.
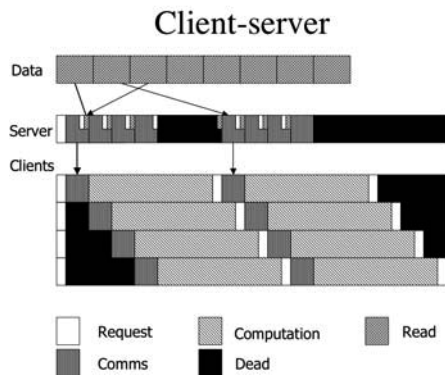
Figure 5.4: Client–server parallelism. The server process reads the data from the disk and the clients request as they finish processing each chunk of work. The key at the bottom indicates the processes occurring at each stage in the workflow, which proceeds from top to bottom for each chunk of the original series.

5.4 Use of Metacomputing in Dedispersion

5.4.1 Issues of Heterogeneity

Metacomputing refers to the running of applications in an environment where there is a significant degree of heterogeneity. The motivation for using meta-computing is that different stages of the workflow require different amounts of processing power. On a Grid, for example, different numbers of processors can be dynamically accessed to compensate for this imbalance and to keep data flowing through the workflow. Heterogeneous computing introduces some problems that are similar to those well known from cluster computing and some that are very specific [371].

We need to send data between the different computing resources that will be used to perform the dedispersion processing. This requires message-passing systems that can work between parallel machines and clusters as well as within such clusters. We used PACX-MPI [66], a version of MPI that allowed us to couple big systems into a single resource from the communication point of view. The layout of the metacomputing architecture used is shown in Figure 5.5. Essentially this is a hierarchical cluster with two levels of the hierarchy. At the *host* level, we have a tightly coupled cluster with many processors connected by a rich interconnect topology and with processor speeds and interconnect bandwidth rates being uniform within the host. The second level of the hierarchy is a serial interconnect between hosts. In the original work, the hosts were Cray T3E supercomputers with a three dimensional
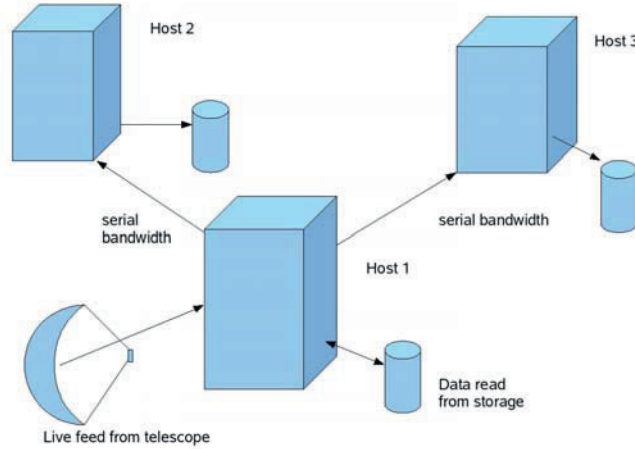
Figure 5.5: The metacomputing network for the pulsar search. Host 1 is close to either a live feed or (as was actually implemented) a signal stored on disk. For processing, data need to be sent to the remote nodes via serial Internet connections. Within each host the connectivity is much richer, with a 3D bidirectional torus in the case of the T3E machines used.

torus interconnect running at speeds on the order of 50 MB per second [354]. The serial interconnect was arranged in a star topology with connections of a maximum bandwidth on the order of 1 MB per second. Today we have Grids that have a similar hierarchical model but are connected via high-bandwidth dedicated links at speeds of 125 MB/s (1 Gbps network links).

5.4.2 The Algorithm for Parallelization of the Dedispersion

We now describe the algorithm that is represented graphically in Figure 5.4 to be run on a metacomputer of the star topology structure shown in Figure 5.5. Let N_h be the number of hosts, and let n_i be the number of client processors to use on host i, discounting the extra two processors required by PACX-MPI and those processors on host 1 that have been assigned to server duties. Referring to Figure 5.5, we have $N_h = 3$, and n_i had a maximum of 512 on each host in the experiments. Denote the bandwidth, in MB/s, from host 1 (where the input data reside) to host i by w_i. The rate, in MB/s, at which data can be read from disk on host 1 is denoted by r; it is assumed that this rate is approximately independent of the number of processors accessing the disk at any one time. The bandwidth within the hosts is assumed instantaneous since it is so much greater than w_i.

The size of one record is denoted by u, and this is determined by the need to secure phase coherence. The computational time required to process one record of data on host 1 is determined experimentally and denoted by t_1.

The time to process the same quantity of data on other hosts is estimated by multiplying t_1 by the ratio p_1/p_i, where p_i is the peak speed in Mflops of the processors on host i.[1] This approximation is justified in a metacomputer whose hosts have the same architecture. In a more heterogeneous architecture, processing speeds for the data would have to be determined by experiment on each host.

The amount of processing per record can be determined by the parameter N_s, which gives the number of dispersion slopes to be evaluated. t_1 is now to be reinterpreted as the average compute time per record per unit slope in the regime where N_s is large enough that the compute time per megabyte can be well approximated by $\tau \times N_s$.[2] We define a job as a task that processes a total
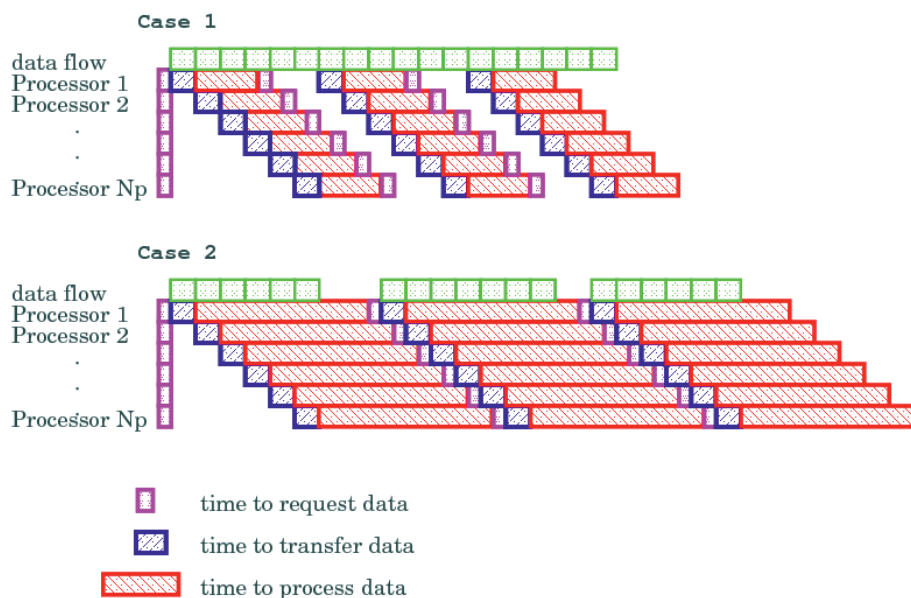
Figure 5.6: Data flow. In case 1, processors have dead time to wait for the next processing. In case 2, processors carry out the successive work quickly, without dead time.

of V records, and hence Vu MB of data. The load-balancing problem is to determine the proportion

---

[1] This assumes that all the processors on any given host are clocked at the same speed.

[2] The forward FFTs are computed only once, but the inverse FFTs must be computed for each slope.

$$v_1 : \ldots : v_n, \qquad \sum_{i=1}^{N_{\mathrm{h}}} v_i = V,$$

in which to distribute data to the hosts. Here V is fixed by the amount of data in the observations (the sum of all the "Read" rectangles in Figure 5.4), and v_i is the amount to be sent to each host.

Now the essence of our problem is that the remote hosts cannot process faster than they receive the data. From this we can see that the most advantageous situation would be where the metacomputing problem was mapped onto a single host since the intrahost bandwidth was over one hundred times larger than the interhost bandwidth (but see also Section 5.6). Our only justification for using the "Grid" approach would be if V and N_{s} were sufficiently large and/or if we require processing within a wall-clock time constraint T.

The elapsed wall-clock time t_i to process v_i records on host i is estimated by

$$t_i = v_i t_{\mathrm{proc}}\left(i\right)/n_i + n_i t_{\mathrm{wait}}\left(i\right), \tag{5.1}$$

where $t_{\mathrm{wait}}\left(i\right) = u\left(1/w_i + 1/r\right)$ is the time that a client processor has to wait for a single work unit. The time that it takes the client to process it is $t_{\mathrm{proc}}\left(i\right) = N_{\mathrm{s}}\tau p_1/p_i$. If we substitute this expression in (5.1), we have the processor performance and the number of processors, giving the total rate of processing for each unit record u.

Since we are using a pipelining algorithm, each node on the host starts up immediately after the previous one (we can use the MPI ordering for this). The time to get all the nodes processing is essentially half the time to send all of the data and is given by $t_{\mathrm{wait}}\left(i\right) = u(1/w_i + 1/r)$. The reason for this can be seen in Figure 5.6, where there is also a *run-down time* as each node stops working. If the diagonal "staircases" representing the start-up and run-down times are joined, they give a rectangle whose area represents $t_{\mathrm{wait}}\left(i\right)$, and hence each staircase is half the time to send the data. This term will be dominated by communications bandwidth on remote hosts and by disk access speed on the local host. In the original experiments, the latter was neglible, but as wide-area networks increase in speed, this will not always be a valid approximation.

The condition used to balance the workload is that all hosts finish at the same time. This is a sensible condition in a parallel algorithm and essentially states that all the hosts contribute equally to the speedup, $t_1 = t_2 = \ldots = t_{N_{\mathrm{h}}}$. Using these equations leads to a linear system with $N_{\mathrm{h}}+1$ equations and $N_{\mathrm{h}}+1$ unknowns $(v_1, \ldots, v_n, t)$.

$$\begin{pmatrix} a_1 & 0 & \cdots & 0 & -1 \\ 0 & a_2 & \cdots & 0 & -1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & a_{N_{\mathrm{h}}} & -1 \\ 1 & 1 & \cdots & 1 & 0 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_{N_{\mathrm{h}}} \\ t \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{N_{\mathrm{h}}} \\ v \end{pmatrix}.$$

Here $a_i = (t_1 p_1)/(n_i p_i)$ and $b_i = -n_i u \left(1/w_i + 1/r\right)$. The validity of this method depends on the implicit assumption that no client processor experiences dead time waiting for other clients to receive their data. A global condition that expresses this is the inequality $t_i > v_i u \left(1/w_i + 1/r\right)$. More carefully, we may define $t_{\mathrm{else}}\left(i\right) = \left(n_i - 1\right) t_{\mathrm{wait}}\left(i\right)$ as the time that it takes all the other processors on host i to receive their work units. Then the dead time $t_{\mathrm{dead}}\left(i\right)$ that is lost between work units is expressed as $t_{\mathrm{dead}}\left(i\right) = t_{\mathrm{else}}\left(i\right) - t_{\mathrm{proc}}\left(i\right)$ for $t_{\mathrm{else}}\left(i\right) > t_{\mathrm{proc}}\left(i\right)$ or else $t_{\mathrm{dead}}\left(i\right) = 0$. Figure 5.6 shows the relationship of time to request/transfer/process data and data flow. The processors have spent dead time waiting for the next processing unit in case 1 because N_{s} is too small. In case 2, N_{s} is sufficiently large that processors can do the successive work quickly without dead time.

A drawback of this approach, when coupled with the FIFO processing of requests by servers, is that in practice the run-down time is usually longer than the start-up time. Typically there will be some clients on a given host with one more work unit than other clients on the same host, but there is no guarantee that the more heavily loaded clients will be served before the others.

5.5 Workflows of Online Pulsar Searches

5.5.1 Real-Time Constraints

In 2002, Jodrell Bank acquired a 180 processor Beowulf cluster with ten dedicated data-capture cards each of which can receive a different frequency range of the broadband spectrum. The network bandwidth problem is now even easier to solve since the whole distributed metacomputer is realized within the cluster interconnect. However, the concept of the metacomputer still remains. Thus the work in analyzing the behavior of the distributed metacomputer provides a basis for a cluster-based solution. This is currently working at Jodrell under the name of COBRA (Coherent On-Line Baseband Receiver for Astronomy).

The total signal-processing workflow (Figure 5.7) currently being employed is of great interest from a computer science as well as an astronomical perspective. We see that the actual computational workflow running on COBRA is only a stage in a total workflow process involving multiple electronic and observational devices. The workflow itself is fed by data originating from a natural process of radio emission from distant objects. The signal-processing workflow has two branches. In the left-hand branch, the dedispersion stage is done via dedicated electronic components (hardware filterbanks and dedispersers). They represent the traditional method of observation when data would eventually be recorded onto tapes that were shipped elsewhere for computational processing. The right-hand branch represents the replacement of dedicated hardware by a Beowulf cluster (COBRA). The advantage of the

software method is that it can be reprogrammed to deal with different observational radio frequency windows and the processing power is available for other purposes when not used for observing. However, the requirement that COBRA replace dedicated hardware leads to an architecture for the cluster. Thus our metacomputing analysis for pulsar workflow processing remains applicable, only now we apply metacomputing solutions within a dedicated tightly coupled architecture. Computational Grids are in this sense generalizations of the metacomputing concept, with the added complexity of crossing administrative and security domains. The 180 COBRA processors are grouped in crates, and the first ten processors in crate 1 have access to the data-capture cards. The instantiation of the workflow must be aware of this underlying metacomputing architecture and must place data-reading tasks on the processors with the data-capture cards. There may also be considerations of efficiency in the grouping of data-processing components of the workflow to limit bottlenecks in message passing that can lead to overflows in buffers. Figure 5.8 is a conceptual diagram of the workflow process without such awareness. Each data server is associated with a process that receives data from a particular data-capture card, and other workflow tasks are placed arbitrarily on the COBRA processors. In actual instantiation, this leads to disorder in the topology of message passing, which has the consequence of message buffers overflowing and stalling the application. This breaks the total signal processing workflow, which needs to keep pace in real time with the data rate dictated by the telescope signal.

Based on early experiments with COBRA, a revised and ordered mapping of the workflow process onto the architecture was developed, and it is shown in Figure 5.9. As before, each data server is associated with a particular data-capture card, but now the other tasks in the workflow associated with its master–slave algorithm are placed on a dedicated group of physical processors. In effect, the machine is virtually partitioned to enact the workflow parallelism. Therefore, the engine that enacts the workflow must be able to handle logical numbering of the physical processors. This is exactly what the Message Passing Interface (MPI) does, and it is the underlying software used for the workflow. This is of some theoretical interest since MPI is not generally regarded as a workflow language; however, it has features that make the efficient mapping of signal processing workflows to metacomputing architectures possible. We return to this discussion later (Section 5.6).

5.5.2 Data-Processing Aspects of Online Processing

If the dedispersion method is to be used for real-time observations, our previous data-processing method of a scheduled service described in Section 5.4 cannot be applied. Instead, a hybrid of master–slave and pipelining is required. Figure 5.10 shows the essence of this revised workflow. Following the workflow from the top, we have the data being captured at a rate dictated by telescopic observation and subsequent electronic processing. The data are sent
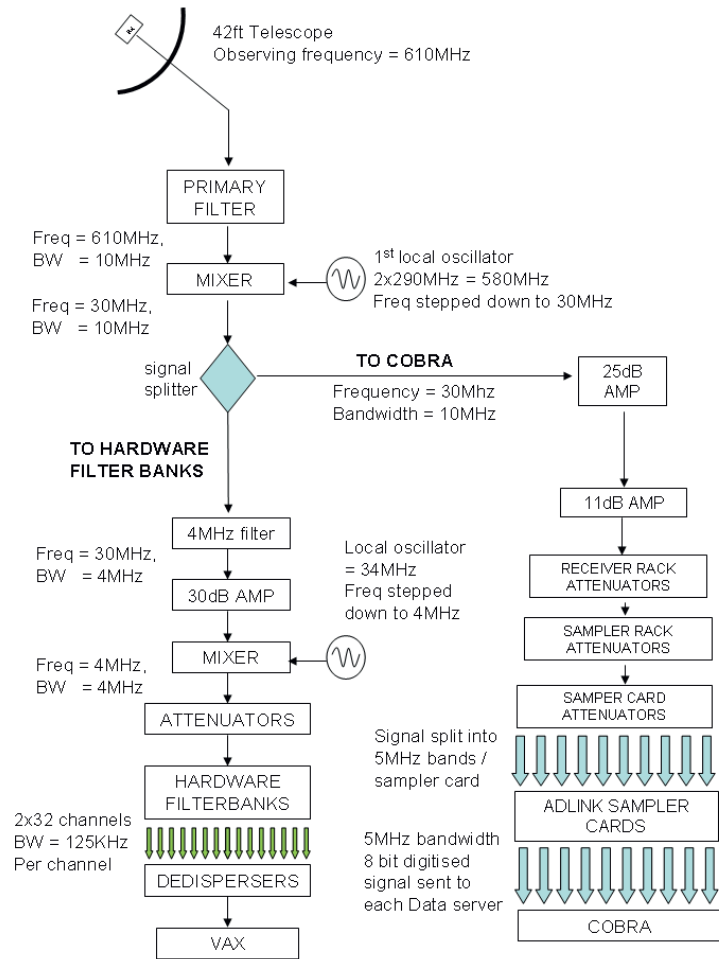
Figure 5.7: The signal processing workflow used in pulsar timing measurements on COBRA. The actual computational workflow is only a component in a workflow that includes multiple electronic devices fed by signals arriving from natural processes from distant astronomical objects.

to a particular master in one of the logical machine partitions of Figure 5.9. The master has a number of slave processors associated with it as well as other logical processes associated with the bookkeeping of the algorithm. The master has to send data to the subcollector associated with each data-processing slave in turn. We have an overhead of a start-up for all of the slave processes represented by the diagonal "staircase," which is the same as in the previous section. However, at the end of the dedispersion processing on each slave, the processed data are sent to one of a smaller number of processors that are doing
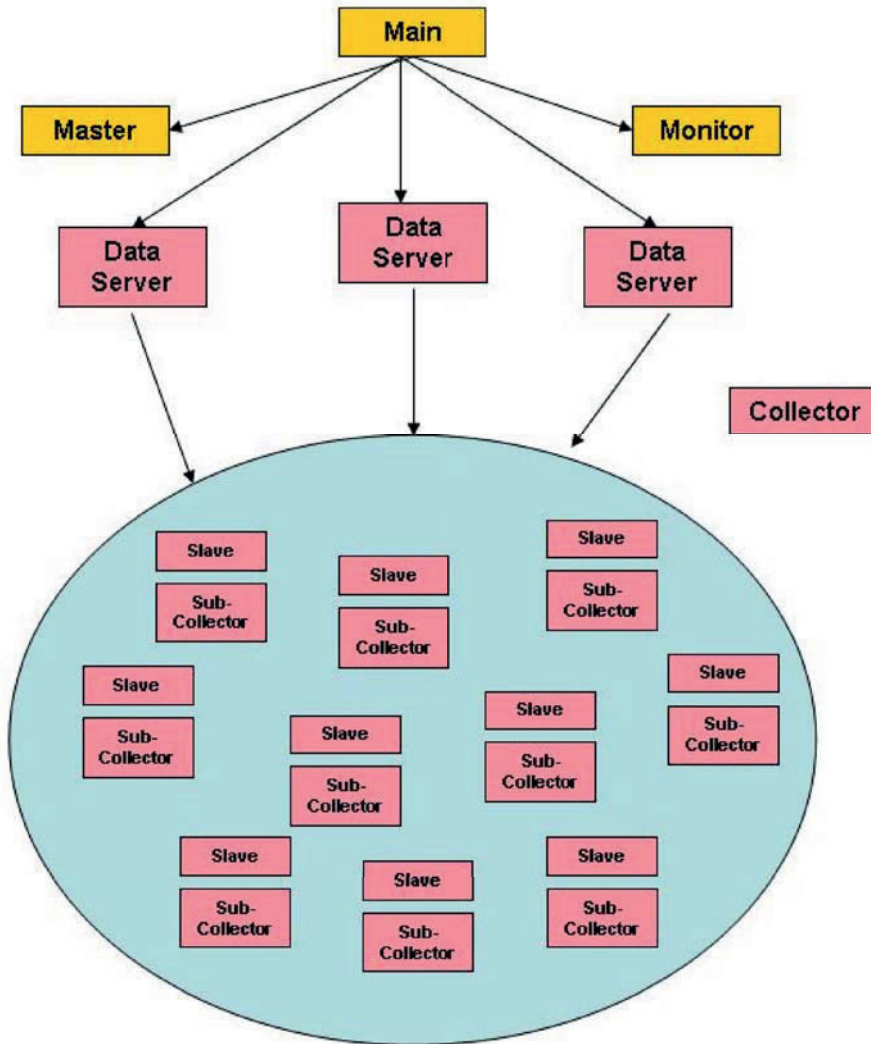
Figure 5.8: An unordered method of instantiating a COBRA workflow. The data servers placed on processors associated in the machine with data-capture cards distribute their data-processing tasks to other processors anywhere in the machine.

the postprocessing as the dedispersion is progressing. When this is done, the slave receives the next chunk of data for the application of the dedispersion measure.

In this model, we assume that the postprocessing work for a given unit of data requires less time than the dedispersion processing. Thus the post-
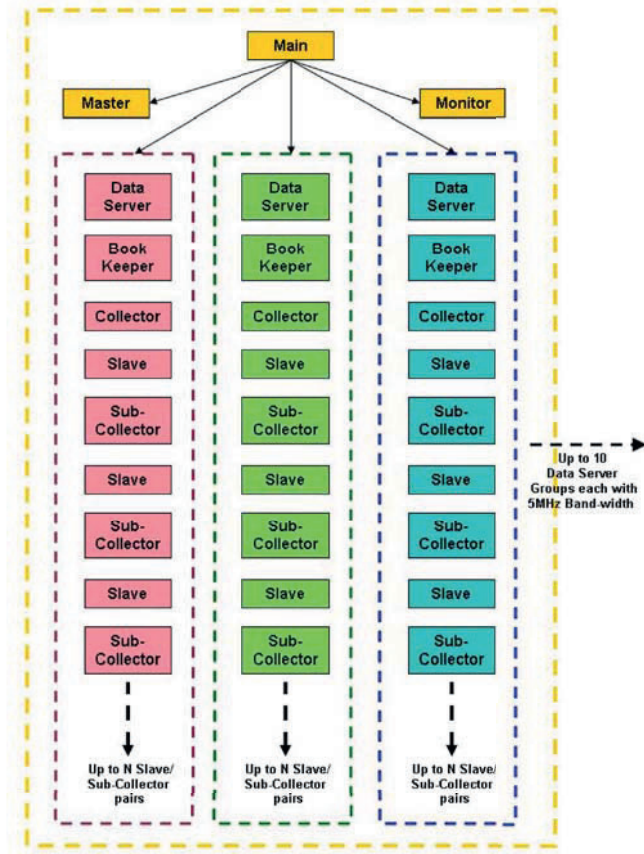
Figure 5.9: An ordered method of instantiating a COBRA workflow. Each data-server process now has a dedicated group of data-processing slaves placed on processors in ordered groups and only sends data processing to this physical group of processors.

processor nodes can receive data from several of the dedispersion slaves. The data that remain after the post-processing are similar in size to the original data since only one dedispersion measure is chosen. In the analysis presented in Section 5.4, there was an order *nslopes* as much data as the original data set. Clearly, it is not an effective use of bandwidth to send data corresponding to each slope back to the server to then be redistributed. Thus, even in offline processing, the postprocessing step must also be distributed.

In the online processing via COBRA, we can assume that the networks that carry the data have dedicated bandwidth. Thus the need for the algorithms to respond to differing bandwidths is no longer present, and the distributed
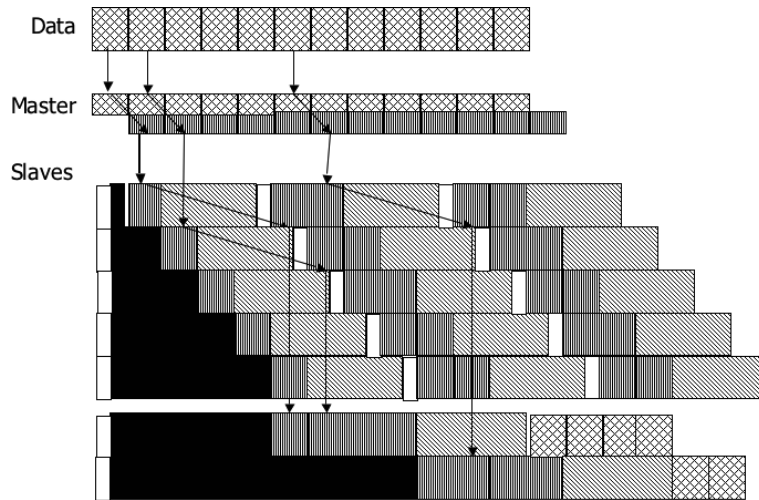
Figure 5.10: Master–slave parallelism for online processing of a radiotelescope signal. The postdedispersion steps of the processing are now specifically represented in the parallel workflow. This figure should be compared with Figure 5.4.

metacomputer can be balanced between capture nodes (master nodes), with a fixed number of slaves depending on the number of dedispersion slopes to be tested. Similarly, the number of postprocessing nodes can also be fixed in this case. This methodology is now being applied in real pulsar data processing involved in very precise timing measurements of known pulsars with known dispersion measures. Such timing measurements are vital in testing predictions of fundamental physical theories such as general relativity. Already the coherent dedispersion method has produced an increase in accuracy.

5.6 Future Work: Toward a Service-Oriented Approach

The GridOneD project[1] has been using Triana to process signals from gravitational wave detectors (see Chapter 20 for details of Triana). Since 2005, GridOneD has received funding to investigate the inclusion of search algorithms for processing pulsar data. A topic of current interest is the detection of pulsars in binary systems [75]. New methods have led to the most successful search for pulsars, using data from the Parkes radio telescope [139]. Modern Grids, such as the TeraGrid in the United States, the National Grid Service (NGS) in the United Kingdom, and DEISA in Europe,[2] have a hierachical structure to which our metacomputing analysis maps very well. The major

[1] GridOneD at http://www.gridoned.org
[2] http://www.teragrid.org, http://www.ngs.ac.uk, http://www.deisa.org .

difference is that the intercluster interconnect is now on the order of gigabits per second.

In all the work described here, the workflow processes have been enacted by methods not considered to be workflow languages in the conventional sense. Features of MPI have been successfully used to instantiate the workflows on metacomputing architectures that rely on either wide-area networks or tightly coupled interconnects. In GridOneD, we are now in the process of examining how the metacomputing approach could be updated to make use of current work on Grid middleware. A particularly useful feature of current Grid methodology is the adoption of a service-oriented architecture approach. By representing various stages of the data-processing workflow as services, we can hope to represent the workflow as a successive invocation of services. These services *must* be resource-aware since we are dealing with large amounts of data flowing through the workflow in parallel, and if resources are insufficient, overflows of several kinds will occur and the whole process will fail or stall. In this sense, the proposed WSRF, standard [321], where each service can be associated with resource properties meets our requirements. In this case, our resource requirements estimate for each stage in the workflow can be implemented via the resource properties.

We are currently exploring these issues in the context of the National Grid Service in the United Kingdom. This Grid has a structure similar to the metacomputer described in Section 5.4; namely, it has 4 clusters, with head nodes that are addressable from the Internet, each being a gateway to a much larger number of backend processors that can only be accessed via internal message passing. We have utilized a lightweight implementation of the WSRF standard, WSRF::Lite [483]. Being based on Perl rather than Java, this has a small footprint in terms of utilization of scarce resources (chiefly processor memory) for a general-purpose Grid hosting many users. Our approach is to have an overall master scheduling service implemented as part of the application that acts like the server in Figure 5.4. Each cluster can have its own local scheduler, with a queue of work tasks that are sent to worker processes on the backend nodes. With this approach, we can overcome the problem that access to the Grid resources is controlled by batch queuing methodologies that do not allow us to reserve resources on demand. Essentially, the workers are launched by the native batch queuing system and receive work units from the application-level queue. We mark the worker progress by milestones, and as these are attained messages are sent to the local manager, which can thus keep the master scheduler informed of the total progress in the data processing. In effect, this allows us to use the abstraction of a worker as the addition of a processor to the Grid, representing one of the horizonal strips of Figure 5.4. In our batch job request, we can indicate the number of workers to be created on behalf of the local manager. We are currently implementing this on the NGS. If successful, it will represent the transition of our methodology to Grids. We note that some of the more awkward features of the actual implementation are caused by the fact that WSRF is not yet a stable standard and that the NGS

(like many working Grids) is based on pre-SOA Grid middleware (Globus 2.4). Thus the SOA is created on top of non-SOA components.

This pragmatic compromise is essential since the pulsar astronomers wish to explore Grid processing of pulsar data and develop their signal-processing algorithms. For reasons of performance, it currently very difficult to employ workflow languages such as Triana, Kepler (Chapter 7, Taverna (Chapter 19), BPEL [24], etc. to orchestrate the pulsar workflows. Moreover, such languages are not designed to work within tightly coupled clusters such as COBRA. The ability of MPI to logically order and group the physical processors is absolutely essential for these types of architectures. Moreover, given the severe temporal constraints of real-time processing, a message-passing interface that has been specifically engineered for performance is critical. This is not the case with the workflow languages mentioned. Therefore, although not considered a workflow language, MPI can be used to implement a high-performance data-processing workflow, especially when this is integrated with electronic components in a hybrid digital–analog workflow process.

If we consider other types of Grids that are more peer-to-peer in nature (e.g., by aggregating spare cycles on machines used normally for other tasks), then the algorithms described here would not be appropriate. We consider resource utilization to be an important factor. Thus, in Section 5.3.4 we are concerned with minimizing dead time on expensive processing units. In P2P Grids, processor time is a resource that is cheap, and therefore the algorithmic constraints are much more concerned with maintaining coherence and using replicated computation to compensate for the unreliability of the processing hosts. The `einstein@home` [129] project is adopting this approach. It will be an interesting task for future work to gather and classify different search algorithms and relate these to the types of Grid or metacomputing environments to be used for each search. Our aim in this chapter has been to expose one such methodology in a way where its methods can be evaluated in a quantitative manner and mapped to the structure and dynamics of the Grid. We need to extend our work in terms of adaptation to variable processing rates since our previous work focused on dedicated processors. We also need to consider varying network bandwidths since we made the modeling assumption of constant bandwidth (using measurements that showed network turbulence timescales were short compared with data-processing timescales).

Acknowledgments

6

Workflow and Biodiversity e-Science

Andrew C. Jones

6.1 Introduction

Biodiversity e-Science is characterized by the use of a wide range of different kinds of data and by performing complex analyses on these data. In this chapter, we discuss the use of workflow systems to assist biodiversity researchers and consider how such systems can provide repeatability of experiments and other benefits. We argue that nevertheless there are also limitations to this kind of approach, and we discuss how more flexibility in a more exploratory environment could be achieved.

In the remainder of this chapter, we commence by describing the interrelationship between biodiversity and e-Science, contrasting biodiversity e-Science with other kinds of bioinformatics. Next we describe the BiodiversityWorld project, which is a major example of the use of workflows in biodiversity e-Science. The choice of BiodiversityWorld as the main example is partially due to the author's involvement in, and familiarity with, this project. But this chapter is not intended to be restricted to the requirements and achievements of BiodiversityWorld: In the following section, we discuss related work aimed at providing access to, and providing tools to manipulate, biodiversity resources. We then consider how a workflow-oriented environment might be extended in order to support more exploratory modes of use. We conclude with a summary and suggestions for future work.

6.2 Background: Biodiversity and e-Science

Biodiversity informatics differs considerably from bioinformatics, both in the kinds of data being used and typical tasks to be performed. In biodiversity research, it would ideally be possible for scientists to work collaboratively and simultaneously on research tasks, with support provided for "wet lab" experiments and for use of data from these and other sources in complex analyses.

Biodiversity has been defined as: "the variability among living organisms from all sources ... and the ecological complexes of which they are part: this includes diversity within species, between species, and of ecosystems" [83]. It follows that a scientist needs access to many different kinds of data when researching biodiversity-related phenomena. Examples include

- species catalogs (which include lists of species names and synonyms);
- species information sources (including species geography; distribution data comprised of individual specimen observations; descriptive data— both of individual specimens and of scientific groups such as species);
- geographical data (e.g., country boundaries); and
- climate data (e.g., maximum/minimum temperatures from various observation stations).

A significant problem at present is that typically a scientist may need to perform a number of distinct kinds of analyses using data such as we have enumerated above but will often need to perform these analyses using a number of distinct tools, manipulating the results of one analysis by hand before submitting them to another analysis process. (We shall give specific examples in the next section.) This is a major problem, particularly because in many cases the data standards are proprietary and incompatible. These difficulties have arisen because many of the data sets of interest were originally created for the use of an individual or small group. The data are designed for the original users' needs, perhaps with unusual data structures, representation, etc.

This is in contrast with more traditional bioinformatics research, in which significant standardization efforts have been made, leading to widely adopted standards for representing sequence data[1] and significant efforts to standardize metadata terms.[2]

Because of the diversity and breadth of data and tasks associated with biodiversity research, there is a need to support researchers with an integrated environment that minimizes the attention they need to give to manual, mundane tasks. In the next section, we shall see that BiodiversityWorld approaches this problem primarily by defining an interoperation environment in which heterogeneity is accommodated by wrapping and conversion software that allows the user to specify complex tasks as workflows.

---

[1] The EMBL/GenBank/DDBJ repositories are a good example of this.
  `http://www.ebi.ac.uk/embl/`.
[2] For example, The Gene Ontology.
  `http://www.geneontology.org/`.

6.3 BiodiversityWorld as an e-Biodiversity Environment

6.3.1 BiodiversityWorld Exemplars

The aim of BiodiversityWorld is to explore the design and creation of a problem-solving environment for global biodiversity. There is both a computer science and a biological aspect to the project: It was seen as important for the project to be biology-led so that the computing technologies developed would be designed very much with practical application in mind. Three exemplars were chosen on which to base our investigations:

1. *Bioclimatic and ecological niche modeling,* in which predictions are made about the suitability of the climate in a given region for the organisms of interest—either in present conditions or in hypothetical past or future conditions. This entails producing a *climate preference profile* by cross-referencing the known localities of a species with present-day climate data. This climatic preference is then used to locate other areas where a similar climate exists, indicating areas that are climatically suitable for the species. Present-day climate data may be used (e.g., to identify areas under threat from invasion by invasive species), or climate model predictions for either the future or the past may be used instead (e.g., to predict the possible effects of global climate change on the species distribution).
2. *Biodiversity modeling and the prioritization of conservation areas,* in which species distribution data are analyzed in order to produce a species richness map, which can then be used as a basis for proposing priority areas for biodiversity conservation.
3. *Phylogeny and paleoclimate modeling,* in which phylogenetic analysis and bioclimatic modeling are combined. The purpose of phylogenetic analysis is to reconstruct the most likely model of historical relationships among species and to use this to explore scenarios that have led to the diversity we see. This involves using DNA sequence data, and so at this point there is some overlap between our scenarios and tasks more typical of bioinformatics. Phylogenetic analysis generates large numbers of trees containing *taxa*[1] and their hypothesized relationships. The distinctive aspect of this part of the BiodiversityWorld research is that we gather distribution data for these taxa and fit climate models to each taxon. This allows explicit scientific interpretation of the role of climate in the development of biodiversity.

At present, tasks such as these require substantial manual work on the part of the scientist in preparing data sets, running stand-alone analysis tools, performing further data preparation, etc. This, combined with the fact that there are a good number of cases in which a standard analytic sequence can

---

[1] Such as species.

be defined, has led us to adopt a workflow-oriented approach in Biodiversity-World. It will be noted that exemplar (1) addresses a problem that is covered in more depth in Chapter 7.

6.3.2 Workflows in BiodiversityWorld

For each of the three exemplars chosen for BiodiversityWorld, it has been possible to devise a standard workflow for a single analysis, the variation in use of each workflow being in the choice of data sources and analytic tools for a given instance. Moreover, these workflows have a certain amount of commonality in the resources used and tasks performed: e.g., species distribution data and a "taxonomic verification" task are common to all three exemplars. The ease with which these workflows could be defined on paper, combined with the ease with which possible extensions and modifications could be identified, implied that a user interface based upon the workflow metaphor was a suitable starting point for the design of the BiodiversityWorld system. A simple example of a possible workflow extension is to "batch process" a group of related species instead of performing computation relating to each one individually.

In this section, we shall concentrate on one particular example—bioclimatic and ecological niche modeling. As explained earlier, the purpose of this task is to predict the suitability of climate in a given region for the organism of interest. Figure 6.1 illustrates in schematic form a typical workflow for this task. The task involves using records of where the species has been observed and combining this information with climate data to produce a model of climatic conditions that characterize these locations. To this end, we need:
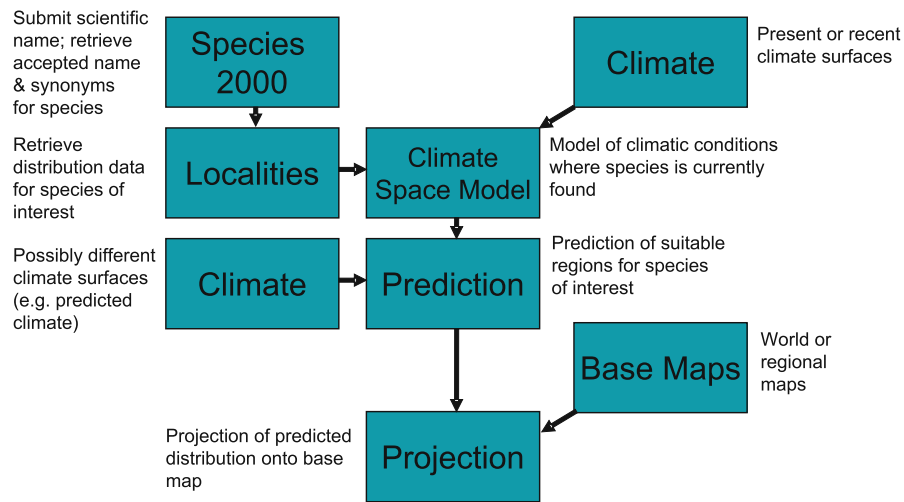

Figure 6.1: A BiodiversityWorld workflow.

- *A mechanism for specifying the species of interest.* Due to the nature of biological nomenclature, and differences of opinion among experts, more than one scientific name may be used to identify a given record. Thus, a catalogue of names, such as Species 2000[1], is used to retrieve synonyms [224].
- *A means of deriving a model relating to the climate and species data provided.* Various algorithms can be used, such as GARP [399] and CSM [372].

This model is then used to predict potential distribution, combining it with climate data to determine which geographical regions fall within the climatic model derived. This could be present-day climate data (to predict regions not currently occupied by the species but that might be able to sustain it) or historical or predicted future climates, as explained in Section 6.3.1. This can then be overlaid onto map data in order to produce a graphical representation of the predicted distribution.

Various workflow systems were considered as the basis for the Biodiversity-World user interface. Although many of these would have been reasonably suitable, the Triana (Chapter 20) system was selected primarily because of its attractive user interface and the direct access we have to Triana developers at Cardiff University.

Triana provides a means for categorizing resources hierarchically and composing them into workflows. Figure 6.2 illustrates a workflow that has been created using units from the palette displayed on the left-hand side to perform the task we have been describing above. It should be noted that in this realization of our conceptual workflow, climate space modeling, prediction and projection have been combined into a single unit in this example, using the same climate layers for modeling and prediction. Also, some additional units are needed, such as PopupStringInput, for user interaction. Figures 6.3 and 6.4 illustrate two stages in executing our workflow: selecting a species and displaying a map of predicted distribution. The GetMapFromDataCollection unit is included in consequence of the BiodiversityWorld architecture, which we shall describe below.

6.3.3 Triana and the BiodiversityWorld Architecture

The BiodiversityWorld architecture has been described in detail elsewhere [223]. For the purpose of this chapter, the main features of relevance are:

1. An abstraction layer has been defined, the *BiodiversityWorld-Grid Interface* (BGI), which defines an API that resources must implement in order to be usable in the BiodiversityWorld environment. DataCollection, referred to above, encapsulates data for communication between units: Units pack and unpack their data into and out of this representation to
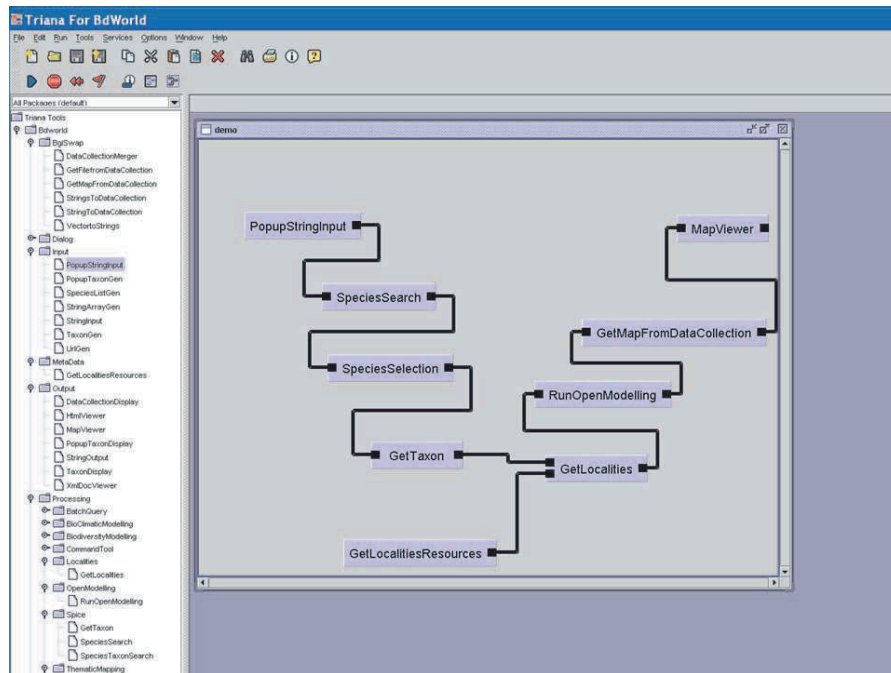
---

[1] `http://www.sp2000.org/`.

Figure 6.2: Triana/BiodiversityWorld environment.

reduce the knowledge that the middleware must have at a syntactical level about data types.

2. Initial implementations of this layer were proprietary in nature and illustrated the flexibility of this architecture for use with various kinds of Grid [145] and Grid-like middleware. Triana units were implemented that were able to communicate directly using the various BGI implementations.

3. More recently, we have concentrated upon providing Web and Grid services [325], for which the Triana Grid Application Toolkit is more directly suitable.

4. Performance of the BiodiversityWorld middleware has not been a major concern because interoperability has been seen as more important than high throughput for many of our tasks. Nevertheless, we are currently exploring the use of Condor pools [262] for some of the more data-intensive tasks within our workflows. For example, for ecological niche modeling, we have recently performed 1700 modeling jobs over a period of 52 hours, with data sets of the order of seven MB being used in each job, using our existing architecture. It would be desirable to perform these jobs much faster or over a larger number of iterations: This is the main motivation for our interest in Condor.
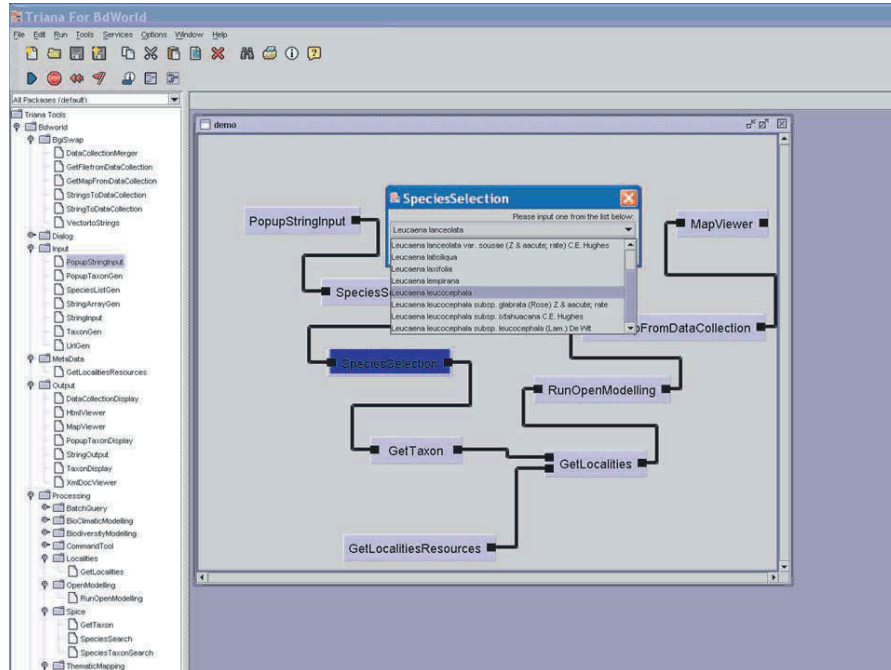
Figure 6.3: Selecting a species.

Thus it has been possible to use Triana as a front end to the BiodiversityWorld system with reasonably little effort. Nevertheless, there are some refinements that would be desirable, as we shall discuss further in Section 6.5.

6.4 Related Work

A number of other projects are using workflows for application areas related to our own. In this section, we briefly outline two of the most important ones: SEEK[1] and myGrid[2].

Among other things, the SEEK project aims to support acquisition, integration and analysis of ecological and biodiversity data. The aims of SEEK therefore overlap with those of BiodiversityWorld, but SEEK has used the Kepler workflow system [19] and concentrated particularly on some specific issues that, due to limited resources, we have not been able to give much attention to within BiodiversityWorld. One of the most notable of these issues is semantic mediation [60]: Techniques are being developed to support automated transformation of data and analytical components within a workflow to

---

[1] `http://seek.ecoinformatics.org`. See Chapter 7.

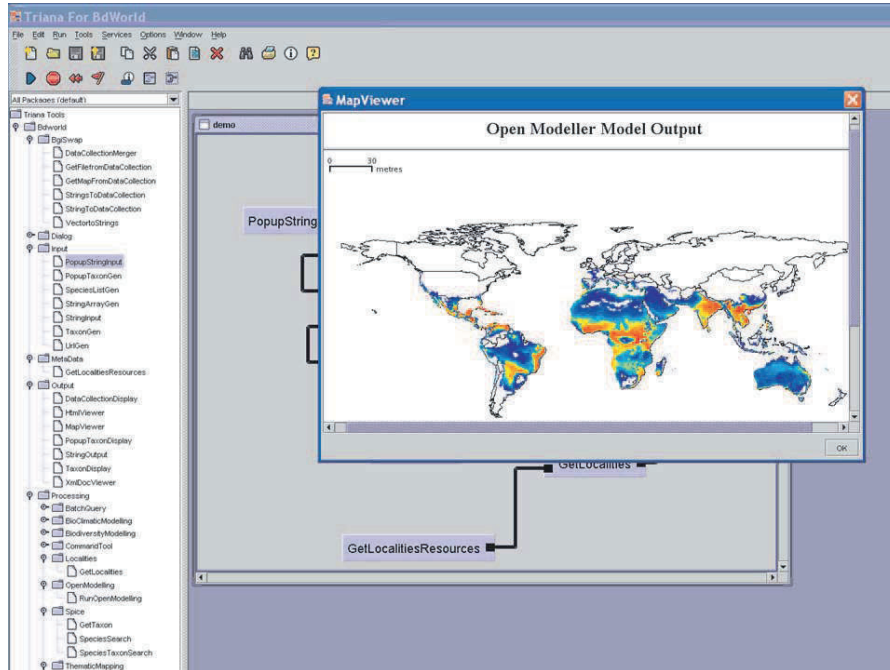[2] `http://www.mygrid.org.uk/`. See Chapter 19.

Figure 6.4: Display of results.

provide compatibility between workflow elements. In contrast, in Biodiversity-World, transformation units must at present be manually incorporated into workflows. However, there is a metadata repository that provides the information needed to determine the nature of the transformation needed.

The myGrid [396] project aims to support more general bioinformatics requirements, providing an environment for data and application integration. As in the SEEK project, particular attention is being given to a number of important issues that arise: In the case of myGrid, these include provenance, personalization, and change notification. The Taverna [326] tool is being developed to support the creation and enactment of workflows. Careful consideration was given to the possibility of using Taverna within the BiodiversityWorld system. One of the main reasons that it was not adopted was because it provided a lower-level view of workflow composition than we considered appropriate for our needs; for example, in the version we evaluated, separate units were required to represent starting and finishing a given process. The less finely grained approach provided by Triana more closely met our understanding of the users' concept of a workflow.

A distinctive feature of BiodiversityWorld, in comparison with the projects above, is our creation of a middleware architecture that is intended to be insulated from the underlying Grid software. This was important at the

time the BiodiversityWorld project commenced, as Grid software was evolving rapidly. With the advent of Grid services and WSRF [461], this concern is perhaps not as serious now as it was when we commenced the project.

6.5 Toward an Exploratory Workflow Environment

Current workflow-based approaches to e-Science are proving to be of value for biodiversity science and other related disciplines. However, the simple approach that is currently typical has serious limitations: A designer may be provided with little more than a palette of units/actors from which to compose a workflow, perhaps aided by some resource discovery tools. These limitations are particularly in the areas of

- resource discovery,
- reuse of workflows, and
- exploratory experimentation.

The first two of these limitations are addressed, to a greater or lesser extent, in existing systems. For example, a prototype "intelligent unit" has been implemented for BiodiversityWorld: This can be queried for resources matching criteria supplied by the user. In regard to the second limitation, some systems maintain metadata relating to specific workflow enactments and some—notably Kepler—allow "smart reruns" in which modified workflows are not necessarily fully executed: Results from previous runs are used wherever possible.

The problem with even a reasonably sophisticated workflow environment, incorporating features such as those listed above, is that workflow construction requires a level of planning that may restrict the user's freedom to explore readily, find information of interest, perform tentative analyses, etc. This is in sharp contrast with widely used software such as Microsoft Internet Explorer, which allows users to navigate freely, maintaining a history of sites visited, and provides a facility (albeit primitive) to search previously used pages for keywords of interest.

In an exploratory environment of this kind, if all interactions were logged transparently and automatically annotated with appropriate metadata, mechanisms could be devised to find and reuse ephemeral workflow fragments as parts of a larger task. This mechanism could be supported by a knowledge-based system to assist in selecting resources and workflow fragments, anticipating ways in which the user might wish to recombine them. Three simple scenarios that could serve as a partial specification for this environment are:

1. The user runs the same task on multiple data sets and selects those having interesting derived properties. For example, a set of tests may be performed on a large number of databases and the user then selects those having certain properties as a base set for use in subsequent analyses. For

this we need, at the very least, to be able to store a list of data sets on completion of the tests. Preferably the user would also be able to browse back through the history of interactions and do a filtered search of this history.

2. The user does a sequence of operations manually (not by constructing a workflow) and then wants to backtrack and try some alternatives. Having found paths of particular interest among those explored, the user then wishes to construct one or more workflows for reuse, possibly in modified form. To support this, an automatically constructed graph of alternative dataflows could be provided from which the user could select the parts of interest.

3. The user wishes to generalize a specific workflow. For example, a knowledge-based system could allow a user to replace a very specific task such as *Run version 3.14159 of phylogenetic tree-generating tool X* by (for example) *Select and run a phylogenetic tree-generating tool.*

These scenarios are clearly not fully specified at present. We have merely tried to present our vision of how a workflow-based environment could be enhanced to support more exploratory modes of interaction. We suggest that the most significant difficulties that need to be overcome if an environment of this sort is to be created successfully are as follows:

- Design of a suitable user interface, and trialing prototypes with suitable users
- Automating the generation of suitable log metadata—any significant involvement of the user in annotating his or her actions is a potential distraction from the experimental approach we are arguing should be supported (although perhaps some retrospective annotation may be useful)
- Related to the above, the design of suitable metadata and inference mechanisms to support exploration, deduce appropriate workflow generalizations, etc.

6.6 Conclusions

We have illustrated how biodiversity e-Science can be supported by the use of workflows, discussing particularly their use within the BiodiversityWorld project. Although the workflow metaphor is a powerful one in this context, we have explained our concern that a scientist's creativity may be potentially hindered by the workflow design/enactment cycle, and we have discussed ways in which more flexibility could be introduced. To explore these ideas, we would like to augment Triana with a browser-like mode, supporting exploration of data sets and performance of individual tasks and incorporating logging, replay, and automated workflow-construction features.

Acknowledgments

7

Ecological Niche Modeling Using the Kepler Workflow System

Deana D. Pennington, Dan Higgins, A. Townsend Peterson,
Matthew B. Jones, Bertram Ludäscher, and Shawn Bowers

7.1 Introduction

Changes in biodiversity have been linked to variations in climate and human activities [295]. These changes have implications for a wide range of socially relevant processes, including the spread of infectious disease, invasive species dynamics, and vegetation productivity [27, 70, 203, 291, 294, 376, 426]. Our understanding of biodiversity patterns and processes through space and time, scaling from genes to continents, is limited by our ability to analyze and synthesize multidimensional data effectively from sources as wide-ranging as field and laboratory experiments, satellite imagery, and simulation models.

Because of the range of data types used, biodiversity analyses typically combine multiple computing environments: statistical, mathematical, visualization, and geographic information systems (GIS), as well as application-specific code that may be written in any programming language. A mix of proprietary and open-source software is typically cobbled together by manual, scripted, and programmed procedures that may or may not be well designed, documented, and repeatable. Legacy FORTRAN programs written decades ago, as well as more recent C/C++ programs are commonly modified and used, and Unix scripts abound. The details from the entire range of analyses conducted are either unavailable or hidden within complex code that combines many tasks and is not robust to alternative uses without comprehensive user knowledge of the code. Some procedures are computationally intensive, but parallelized approaches are not in widespread use for lack of access to high-end computing resources and lack of knowledge about how to make use of those resources.

Hence, challenges in biodiversity analyses include data-intensive, computation-intensive, and knowledge-intensive components. Scientific workflows in general and the Kepler Workflow System in particular [19, 20, 272] provide an opportunity to address many of these challenges. Here we examine the details of a specific analysis within Kepler to illustrate the challenges, workflow solutions, and future needs of biodiversity analyses. The example

Table 7.1: Challenges from ecological niche modeling and workflow solutions

Challenge	Workflow Solution
Model complexity	Hierarchical decomposition
Exploratory modeling	Modular components for models can be substituted
Distributed data	Integrated data access via EcoGrid
Heterogeneous data	Rich transformation components (including spatial operations) and emerging semantically based data-integration tools
Computational intensity	Support for Grid computing (e.g., Nimrod) and emerging peer-to-peer support

analysis is drawn from a general approach called ecological niche modeling [391], which has a number of technical challenges relevant to scientific workflow solutions that are summarized in Table 7.1. Analyses are complex, incorporating many computational steps in diverse software environments. Within a given segment of the analysis, multiple approaches may be used, sometimes in tandem for comparison between approaches. Hence, the same analysis may be conducted with some variation many times. Input data are drawn from a variety of distributed sources and represent different data categories: observational data from the field, derived data from digital elevation models, and simulation output, each of which has its own semantics. These characteristics lend themselves readily to workflow approaches.

In the following sections, we briefly review ecological niche modeling from the domain perspective and then address each of the challenges and workflow solutions listed in Table 7.1 in detail.

7.2 Approaches in Ecological Niche Modeling

Ecological niche modeling is a conceptual framework for understanding and anticipating geographic and ecological phenomena related to biodiversity [391]. The ecological niche of a species can be defined as the conjunction of conditions within which it can maintain populations without input via immigration [177, 178]. Extensive research by diverse investigators has built the case that niches can be estimated based on associations between known geographic occurrences of species and features of landscapes summarized in digital GIS data layers [28, 205, 340, 345] (see Figure 7.1a).

The ability to predict ecological and geographic phenomena using ecological niche modeling generates many opportunities for investigators. The simplest applications, of course, are those of characterizing distributions of species in ecological space, which offers a view of the ecological requirements of species [99] (Figure 7.1a). A second level of prediction comes from projecting the ecological model onto geographic space to interpolate a predicted potential
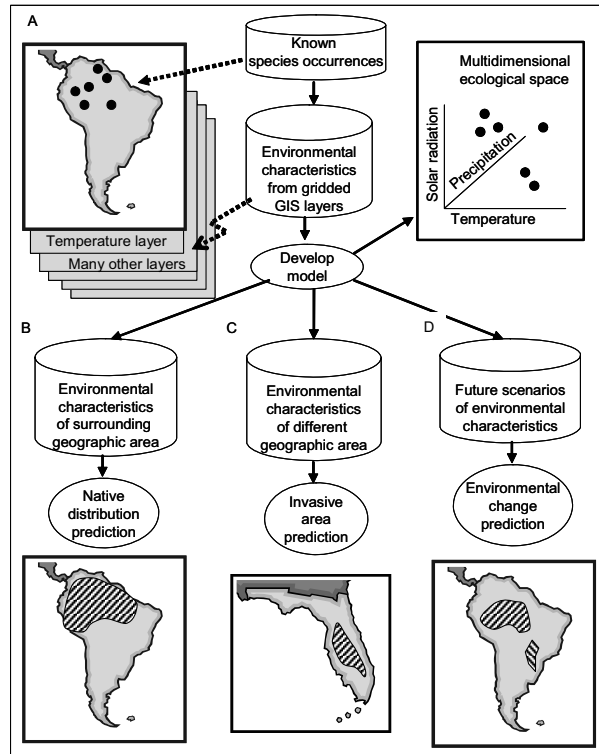
Figure 7.1: Ecological niche modeling approach and applications: (a) species' occurrence data are analyzed with environmental data to develop a model of the species' occurrence in ecological space; (b) the model is applied across geographic space to predict the spatial distribution of suitable habitat on the species' native range; (c) the model is used to predict suitable habitat in a new distributional area in the case of species' invasions; and (d) the model is applied against changed environmental conditions, such as climate change, to predict the distribution of suitable habitat under the new conditions.

geographic distribution—effectively filling gaps in knowledge between known sampling locations to provide a more complete picture of a species' geographic distribution [79, 253, 313, 346] (Figure 7.1b).

Further levels of prediction results under the assumption of conservative evolution of ecological niches. This assumption now has considerable support, both from theoretical considerations [196, 197] and from empirical evidence [204, 280, 349]. If ecological niches change only rarely and under limited circumstances, then the ecological potential of species remains relatively constant even under scenarios of change, e.g., transplantation to another continent as invasive species (Figure 7.1c), or distributions of species in changing

climates (Figure 7.1d). Ecological niche modeling has seen extensive application to these situations. Numerous studies [43,190,345,350] have confirmed the predictability of potential distributions as invasive species on other continents or in other regions as well as potential distributional shifts under scenarios of climate change [28, 281]. As such, the suite of situations in which ecological niche modeling is informative is quite broad.

Numerous conceptual approaches and software tools can be used in ecological niche modeling. In the simplest sense, an ecological niche model is just a description of the ecological conditions present across a specie's range [177, 178], and as such some very simple tools have seen very broad application [313]. Beyond these simplest tools, however, a number of improvements have been made—first, a suite of methodologies improved on the simple range rule approach [313] to develop more flexible depictions of species' ecological niches [79, 459].

Further developments of niche modeling tools proceeded along two main lines: (1) multivariate statistical tools beginning with logistic regression [297] and progressing through generalized linear and generalized additive models [131]; and (2) evolutionary computing applications such as genetic algorithms [399], neural networks [340], and maximum entropy approaches [353]. Each of these two classes has its advantages and disadvantages for niche modeling, but the basic message is that many computational options exist for modeling ecological niches.

Many recent studies have addressed likely effects of global climate change on distributions of species. The general approach is one of modeling and validation of basic model predictions based on present-day ecological and geographic distributions of species and then projection of niche-model rule sets onto future changed climate conditions drawn from general circulation models of global climates [361]. Although the number of studies using this approach is large—see a recent review and meta-analysis [426]—most have been limited by practical and technical limitations to between a few dozen and a few hundred species. The largest such study to date [348] reviewed approximately 1800 species of Mexican birds, mammals, and butterflies.

We are conducting a prototype project using the Kepler Workflow System designed both to demonstrate the power of scientific workflows in solving large-scale computational problems and to shed light on a still-open question: What is the magnitude of likely climate change effects on biodiversity across the Americas? We are using the data resources of the distributed Mammal Networked Information System (MaNIS) [394] to carry out a review of likely climate change effects on the over 2000 mammal species of the Americas, constructing maps of potential species distributions under future climate scenarios. Not only will this analysis be the broadest in taxonomic and geographic scope carried out to date, but the computational approach involved (the workflow) will be completely scalable and extensible to any region and any suite of taxa of interest.

7.3 Data Access via EcoGrid

In Kepler, distributed data access is provided through the set of EcoGrid interfaces [225,343]. EcoGrid allows data and computation nodes to interoperate through a standardized high-level programmatic API. Resources are added to the EcoGrid through a distributed registry. The registry is also used to locate resources and to choose among alternative versions when they exist.

The ENM (Ecological Niche Modeling) workflow uses data from three sources on the EcoGrid: (a) mammal occurrence data from MaNIS, (b) modeled present and future climate data from the Intergovernmental Panel on Climate Change[1] (IPCC), and (c) Hydro-1k digital elevation data from the U.S. Geological Survey.[2] MaNIS consists of a consortium of 17 North American mammal collections developed using the Distributed Generic Information Retrieval (DiGIR) protocol, an open source client/server protocol for retrieving distributed information using HTTP, XML, and Universal Description, Discovery, and Integration (UDDI).[3] MaNIS outputs mammal point occurrence data in the form of tables of species name and requested attributes, which include longitude and latitude. IPCC provides gridded global maps of present and future climate data predicted using a number of different climate change models. Data include cloud cover, diurnal temperature range, ground frost frequency, maximum annual monthly temperature, minimum annual monthly temperature, precipitation, radiance, vapor pressure, wet day frequency, and wind speed. The present-day data are available worldwide with a resolution of $0.5°$. Future modeled climate predictions have variable resolution but considerably lower resolution than historical data. Hydro-1k data were developed by the USGS EROS Data Center.[4] These spatial grids were created using the $30''$ digital elevation model (DEM) of the world (GTOPO30), recently released by the USGS, and provide a standard suite of georeferenced data sets at a resolution of 1 km. Hydro-1k data include such derived features of landscapes as aspect, slope, and elevation, with the data divided by continent. Total data size is roughly 10 GB.

7.4 Hierarchical Decomposition of the ENM Workflow

The ENM conceptual workflow is divided logically into three separate parts (Figure 7.2): (1) data preparation; (2) model construction, including prediction on the environmental layers used to construct the model; and (3) application of the model to changed climate conditions and comparison of model output. These three parts are captured within Kepler as a set of hierarchical,

---

[1] http://www.ipcc.ch/

[2] http://edcdaac.usgs.gov/gtopo30/hydro/

[3] http://digir.net

[4] http://lpdaac.usgs.gov/gtopo30/hydro/readme.asp

nested subworkflows (Figure 7.3). Subworkflows are used to wrap the functionality of multiple components that form logical groupings. The three parts of the conceptual workflow are captured by six subworkflows, four of which are necessary just for the first part.

7.4.1 Data Preparation

Data preprocessing and transformation (Figure 7.2a) is incorporated into four subworkflows within Kepler (Figure 7.3: subworkflows I through IV): (1) Create Species Occurrence List, (2) Create Spatial Data Layers, (3) Create Convex Hull Mask, and (4) Revise Spatial Layers. This portion of the workflow includes analytical components carried out by the EcoGrid query interface, Geographic Information System (GIS) processing, Java and C++ programs, and statistical functionality provided by the open-source R package. The data are manipulated into compatible formats for integration, including restructuring and rescaling the data and changing their syntax (Figure 7.2a). The MaNIS occurrence points are used to construct a buffered convex hull around the area of known occurrence; areas outside of this are masked out during the model training phase.

7.4.2 Model Development

Data sampling, division into training and testing sets, model training, and model testing (Figure 7.2b) occur within the Calculate Rulesets composite actor (Figure 7.3 subworkflow V). Each known species occurrence from MaNIS is used to query the climate and topographic data sets at that location (Figure 7.2b). Sampled data are divided into two sets, one of which is used to train the algorithm used to model the data and the other of which is used to test the predictive model generated by the algorithm and calculate the predictive error. For the ENM workflow, we are using the Genetic Algorithm for Rule-set Production (GARP) model, developed specifically for ecological niche modeling applications [398, 399]. GARP is a stochastic model, so each run generates a different result. For each species, GARP is run many times (typically 100 to 1000, averaging 10 to 20 seconds per iteration), predictions are made for each run, and the distribution of model error results is used to select the best subset of models. Models with high omission error (those that fail to predict known presence points) are excluded, either through a hard threshold (e.g., omission error $< 10\%$) or as a soft threshold (e.g., the 10% of models with the lowest omission errors). Models passing this first filter will range from very small to very large areas, each of which are predicted present. Because areas of overprediction are difficult to detect with presence-only data, indirect methods are used to select from the remaining models. A commission index is calculated as the proportional area predicted to be present by the model [23]. A user-defined number of low-omission models closest to the median commission index are then selected as the "best subset" of the models.
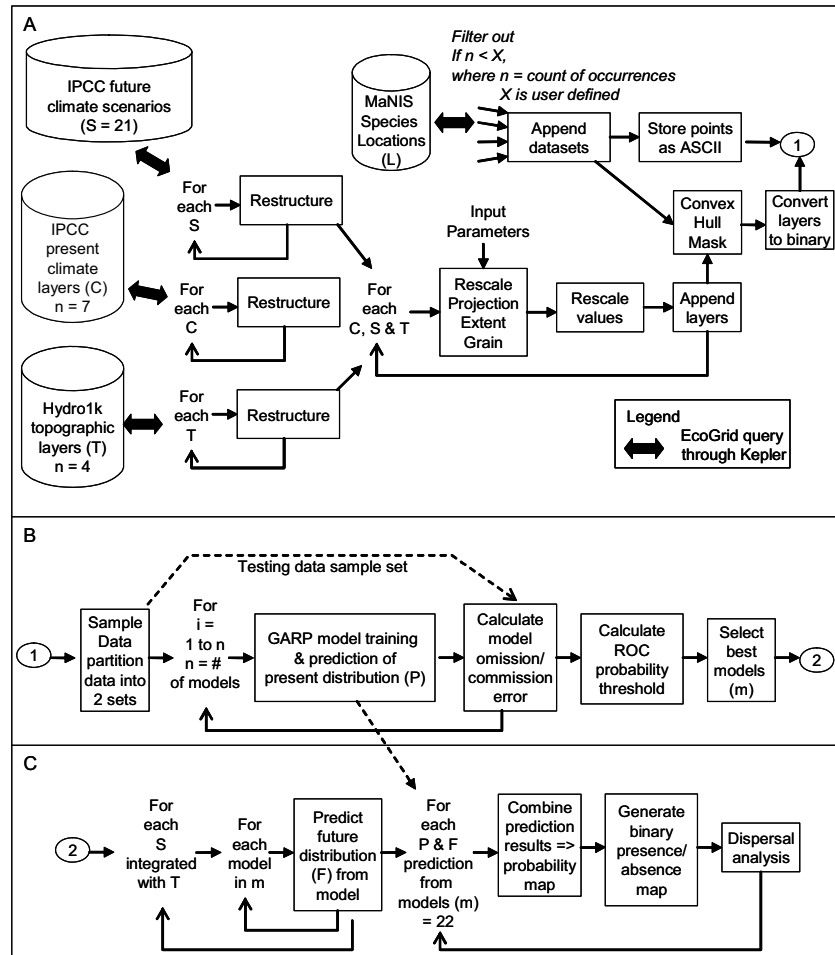
Figure 7.2: Conceptual workflow for the mammal project: (a) data prepro-
cessing, where distributed data are obtained from the EcoGrid and manipu-
lated into the required formats; (b) model development, including training,
testing, and prediction on present climate scenarios; (c) model prediction on
future climate scenarios and comparison of output. Final output consists of
predicted distribution maps for each species and climate scenario.

7.4.3 Model Application and Comparison

Application of the model and comparison between predictions (Figure 7.2c)
occurs within the Calculate Best Rulesets subworkflow (Figure 7.3: subwork-
flow VI). Once a best subset of models has been selected, they are used to
predict the specie's future distributions for the many possible future climate

scenarios available (Figure 7.4). Because the best subset of models generates multiple predicted distributions for each present and future scenario, they are combined for each scenario to produce an occurrence probability map. Error for the model set as a whole is evaluated using threshold-independent receiver operating characteristic (ROC) plots [468]. ROC analysis evaluates the specificity (absence of commission error) and sensitivity (absence of omission error) of a model set in comparison with a random prediction using a z test. The results of the ROC analysis are used to validate the predictive ability of a model for a particular species; for those species passing the validation test, we construct a final map of the specie's predicted distribution under present and numerous versions of future conditions. This distribution may then be further limited by the use of spread (contagion) algorithms, which evaluate the ability of the species to colonize new areas under different assumptions.

For each species, the ENM workflow results in predictive maps of the current distribution and of potential future distributions under different climatic scenarios that can be compared to analyze effects of climate change on each species. Collectively, results for all species can be analyzed for current biodiversity patterns and effects of climate change on biodiversity. Additional workflows will be developed to conduct these analyses. The derived data, and all workflows associated with the analysis, are archived to the EcoGrid.

7.5 Modular Component Substitution

Each actor or subworkflow of the ecological niche modeling workflow can be replaced easily and as needed. For instance, a scientist may want to run the workflow using all of the data preparation, sampling, and postprocessing on the model output but using a different niche modeling algorithm. Such functionality would require an actor substitution. Alternatively, the scientist may wish to run the same workflow but using different data sources, requiring construction of a new data-preparation workflow, conversion of that workflow into a subworkflow, and substitution of the new subworkflow for the existing one that it is replacing. As any number of variations on the workflow might be needed, modular construction of the workflow allows individual components or sections of the workflow to be substituted readily. Actor and subworkflow substitutions are illustrated below.

7.5.1 Actor Substitution

The ecological niche modeling workflow was originally designed to make use of the GARP model, available in Desktop GARP.[1] Desktop GARP is written in C code and includes three parts, which we subdivided into separate Kepler

---

[1] `http://www.lifemapper.org/desktopgarp/`
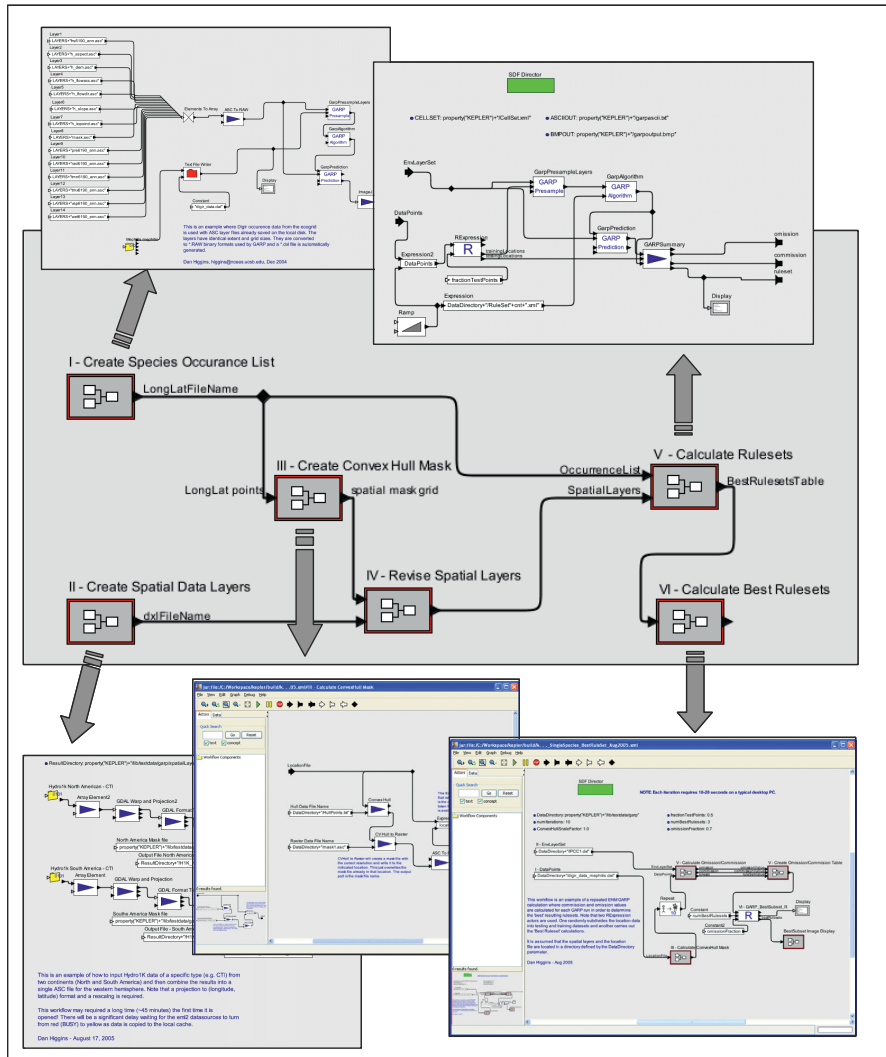
Figure 7.3: Hierarchical decomposition of the ecological niche modeling workflow in Kepler. Each of these subworkflows consists of a nested workflow, which itself may contain subworkflows and further nesting.

actors: (1) GARP Presample, (2) GARP Algorithm, and (3) GARP Prediction (Figure 7.4). The decision to subdivide these modules was based on the consensus that each could be reused independently from the others.

Actor substitution could occur by simply deleting any of these three actors and replacing it with the new desired actor, which might be a differ-

ent sampling routine or a different algorithm with which to construct the model. Numerous issues arise during actor substitution regarding the syntactic, structural, and semantic compatibility of the replacement actor. In this particular instance, the GARP algorithm requires input in a specific format, namely a comma-delimited ASCII text file, with each row containing the location <latitude, longitude> where the species is known to occur and a vector of numeric data summarizing environmental characteristics at that location. All attributes, with the exception of <latitude, longitude>, must be integers between 1 and 254. Substitution of a different actor requires either that the new actor have the same input requirements or that additional actors be incorporated into the workflow to transform the output from the GARP Presample actor into the required input format for the new algorithm. Likewise, the output from the new actor may require transformation to meet the input requirements of the GARP Prediction actor.

Additionally, actor substitution may entail major changes to the overall workflow design. For instance, the GARP algorithm is stochastic rather than deterministic—it is run many times for a given experiment, and each run produces a different model. The workflow is designed to iterate many times over the GARP algorithm for each species. Substitution of the GARP algorithm with another stochastic algorithm would not require major changes to the workflow structure, but substitution with a deterministic model would. Hence, actor substitution, while simple conceptually, requires additional effort that could range from minimal (the actors are completely compatible and just need to be rewired) to quite extensive (the workflow must be redesigned and new portions constructed). In any case, the workload involved is less than if the entire design was reworked from scratch.

7.5.2 Subworkflow Substitution

The GARP Presample actor requires that all of the input environmental layers be spatial raster grids with a custom binary format that must have identical extent and resolutions. Substitution of a different sampling algorithm may require a different preprocessing workflow. Alternatively, rather than Hydro-1k data, the user may wish to use a different data source that has its own preprocessing requirements. In either case, one or more subworkflows would have to be replaced, substituting major portions of the workflow. Since Kepler is designed with hierarchical components, these kinds of substitutions can be handled more readily than if no logical grouping of components existed.

Multiple subworkflows are used to put the data in required formats (Figure 7.3 subworkflows II, III, and IV). Embedded within the subworkflows are complex data-processing workflows for IPCC present climate data, IPCC future climate data, and Hydro-1k topographic data (see Figure 7.5 for the Hydro-1k workflow). If, for instance, the Hydro-1k data source were replaced with another, this specific subworkflow could be deleted and replaced with a new one without modifying the subworkflows that handle the other data
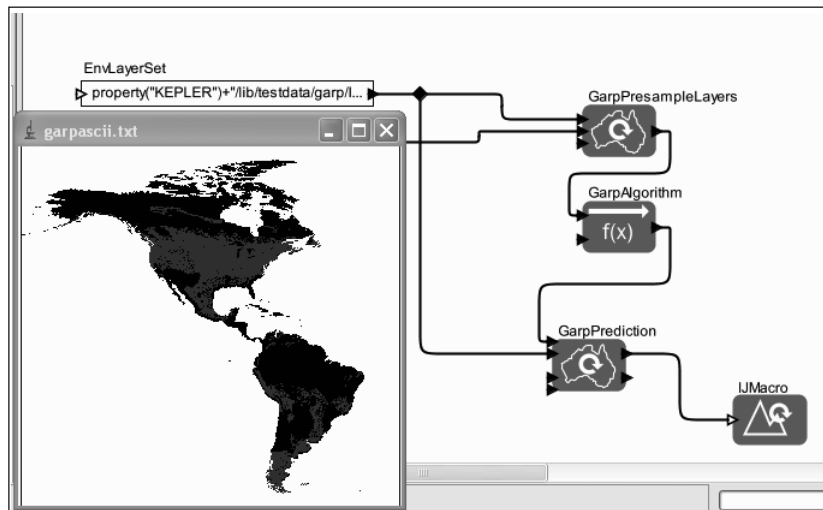
Figure 7.4: The GARP workflow, consisting of the Desktop GARP software subdivided into three actors (GARP Presample, GARP Algorithm, and GARP Prediction), and an image display actor to display output. The three GARP actors together constitute the GARP model training box shown on the conceptual workflow (Figure 7.2b). Input data must be preprocessed (not shown). Output data are in the form of an image showing the predicted distribution of the species.

sources. The primary issue to address is that of ensuring that the output from the replacement workflow is compatible with the input requirements of the next step.

7.6 Transformation and Data Integration

A comparison of the research design for integrating species occurrence data and environmental data (Figure 7.1a) with the corresponding conceptual workflow (Figure 7.2) and with the details of execution (Figures 7.3 and 7.5) illustrates the tremendous expansion of computational detail required to preprocess and integrate biodiversity data, even when the conceptual research design is relatively simple. Much of the expansion occurs early in the workflow, as the source data are being manipulated and transformed into formats required by the first analytical step (Figure 7.2b—Sample Data). These preprocessing steps require substantial time and effort in most biodiversity analyses. One goal of Kepler is to reduce the amount of effort required by scientists to accomplish such tasks through a rich set of transformation components,
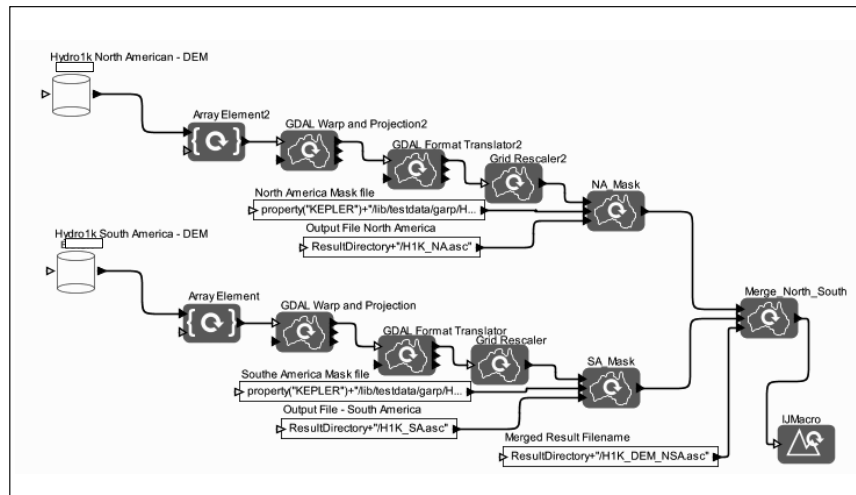
Figure 7.5: The Kepler workflow for preprocessing of Hydro-1k environmental layers for North and South America.

and eventually, automatic or semiautomatic data integration and transformation [46].

7.6.1 Transformation Components

Historically, data preprocessing has required extensive manual effort because of the diverse set of functions needed. Syntactic and structural transformations have most often been accomplished through cutting and pasting in a spreadsheet application. Kepler provides a range of transformation components that can automate many of these conversions. It also includes some simple semantic conversions, such as standard unit conversions. As the library of Kepler components expands, a rich set of transformations will be provided.

In ecological niche modeling analyses, manual data-preparation steps may require six or more months of labor, much of which is done within a GIS. However, most ecological niche modeling studies use only a small subset of GIS functions, typically those that allow for integration of multiple data sources (projection, resampling, clipping, rasterization), changing grid values (map algebra), and sampling grids or polygons from point data. Kepler provides a way for ecological modelers, often not GIS specialists, to access and use transparently the geospatial functionality that they need.

We have implemented geospatial actors using the Geospatial Data Abstraction Library[1] (GDAL) and Geographic Resources Analysis Support Sys-

---

[1] http://www.remotesensing.org/gdal/

tem[1] (GRASS) because they are open-source, free software packages with powerful and proven raster, topological vector, image processing, and graphics production functionalities. We have also implemented some spatial functionality with Java-based ImageJ.[2] Lastly, we have implemented several Environmental Systems Research Institute (ESRI) ArcGIS functions as Web services that can be invoked through Kepler (our collaborators have a license from ESRI for such services). We are adding geospatial functionality as needed for our applications, with plans to develop the geospatial functionality base more broadly in the future. We do not intend to duplicate a full GIS; complex geospatial analyses are best carried out within existing software systems. The goal in Kepler is to provide functionality that will allow the products of a geospatial analysis to be integrated more easily with other types of data and software through standard transformations in batch mode.

7.6.2 Semiautomatic Data Integration.

Given a set of transformation components in Kepler, it is possible to annotate them in such a way as to enable partially automated transformations by the system. To illustrate this nontrivial task, we use the Sample Data step (Figure 7.6). At the conceptual level, the Sample Data step requires three input types (Figure 7.6a): (1) species presence points, (2) environmental layers, and (3) user-defined parameters that specify the kind of sampling to be conducted, the number of desired sample sets, and the number of samples within each set. The algorithm itself is designed to perform spatial integration of point data with grid data. All steps prior to this one are syntactic, structural, and semantic transformations to place the point and grid data in the correct format for input into the "sample data" step. At the syntactic level (Figure 7.6b), the input point data must be a single comma-delimited plain text file. Point data retrieved from MaNIS consist of multiple tables that must be combined into a single table and written to an ASCII file.

Both IPCC and Hydro-1k data begin in an ASCII format but must be rewritten into the binary format required by the sample data algorithm. At the structural level (Figure 7.6c), columns within the point file must be restructured into the expected order (longitude, latitude). Numerous structural transformations must occur on the environmental layers. The climate data occur as a single global map; the Hydro-1k data are subdivided by continent. Either the Hydro-1k data must be merged or the climate data can be clipped, depending on the spatial extent of the specific point data being used in the analysis. Grids must be in comparable cartographic projections to be combined. The Sample Data algorithm requires that spatial extent and resolution be identical, requiring spatial clipping and resampling to change resolution. Lastly, layers must be submitted together as a set that is formally a list of

---

[1] `http://grass.itc.it`

[2] `http://rsb.info.nih.gov/ij/`

vectors, where the x and y locations can be inferred from the position of the z value in the vector and metadata regarding the spatial extent and resolution.

The required syntactic and structural transformations may be automated by the system if sufficient data and algorithm annotation is available, and if the semantics of the transformations are known. For example, the sample data actor could be annotated to specify the syntactic and structural requirements of the input data where these are not already formalized through the input type. Semantic annotations can be made that specify that the input grid data are spatial rasters, must be spatially equal, and must spatially contain the input point data (Figure 7.6d). If ontologies exist that formally define spatially equal for raster data as having equivalent projections, extent, and resolution, and annotated actors exist that perform those tasks, then the system could infer that those steps are necessary and perform them without their being specified within the workflow. If the sample data actor is annotated as requiring geographic longitude and latitude, the rescale step could automatically select that projection without user input.

Likewise, if ontologies exist that formally define the spatially contains relationship between raster and point data, a point is formally defined as consisting of a composite of longitude and latitude, and the input longitude and latitude data are annotated as a point composite, then (given a specific input point data set) the system could infer the appropriate extent of the raster data for the rescale step. The only remaining parameter for the rescale step is the desired resolution (grain), which would still either need to be specified by the user or a default value could be determined if associated with another knowledge base such as a decision support system. There may still remain some initial processing of any data set that is collected for purposes other than use in a given workflow, but automation of those syntactic, structural, and semantic steps for which information is available to the system would be an exciting step forward toward allowing scientists to focus on the scientific portion of the analysis (Figure 7.2b) rather than the transformation and conversion portion (Figure 7.2a).

7.7 Grid and Peer-to-Peer Computing

Currently, Kepler downloads all data sets to a cache and executes locally, but we will soon be incorporating distributed computing. We estimate that there are several thousand species of mammals in MaNIS that might be considered in this prototype application. If we can do all the calculations needed for a single species in an hour, there is a need for several thousand hours of computing time for the entire list. Thus, there is clearly a need for distributing calculations over numerous computers. These might be specially designed parallel clusters, but since Kepler will run on standard desktop PCs, one could also consider other, less specialized methods for distributing the calculations. Peer-to-peer networking among Kepler clients is a technique for parallel pro-

A

Conceptual Level

Species presence points

Environmental Layers (temperature, vegetation, etc.)

parameters

Sample Data

B

Syntactic Level

Data as comma-delimited, plain text file

106.789098, 33.454606;
106.789097, 33.454606;
…

Sample Data

Data as binary

56.25 57.23 54.66 55.56 53.25
56.37 52.29 53.52 53.55 56.21
…

C

Structural Level

2-dimensional matrix

matrix[y, x]

Sample Data

List of 1-dimensional vectors, one vectpr per environmental layer
x & y inferred from position in vector

list(*vector*[z])

D

Semantic Level

Spatial Ontology: point

Spatial Ontology: longitude

Taxonomic Ontology: species

Spatial Ontology: latitude

Sample Data

Spatial Ontology: spatially-contains

Spatial Ontology: raster

Observation Ontology: property

Spatial Ontology: spatially-equal

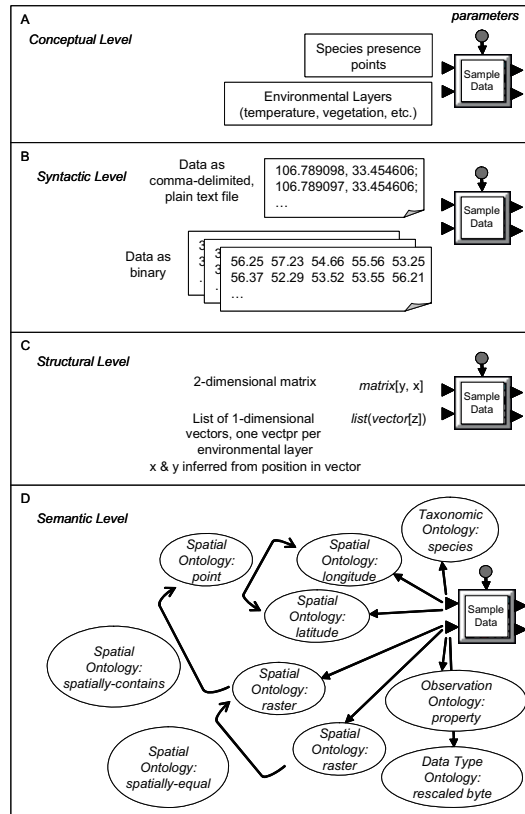*Spatial Ontology: raster*

Data Type Ontology: rescaled byte

Figure 7.6: Description of input data to the Sample Data actor at (a) conceptual, (b) syntactic, (c) structural, and (d) semantic levels, illustrating the complexity of transformations that must be made during preprocessing of source data.

cessing that is being considered. Dividing the problem at the species level (i.e., running all the calculations for a single species on one machine) seems to be reasonable, but results would have to be saved, perhaps to the EcoGrid, for later integration.

7.8 Opportunities for Biodiversity Science Using Scientific Workflows

The ENM workflow is being developed as a prototype application. Once complete, it may be reused for many comparable analyses simply by changing the taxa and/or geographic location of interest, or changing the input biodiversity

and environmental data sources. Substitution of algorithms such as generalized additive models or neural networks for the GARP algorithm provides the opportunity to reuse the workflow with minor modification. We are currently evaluating options for additional workflow development. Our intention is to construct workflows that will (1) be of most use in improving the efficiency of complex biodiversity and ecological analyses, (2) link practically with existing workflows to form even more complex analyses, and (3) complement existing workflows conceptually.

The ecological niche modeling approach deliberately neglects the effects of dispersal on geographic distributions [391]. That is, the niche models summarize the ecological suitability but do not attempt to establish whether the species will be able to disperse to and colonize a given site that may be suitable. The likelihood of successful dispersal is an independent question and has been the subject of numerous development efforts in distinct lines of inquiry [49, 186].

For any future-climate effects on biodiversity modeling efforts, incorporation of dispersal considerations is key—suitable areas may exist but may be out of reach for many species [347]. Most such analyses to date have used simple dispersal assumptions such as "no dispersal," "universal dispersal," and "contiguous dispersal" [426], but have not made attempts to decide which of these scenarios is most likely or most realistic. Clearly, this aspect of the question merits considerable additional effort and thought by the niche modeling community.

In the mammal project, we will implement a series of layers of complexity in dealing with dispersal considerations. At the simplest level, we will apply the no, universal, and contiguous dispersal criteria—this approach has the important advantage of permitting direct comparisons with previous studies [426]. However, the workflow approach will permit a much more interactive assessment of the effects of these different assumptions.

The mammal project will result in grid layers representing many alternative future potential distributions for more than 2000 species. Analyses of these voluminous data will take several forms. Data exploration, reduction, and graphical visualization workflows are needed. For a given species, we need to compare alternative distributions, both in terms of amounts of suitable habitat and spatial arrangement of that habitat. Some species require large blocks of contiguous habitat to survive, whereas others require a heterogeneous mix of habitats. Complex spatial analyses within a given distribution and comparing between distributions are needed. Comparisons between species that allow the delineation of response groups (groups of species that respond to climate change scenarios in similar ways) are needed. Workflows that analyze alternative community structures and the effect of interactions between species will be needed. Error detection and uncertainty analysis of results both within and between scenarios will also be important.

7.9 Advantages of Automated Workflows for Biodiversity and Ecological Science

The benefits of scientific workflows for biodiversity scientists are many: Increased efficiency, replicability, and reusability are obvious. Less obvious, but of no less importance, is the explicit documentation of methods used in an analysis. Historically, analytical methods have been recorded in free-text "Methods" sections of publications. Typically, only the conceptual steps are recorded. The multitude of computational details imposed on the data to enable execution are typically not recorded, yet these may have significant effects on the results of the analysis.

Scientific workflows provide the opportunity to formally document every detail of the analysis within the system. Indeed, methodologies can be "published" explicitly in the form of workflows as part of scientific papers. This enables replication of analyses as mentioned above but also enables scientists to scrutinize their own and other scientists' analytical methods carefully, identify differences in methodology that have significant effects, and compare results given those differences. Additionally, it presents an opportunity to refine a given workflow collaboratively based on group consensus of best practices. By agreeing on and standardizing the details of an analysis wherever possible, truly innovative differences in approaches that occur at the cutting edge of science will be highlighted, and we may focus more readily on analytical outcomes that result from those differences rather than obscuring them through differences in execution.

Science is about exploring those areas of knowledge where no consensus exists and where no established methodologies guide the investigator. By automating analyses, efforts can be concentrated where they are most needed. Fortunately for scientists, many parts of a scientific analysis provide a wealth of technical challenges for computer scientists and software engineers, and emerging technologies such as workflows hold great promise. The single biggest hurdle preventing widespread adoption of workflow technology by the biodiversity science community is the level of technical expertise required to construct executable workflows. Most have limited or no programming background and little knowledge about fundamental technical issues such as data types, structures, and information handling. Nor should they be expected to become technical professionals—domain scientists should be doing domain science! Until the system is populated with a wide variety of components and many reusable workflows, each new workflow will necessitate programming of custom actors. Additionally, the workflow design process itself is not intuitive to scientists who are used to making decisions about their analytical methods on the fly as they conduct their work. We will have to find a way to simplify Kepler for less sophisticated users while still enabling complex functionality. It is quite a challenge to provide the range of functionality envisioned while maintaining a reasonably simple interface that will be intuitive to the domain scientists. Concurrently, we must also develop more sophisticated tools

to enable rapid workflow construction by high-end workflow engineers who may be working in collaboration with a domain scientist. Balancing these orthogonal needs will continue to be a challenge. Ultimately we envision a day when Kepler evolves into a hierarchical system that fully supports users with a wide range of technical capabilities from a wide range of scientific disciplines, presents the appropriate set of interfaces and functionality based on user group, and enables better collaboration between the disciplines.

Acknowledgments

Case Studies on the Use of Workflow Technologies for Scientific Analysis: The Biomedical Informatics Research Network and the Telescience Project

Abel W. Lin, Steven T. Peltier, Jeffrey S. Grethe, and
Mark H. Ellisman

8.1 Introduction

The advent of "Grids," or Grid computing, has led to a fundamental shift in the development of applications for managing and performing computational or data-intensive analyses. A current challenge faced by the Grid community entails modeling the work patterns of domain or bench scientists and providing robust solutions utilizing distributed infrastructures. These challenges spawned efforts to develop "workflows" to manage programs and data on behalf of the end user. The technologies come from multiple scientific fields, often with disparate definitions, and have unique advantages and disadvantages, depending on the nature of the scientific process in which they are used. In this chapter, we argue that to maximize the impact of these efforts, there is value in promoting the use of workflows within a tiered, hierarchical structure where each of these emerging workflow pieces are interoperable. We present workflow models of the Telescience[TM] Project[1] and BIRN[2] architectures as frameworks that manage multiple tiers of workflows to provide tailored solutions for end-to-end scientific processes.

Utilization models for first-generation Grids (and their supercomputing center predecessors) were akin to the hub-and-spoke model utilized by the airline industry. User data environments were treated as the "hub," and at every step, the user was required to login and data were passed (often with a binary executable) to one of the few virtual organizations (VO) [147], or spokes, across the country to execute their computational jobs (Figure 8.1). Initial implementations required users to coordinate the execution of their data-processing tasks using command-line interfaces. They further required users to maintain their own security credentials on each of the resources targeted for their jobs. Today, single sign-on authentication and login mechan-

---

[1] `http://telescience.ucsd.edu`
[2] `http://www.nBIRN.net`

isms have been realized through the use of Grid portals. Instead of logging into specific resources via a command prompt, users are directed to a single Web page, where their authenticated login provides access to the VO or other organizations where a shared-use relationship has been established.

Through the use of Grid portals, complex command-line arguments and syntax are easily replaced with radio-buttons and checkboxes, thereby simplifying the syntactical interface to the use of distributed resources. Even with these simplified interfaces, however, first-generation Grid implementations still operated on a "hub-and-spoke" model. Modern Grid portals, coupled with maturing workflow tools have begun to enable a point-to-point research model that more closely mirrors scientific research.
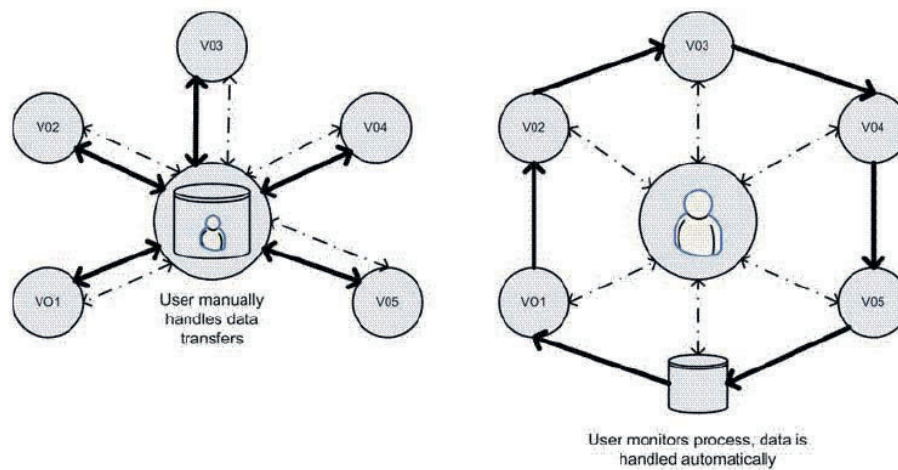

Figure 8.1: Hub-and-spoke versus point-to-point scientific processes. Solid lines indicate data transfer and hashed lines indicate user monitoring tools.

Through the use of Web-based environments, researchers can now access a fully integrated cyberinfrastructure in a nearly ubiquitous manner, with little to no administrative overhead. In current systems, experiment workflows are represented by Web interfaces that provide centralized access to a static collection of sequentially ordered application pipelines, tools for launching batch jobs, or tools for visualizing or analyzing the data at key points in the end-to-end process.

While the advantages of these computational portals are evident, there remains a need for additional flexibility. The natural working paradigm for most scientific experiments requires a level of interactivity that is difficult to capture in a static workflow. Users really require a balanced environment where

they can interactively create, replicate, and reuse workflows or "pipelines" for application components from the larger scientific process without needing to manage the complexities of their planning or execution on end-line physical resources. While there are a number of technologies emerging to enable interactive "plan" creation and/or scalable plan execution on Grids, few if any provide a balanced and unified capability on both fronts. Most offer unique capabilities, with strengths and weaknesses that need to be combined and tailored to meet the requirements of scientific experiments. This model of interoperability is essential for projects such as Telescience [260, 341] and BIRN [169].

8.2 Framework for Integrated Workflow Environments

As described in Part III of this book, a number of technologies have emerged to redefine the workflow concept by providing frameworks for interactive process construction, execution, and replication. Leading efforts such as Kepler[1] and Taverna[2] offer pipelining operations that provide users with a real-time interactive and/or visual environment for constructing and executing end-to-end data-analysis plans (for more information regarding Kepler and Taverna, see chapter 19). Other classes of workflow technologies, such as Pegasus (Chapter 23) and DataCutter[3], excel at the planning and execution of such plans onto heterogeneous resource environments or Grids. The challenge is that workflows, as defined by domain scientists, typically represent the dynamic end-to-end application process that often includes a heterogeneous mix of experimental processes and the corresponding collection of distinct workflows (information gathering, bench/laboratory experimentation, computation, analysis, visualization, etc.) that may require a reconfigurable mixture of the workflow classes described above.

In an era of growing complexity, it is a daunting task for scientists to manually traverse these different workflow classes to complete their multiple experiments. The Telescience and BIRN projects are structured to effectively manage these different classes of workflows and to represent them to the user in a simple sign-on Web portal with a seamless end-to-end data flow. The portal in this case serves as the unifying fabric within which the disparate workflow technologies are integrated and where the state between technologies is brokered on behalf of the user. In the classic model of the Grid, users and applications are connected to physical resources via Grid middleware. The Telescience v2.0 system architecture (Figure 8.2) is a mature embodiment of this concept that consists of four primary layers:

- User interface: portal and applications

---

[1] http://www.kepler-project.org/
[2] http://www.mygrid.org.uk
[3] http://datacutter.osu.edu/

- ATOMIC: Application to Middleware Interaction Component Services
- Middleware/cyberinfrastructure: collective and local Grid services
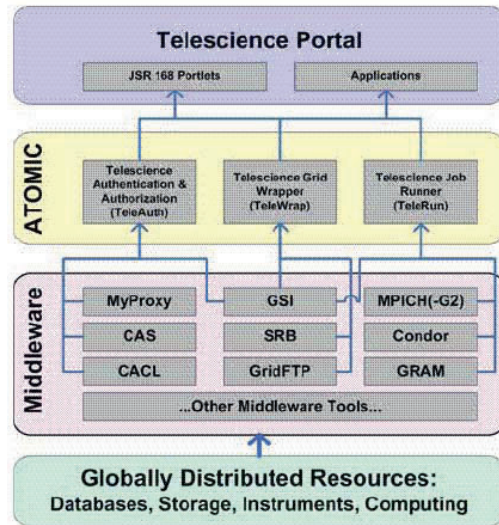- Physical resources: computing, storage, visualization, and instrumentation

Figure 8.2: Telescience architecture. The portal presents a simplified interface to data, services, and collaboration tools to end users and transparently manages access to cyberinfrastructure. ATOMIC insulates application developers from the heterogeneity and volatility of the middleware cyberinfrastructure, streamlining the linkage of client-side resources to distributed physical resources.

Figure 8.2 shows the overall Telescience architecture, which maps directly to the base architecture of BIRN. User interaction occurs via a Web portal interface that ultimately traverses a series of layers to the required physical resources. Telescience and BIRN have deployed a user portal based on the GridSphere framework[1]. GridSphere, being a JSR168 compliant portlet framework [419], allows for the development of portlets that can be utilized in numerous compliant portal frameworks. The Web portal may launch one or more applications that must also navigate the same services to access the physical resources. These portal and application components interact with Grid resources via ATOMIC [259]. ATOMIC is a set of services that organize middleware technologies into thematic bundles with stable and unified pro-

---

[1] http://www.gridsphere.org

grammer interfaces to simplify the process of integrating tools with the Grid for the domain applications developer.

Within this framework, however, there is still a temptation to build complicated software that captures all necessary functionality (across layers) in a single program. Workflow tools aim to reduce that tendency by working across layers to link together disparate codes, modules, and applications (some preexisting) into a single virtual environment, all without significant changes to the original source code.
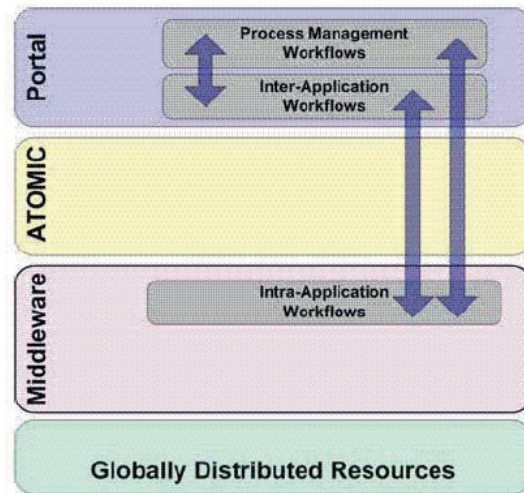

Figure 8.3: Workflow integration across scales and classes of tools. In the context of workflows, the Telescience portal curates and manages user information and session state. ATOMIC delivers that information to downstream applications and workflows.

Within the Telescience and BIRN architectures, those workflow tools fall into the following classes:

1. Process management workflows: Frame highest-level scientific (laboratory) process and provide policy, process, state management, and administrative tools, including the coordination and management of lower-level workflows/pipelines that may comprise a scientific study (or instance within that study)
2. Inter-application workflows: Pipeline or plan-building tools to streamline computational operations
3. Intra-application workflows: Planners and execution engines to optimize the execution of these plans on heterogeneous physical resources.

As shown in Figure 8.3, the BIRN and Telescience approach is to facilitate coordination and sharing of state-full information between these three workflow layers. Each layer has unique abilities and requirements. Process and state management tools (typically portal-based) are necessary to preserve and delegate the contextual information with regard to the user. This information includes management of the scientific process, authentication and authorization, and high-level state information. Within the Telescience Project, much of this information is delivered to the lower level workflows via the ATOMIC toolkit. Inter-application tools create process pipelines, which are subcomponents of the highest-level experimental process management workflow. These tools are typically user-driven GUI environments that are either ordered within the process management workflow or presented as a general tool to serve the process management workflow as needed. The lowest-level "intra-application" workflows are composed of the executable plans that have been mapped to heterogeneous pools of physical resources.

8.3 Scientific Process Workflows: Process and State Management Tools

The laboratory process is the end-to-end process that a scientist embarks upon. This process is defined as all the steps between the conception of an experiment and the final discoveries that are made based upon experimental findings, including but not limited to any initial planning, information gathering, data collection, analysis, and potentially many iterations of this process at one or many decision points. In fact, the laboratory process is not simply a linear stovepipe process, but rather it is a dynamic and highly iterative process with multiple points of user interaction, data visualization, and feedback (see 8.4). In the context of workflows, the laboratory process is the first-order workflow in the hierarchy of workflow tools and is the first workflow level that directly interacts with the end-user. Within an interoperable hierarchy of workflow technologies, these laboratory processes utilize Grid portals in the role for which they were originally intended, to provide a stable base structure for the process as a whole, to broker security credentials, to manage the secure flow of information (through disjointed processes that often involve multiple forks or decision points), to monitor/audit the progress of the overall scientific process (including bench processes that are experimental and non-computational), and to serve as the controller of workflow state information. It is no surprise that portals have emerged as a dominant source of application and information delivery. The Gartner Group[1] has championed the portal as a mechanism that provides access to and interaction with relevant information, applications, business processes and human resources by

---

[1] `http://www.gartner.com`

select targeted audiences in a highly personalized manner. According to Gartner, the enterprise portal has become the most desired user interface in Global 2000 enterprises and is ranked as one of the top ten areas of technology focus by CIOs. Translated to the scientific process, portals provide the tools to transparently manage the contextual information that is required for the different workflow classes to interoperate. Some of this information includes authentication and authorization, data and resource management, and session notifications.
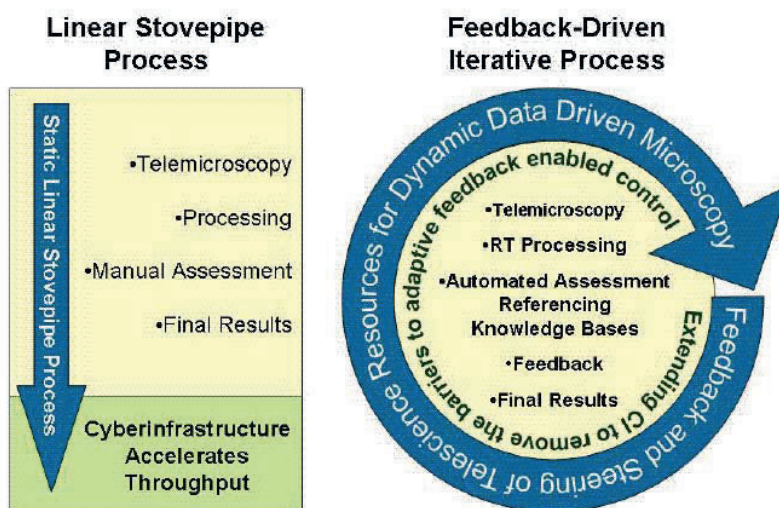

Figure 8.4: Portals are critical to the workflow landscape as scientific workflows move from linear stovepipe processes to dynamic feedback-driven processes.

8.4 The Role of Portals as Workflow Controllers

As we move from a hub-and-spoke model to a dynamic "point-to-point" process, the role of the portal as workflow controller becomes more important. In this role, the portal is utilized more for process workflow management, where more emphasis is placed on the management of state and persistence information of the different components and less emphasis is placed on the mechanics of launching application components. While not traditionally thought of as "workflow" tools, we have found portals to be critical to applications and workflow information delivery.

In the Telescience model, the portal environment is required to curate all the pertinent information regarding the user and session state that is required by lower-level workflow classes. While the portal serves as the curator of user information and state, ATOMIC serves as the delivery vehicle, providing downstream applications and workflows with access to the appropriate information necessary for a given process. This abstraction of the session information is not only necessary to maintain a seamless user environment during the transition between different workflow classes but also scales to the needs of future workflow technology developments. Recently, two important standards have emerged to address the scalable development of session management across scales: Web Services for Remote Portlets Specification (WSRP) [484]and Java Specification Request 168 Portlet Specification (JSR 168). Independent of programming languages and platforms, WSRP defines Web Services Description Language (WSDL) [482] interfaces and semantics for presentation-oriented Web services. JSR168 meanwhile, defines a standard Java portlet API, a portlet container, and the contract between the API and the container. Armed with these standards, portlets have become one of the most exciting areas for presenting applications and workflows to the end user, with the number of vendors (and open-source projects) that support portlets serve as evidence. These include IBM WebSphere, Sun One Portal Server, Oracle 9iAS, the Jakarta Jetspeed project, and the GridSphere project. These two emerging standards have enabled the development of tools to systematically manage persistent contextual information on behalf of the user.

Currently the Telescience and BIRN projects are utilizing these standardizing portlet framework tools to develop an administrative system to allow for the rapid creation and deployment of a process management controller portlet. These controller portlets provide high-level structure to end-to-end experimental processes, framing the logical steps that may then expand into multiple layers of successive workflows and tools.

Application-centric portals, such as Telescience and BIRN, take advantage of not only the portable presentation layers of portlets but also the persistence and management of logic and state information between portal components and lower-level workflow tools. It is this vital information that makes a unified, point-to-point Grid interface possible.

8.5 Interapplication Workflows: Pipeline-Building Tools

A major area of development in workflow technologies has been the development of systems (i.e., application pipeline environments) that allow the management and execution of analysis processes. The utilization of such environments enables not only the initial analysis of the experimental data but also the recalculation for verification of the results and the exploration of the parameter space of the results. Additionally, the use of such environments al-
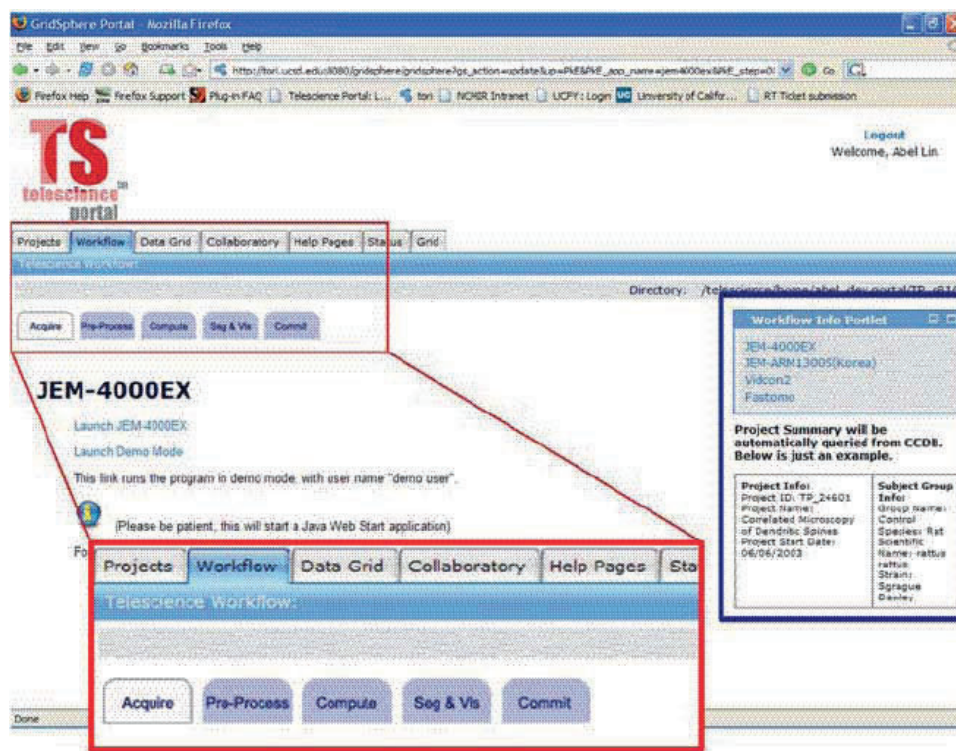
Figure 8.5: Screenshot of the laboratory process workflow (highlighted in solid rectangle). This workflow portlet closely interacts with the data Grid and application selection portlets (highlighted in hashed rectangles) and is configurable by portal administrators.

lows for the processing of scientific data to be well documented so that studies may be explored and analyzed further at a later time by interested researchers. These requirements are found in many scientific communities and have resulted in the development of many such environments across these communities. Some examples of environments developed in different communities are:

- Neuroscience - LONI Pipeline[1] is a graphical environment for constructing analysis pipelines.
- Bioinformatics - Taverna is an environment that merged with the myGrid Project[2] and allows the researchers to access collections of bioinformatics Web services and Grid services hosted by the European Bioinformatics Institute.

---

[1] `http://www.loni.ucla.edu/twiki/bin/view/Pipeline/`

[2] `http://www.mygrid.org.uk`

- Ecological and Geological Sciences - Kepler[1] is a workflow environment based on the Ptolemy II system for heterogeneous, concurrent modeling and design.

While it may be impossible to standardize on a single application pipeline environment due to the requirements of a specific research community or study, it has become increasingly important to provide an environment in which users can build and utilize preconstructed application workflows via a unified portal interface. As communities develop conventions for the processing of certain data (e.g. quality assurance measures for functional MRI data developed within BIRN), it will be important for the process management workflow to be able to integrate these components into the overall scientific process, thereby increasing the interoperability of application workflows across communities and projects.

8.6 Intrapipeline Workflow: Planners and Execution Engines

Typically, each application (or intra-application pipeline components) can be broken down to individual module components that no longer require user intervention. At this level, the component codes are well suited for large-scale computation. Unlike first-generation Grid codes that were large and monolithic, these component modules are small and dynamic. Also unlike early codes, which tended to be "pleasantly parallel," these modern codes are heterogeneously parallel, often requiring more than one precursor component to be completed before computation can begin. With this parallel heterogeneity, mixed with resource heterogeneity, sophisticated workflow planning and execution tools are required to first abstractly plan and then execute the workflow. As with the interworkflow pipeline tools, these requirements are found in many scientific communities and have resulted in the development of many such environments across these communities. Some examples of environments developed in different communities include:

- Physics: The GriPhyN virtual data system[2] provides tools for executing workflows and for the tracking of provenance of all data derived from the workflow.
- Astronomy: The Pegasus[3] (Chapter 23) environment provides a flexible framework for the mapping of scientific workflows onto Grid-based resources.
- Geology: DataCutter[4] is a middleware tool for filtering large archival scientific datasets in a Grid environment.

---

[1] http://www.kepler-project.org/
[2] http://vds.isi.edu
[3] http://pegasus.isi.edu
[4] http://datacutter.osu.edu

8.7 Use Cases

The Telescience and BIRN projects provide a framework for the integration and interoperation of all of these different workflow classes within the context of an end-to-end scientific process. More than offering yet another *one-size fits all solution*, the goal is to introduce a model for interoperability that enables disparate but complementary technologies (process management, inter-application, and intra-application workflows) to work in synchrony. The hierarchal organization of workflows tools is not only aimed at processing more data faster, but to also increase the rate at which native scientific applications be deployed in order to take advantage of the services from different workflow tools.

8.8 The Telescience Project

Imagine an environment where available computing cycles are dynamically gathered for real-time on-demand computing directly from the data-generating instruments themselves (instead of user-managed monolithic large-scale, batch-oriented computation). In this unified, on-demand Grid, data are automatically curated and flow freely from instrument to computation to analysis. In this model, the results of that analysis interface directly with the instrument, providing automated feedback that can constantly refine data-collection parameters and techniques. In this world, the Grid provides more than just a means for faster results; it provides a foundation for the collection of higher fidelity raw data. This is the vision of the Telescience Project.

To monitor that point-to-point data flow, the core functionality of the Telescience portal is the user managed microscopy workflow, where the sequence of steps required for planning experiments and acquiring, processing, visualizing, and extracting useful information from both 3D electron and laser-scanning light microscopy data is presented to the user in an intuitive, single-signon Web environment. Beyond facilitating the execution of these steps, however, the Telescience system audits progress through the workflow and interfaces each component within the workflow with federated databases to collect and manage all of the metadata generated across the entire process.

As with all first generation portals, a major accomplishment for Telescience v1.0 (circa 1999-2004) was to create simple Web accessible interfaces to a heterogeneous set of middleware via a single user name and password. Telescience v1.0, for example, users could browse the data Grid via a custom interface or launch jobs via web wrapped middleware commands. These interfaces, however, were designed in autonomy for singular interactions whose capabilities were developed by mirroring a command line interface to the particular middleware tools. Due to limitations in the infrastructure, first generation portals moved from the original purpose of monitoring the process to also being responsible for the execution of the process.

The Telescience v2.0 infrastructure is designed to move beyond interfaces with singular actions and integrate them into a richer user environment; that is automated and dictated by the process and not by the Grid middleware. This capability is accomplished with the development of a workflow portlet that manages and monitors the highest-level scientific process (see Figure 8.5). Similarly to many scientific processes, the highest-level process remains relatively static to the end user (while the pipeline subcomponents are much more variable/dynamic). The Telescience workflow portlet, however, is amenable to other types of processes (beyond multiscale microscopy) because the persistence and intraportlet logic is separate from the interface layer. Adapting the workflow portlet to another scientific discipline is simply a matter of substituting appropriate headings in the portlet.

Figure 8.6 is a high-level outline of a typical multiclass workflow that is initiated by the enduser. From the main scientific process workflow controller portlet, the user launches an external application (in this example, a Telemicroscopy control session). Session information (i.e., authentication and data-management parameters) that is curated by the portal upon login is passed to the application at runtime via ATOMIC tools and services. Using those parameters, the application initiates a lower-class workflow, in this example a Pegasus planned workflow for parallel tomographic volume reconstruction that is executed by a Condor DAGMAN [97]. Next generation ATOMIC Web/Grid service-based implementations will also further allow dynamic notifications of progress at both the level of the external application and the main portal workflow. All of this takes place in a seamless user environment where the typical overhead of transitioning between different workflow classes is passed neither to the end user nor to the application developers. For example, we anticipate the inclusion of more robust resource and network discovery tools within Pegasus without modification of current applications.

This requirement will be particularly relevant as more complex, real-world workflows are enabled. The example in Figure 8.7 illustrates an end-to-end feedback-driven data-collection scenario that has been requested from the microscopy community. This requirement clearly amplifies the need for extensive coordination of different classes of workflow tools, from high-level workflow-management tools (i.e., portal) to low-level planners (Pegasus). In particular, this example illustrates the requirement for a coordinated mixture of resource usage models, including on-demand, traditional batch, and large memory.

8.9 The Biomedical Informatics Research Network (BIRN)

The Biomedical Informatics Research Network (BIRN) is an infrastructure project of the National Institutes of Health. A main objective of BIRN is to foster large-scale collaborations in biomedical science by utilizing the capabilities of the emerging cyberinfrastructure. Currently, the BIRN involves
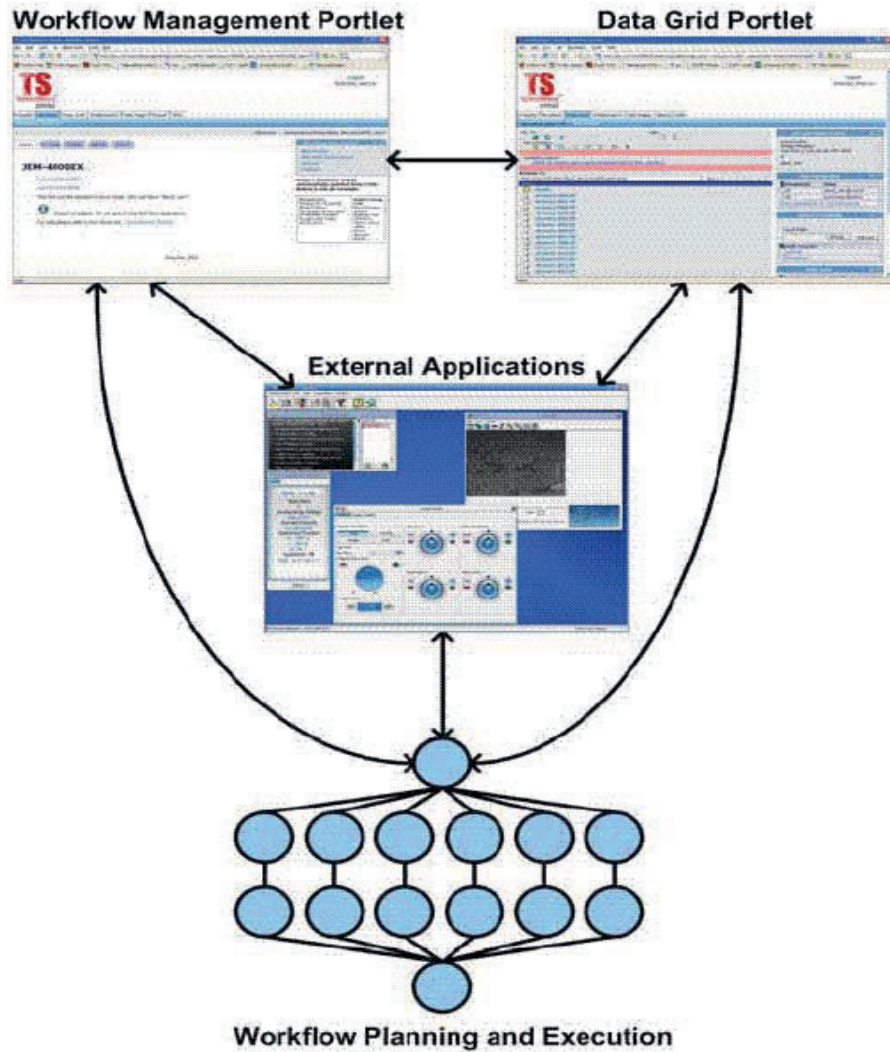
Figure 8.6: Telescience portal. The Telescience portal is a rich user environment, where generalized session information and persistence logic allow actions in the scientific-process-driven workflow management portlets to be reflected in other portlets (i.e., data management portlets). The session information and logic can also be preserved and further delegated to downstream workflow controllers and external applications while retaining notifications for all components.

a consortium of more than 20 universities and 30 research groups participating in testbed projects centered around brain imaging of human neurological

1. User logs into Telescience Portal using username and password
2. User GSI proxy credential is created on TeleAuth Server
   - Telescience credentials are accepted by the following organizations:
     - TeraGrid (computation)
     - OptIPuter (advanced networking and visualization)
     - Telemicroscopy (remote and collaborative instrumentation)
     - Cell Centered Database/CCDB (federated metadata / data management)
3. User builds/registers new project from Portal (transparently interacting with CCDB)
   - After project registration, metadata is collected, auto-parsed, and deposited into CCDB (at every step), from both within the Portal and external applications
4. User uses Portal based digital lab notebooks to "record" non-computational process (transparently interacting with CCDB)
5. User (and possibly remote collaborators) launches collaborative instrumentation control session (Telemicroscopy), user information and credentials are passed to Telemicroscopy software
   a. Preliminary/Preview data is acquired in semi-real-time (during instrument operation) for a particular region of interest (ROI)
      - Total data size is < 500MB
      - Telemicroscopy provisions primarily local resources based on network speed
      - Simple processing workflow plan (DAX) is automatically generated for tomography reconstruction (i.e., TxBR \cite{Lawrence01})
      - Preview data is reconstructed (DAX -> DAG) into 3D volume using selected resources
      - Data flows directly from instrument to computational resources
   b. Preliminary/Preview 3D volume is segmented for visual inspection
      - Telemicroscopy provisions primarily local resources based on network speed
      - Simple processing workflow (DAX) plan is generated for segmentation and visualization (i.e., using ITK and VTK filters – Watershed, Level Set, etc.)
      - 3D volume is segmented and visualized (DAX -> DAG) using local resource
      - Data flows directly from previous step to currently selected computational resources
   c. Data is visualized by user (and remote collaborators) via Portal visualization applications
   d. Decision is made to continue searching for specimen of interest (preview data possibly collected again), tune instrument parameters, or to acquire full resolution data from current ROI
   e. Fully automated, full resolution data collection is executed
      - Collected data is automatically routed to appropriate data grid location and permissions are set accordingly
6. Users ends Telemicroscopy session
7. Users launches Portal tools for image pre-processing
   - Total data size is ~10GB
   - Portal provisions resources based primarily on available computational horsepower. Both local and external resources (i.e. TeraGrid) are utilized.
   - Pre-processing task(s) are selected by user and processing plan (DAX) is automatically generated by Portal
   - Processing plan is converted to DAG and executed
   - Data flows directly from data grid to computational resources
8. 3D volume from full resolution, pre-processed data is computed.
   - Total data size is ~50GB
   - Portal provisions resources based primarily on available computational horsepower. Both local and external resources (i.e. TeraGrid) are utilized.
   - Desired Tomography algorithm(s) is selected by user and processing plan is automatically generated (DAX) by Portal
   - Processing plan is converted to DAG and executed
   - Data flows directly from data grid to computational resources
9. 3D volume is segmented for visualization
   - Portal provisions resources based primarily on available computational horsepower, automatically selected from both local and external resources (i.e. TeraGrid)
   - Desired segmentation methods (i i.e., using ITK and VTK filters – Watershed, Level Set, etc.) are selected by user and processing plan(s) is automatically generated (DAX) by Portal
   - Processing plan is converted to DAG and executed
   - Data flows directly from data grid to computation resources
10. Data is visualized by user (and remote collaborators) via Portal visualization applications targeted at high-memory resources
    - Data flows directly from data grid to high-memory visualization resources
    - Data is collaboratively refined and/or annotated
11. CCDB project maintenance is performed
12. Select data is "published" and available for communities at large (i.e. BIRN)

Figure 8.7: End-to-end feedback-driven data-collection scenario.

disease and associated animal models. The promise of the BIRN is the ability to test new hypotheses through the analysis of larger patient populations and unique multiresolution views of animal models through data sharing and the integration of site independent resources for collaborative data refinement.

In order to support the collaborative nature of scientific workflows from BIRN, a critical component is the collaborative project management portlet that allows for the creation of multiple independent projects, each able to have its own process management workflow (workflow controller portlets) managing the project's experimental process. The first step in the experimental process (a typical use case is portrayed in Figure 8.8) is the collection of the primary research data. Within the BIRN testbeds, these data are collected and stored at distributed sites where researchers maintain local control over their own data. Once the imaging data has been stored within the BIRN data Grid, authorized users from any collaborating site must be able to process, refine, analyze, and visualize the data. In order to satisfy these requirements, BIRN researchers are utilizing multiple interapplication pipeline environments such
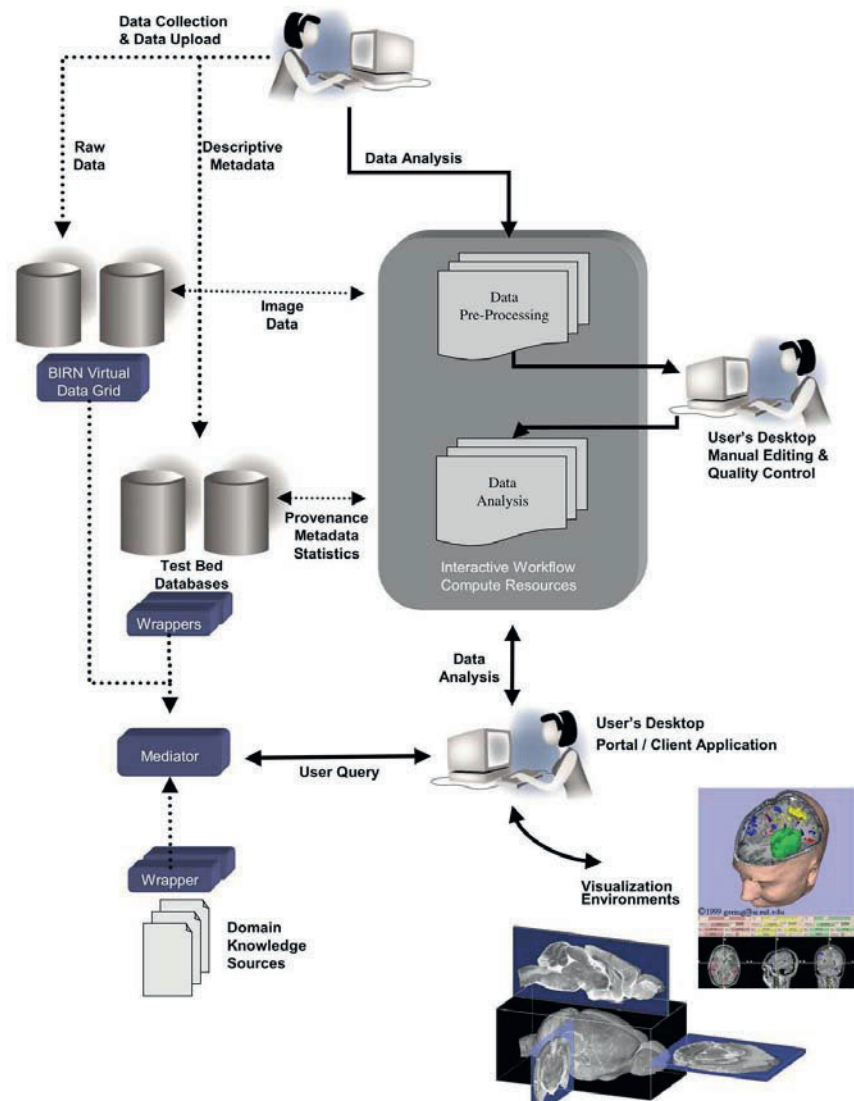
Figure 8.8: Use case scenario for a collaborative experiment within the BIRN cyberinfrastructure. The use case follows the data flow all the way from data collection, to the BIRN data Grid, through interactive processing stages, which is then followed by a query through the data integration layer.

as Kepler and the LONI pipeline. BIRN users are currently able to transparently access and process data through the BIRN portal, a workflow and

application integration environment where applications can interact with the BIRN data Grid allowing researchers to visualize and perform analysis on data stored anywhere within the BIRN data Grid. However, an important objective of the BIRN initiative is to provide the researcher with seamless access to the computational power required to perform large-scale analyses through the use of complex interactive workflows. The sequence of steps within a typical analysis pathway can consist of multiple workflows (e.g., there might be separate application pipelines for the data pre-processing and post-processing) and interactive procedures (e.g., manual verification of the data pre-processing). This complex interactive workflow may be required to utilize distributed computing resources (e.g., to expedite the processing of multiple data sets) while also allowing the researcher to perform any interactive procedures that are required. However, the translation of these workflows from current application pipeline environments to workflows that are able to take advantage of distributed resources is not always straightforward. The current portal environment allows for the management and execution of individual application pipelines that utilize their own execution models. For example, workflows developed in the LONI pipeline or Kepler environment will execute on a specified server or cluster environment, however, they are not able to take full advantage of a distributed and heterogeneous pool of resources. In order to fully enable these application pipelines to access distributed Grid resources, the workflows defined by these pipeline environments are being extended to facilitate interoperability with intra-application workflow planners. As data are processed, intermediary data and ultimately the final results are stored back in the BIRN data Grid along with metadata describing the full history (provenance) of the data files. Much of the metadata information, along with results from statistical analyses, are stored in databases being deployed at all testbed sites.

Similarly to the Telescience use case, BIRN researchers require the coordination and interoperability of workflow tools of many classes. In addition, due to the increased interactivity requirements of their research, many pipelining/workflow tools have already been developed. BIRN must therefore, also ensure that all the pipeline tools interoperate with each other as well as the different classes of workflow tools. The portal (and ATOMIC) play a critical role in the interoperability of pipelining tools by ensuring that the contextual information is compatible and deliverable to each unique pipeline tool. Similarly to the Telescience case, these pipeline/workflow applications are capable of launching large-scale analyses via workflow planner and execution engines. As the BIRN cyberinfrastructure matures, the application integration and workflow environment interoperation must also be enhanced and extended so that researchers are able to more efficiently perform large-scale analyses of their data.

8.10 Discussion

If we define an end-to-end scientific workflow to encompass all the steps that take place between data or information acquisition and the final discoveries that take place as a result of the initial data acquisition, it is clear that no single workflow tool or system is adequate to address this need. It can only be addressed through the integration of several different interoperable tools.

The Telescience and BIRN projects demonstrate this integration (and interaction) of components from different perspectives of the entire end-to-end workflow spectrum. Telescience-based workflows require minimal user interactivity and control but require the coordination of many heterogeneous computational and data resources across several VOs. The majority of the BIRN-based workflow tools, however, are highly interactive but are launched across a more enclosed set of physical resources. In both cases, the portals are critical to the presentation of a unified workflow management environment.

More important than integration of different workflow tools, however, is the development of systems that will maintain interoperability of state and process information between the different workflow classes. Future development will move beyond simple integration to the development of tools to maintain that interoperability in a generalized manner. This is critical as new workflow tools continue to emerge. Within this vision, the portal will continue to serve as the unifying fabric where these integrated workflow technologies will be organized and made to interoperate with the various high-level interaction tools for experimental/bench processes (including database and digital lab notebooks) and also with interactive visualization and/or analysis tools for user intervention at decision points.

Acknowledgments

9

Dynamic, Adaptive Workflows for Mesoscale Meteorology

Dennis Gannon, Beth Plale, Suresh Marru, Gopi Kandaswamy,
Yogesh Simmhan, and Satoshi Shirasuna

9.1 Introduction

The Linked Environments for Atmospheric Discovery (LEAD) [122] is a National Science Foundation funded[1] project to change the paradigm for mesoscale weather prediction from one of static, fixed-schedule computational forecasts to one that is adaptive and driven by weather events. It is a collaboration of eight institutions,[2] led by Kelvin Droegemeier of the University of Oklahoma, with the goal of enabling far more accurate and timely predictions of tornadoes and hurricanes than previously considered possible. The traditional approach to weather prediction is a four-phase activity. In the first phase, data from sensors are collected. The sensors include ground instruments such as humidity and temperature detectors, and lightning strike detectors and atmospheric measurements taken from balloons, commercial aircraft, radars, and satellites. The second phase is data assimilation, in which the gathered data are merged together into a set of consistent initial and boundary conditions for a large simulation. The third phase is the weather prediction, which applies numerical equations to measured conditions in order to project future weather conditions. The final phase is the generation of visual images of the processed data products that are analyzed to make predictions. Each phase of activity is performed by one or more application components.

The entire linear processing of these four phases is done at fixed time intervals, which are not necessarily connected to what is happening with the

---

[1] LEAD is funded by the National Science Foundation under the following Cooperative Agreements: ATM-0331594 (Oklahoma), ATM-0331591 (Colorado State), ATM-0331574 (Millersville), ATM-0331480 (Indiana), ATM0331579 (Alabama in Huntsville), ATM03-31586 (Howard), ATM-0331587 (UCAR), and ATM-0331578 (Illinois at Urbana-Champaign).

[2] University of Oklahoma, Indiana University, University of Illinois at Urbana-Champaign, University Corporation for Atmospheric Research (UCAR), University of Alabama in Huntsville, University of North Carolina, Howard University, and Colorado State University.

weather. The orchestration of the four phases of the process is done with large, complex scripts that are nearly impossible to maintain and enhance, except by very few experts or the original authors.

The LEAD vision is to introduce adaptivity into every aspect of this process. In fact, there are four different dimensions to adaptivity that are important to LEAD:

- Adaptivity in the way the simulation computation uses a multilevel coarse-to-fine forecasting mesh (to improve resolution)
- Adaptivity in the way the instruments gather data based on the needs of the simulation
- Adaptivity in the way the entire assimilation and simulation workflow uses computational resources to its advantage
- Adaptivity in the way the individual scientist can interact with the prediction workflow

To understand these concepts as they relate to the LEAD mission, we discuss them briefly below, and then in later sections of this chapter, we describe how these goals impact the workflow system.

Adaptivity in the Computation

In the simulation phase of the prediction cycle, it is essential to introduce adaptivity in the spatial resolution to improve the accuracy of the result. This involves introducing finer computational meshes in areas where the weather looks more interesting. These may be run as secondary computations that are triggered by interesting activities detected in geographic subdomains of the original simulation. Or they may be part of the same simulation process execution if it has been reengineered to use automatic adaptive mesh refinement. In any case, it is essential that the fine meshes track the evolution of the predicted and actual weather in time. The location and extent of a fine mesh should evolve and move across the simulated landscape in the same way the real weather is constantly moving.

The Adaptive Data Collection

If we attempt to increase the resolution of a computational mesh in a local region, it is also likely that we will need more resolution in the data gathered in that region. Fortunately, the next generation of radars will be lightweight and remotely steerable [121]. That means it will be possible to have a control service that a workflow can use to retask the instruments to gain finer resolution in a specific area of interest. In other words, the simulation will have the ability to close the loop with the instruments that defined its driving data. If more resolution in an area of interest is needed, then more data can be automatically collected to make the fine mesh computationally meaningful.

Resource Adaptivity

There are two important features of these storm prediction computations that must be understood. First, the prediction must occur before the storm happens. This better-than-real-time constraint means that very large computational resources must be allocated as predicated by severe weather. If additional computation is needed to resolve potential areas of storm activity, then even more computation power must be allocated. Second, the computations in these predictions often require *ensembles* of simulation runs that perform identical tasks but start from slightly different initial conditions. As the simulations evolve, the computations that fail to track the evolving weather can be eliminated, freeing up computational resources. These resources in turn may then be used by a simulation instance that needs more power. An evaluation thread must be examining the results from each computation and performing the ensemble analysis needed to gather a prediction. In all cases, the entire collection of available resources must be carefully brokered and adaptively managed to make the predictions work.

The Experiment: Adapting to Scientific Inquiry

The final point at which LEAD attempts to depart from tradition and to change the paradigm of meteorology research is the way the project intends to allow the research scientists and students to interact with the components of the system. The philosophy of LEAD is to allow users to access weather data and to launch workflows through a portal. From the portal, the user can select data and then instantiate a workflow from a precomposed library of workflows to analyze the data, or the user may create new workflows on the fly by composing existing analysis and simulation components. The LEAD workflow system needs to be completely integrated into a framework for conducting scientific experiments. The experiments should be repeatable, and consequently every step that a workflow takes must be recorded and all intermediate data must be saved. A scientist should also be able to interact directly with the workflow, allowing the execution path to be interrupted and sent in a new direction.

To completely understand LEAD as a platform for research, it is essential to understand the LEAD data architecture, so we devote the next section of this chapter to an overview of that topic. In the sections that follow, we will describe the requirements that this litany of data and adaptability requirements places on the LEAD workflow system. This will be followed by a discussion of the current approach to meeting these requirements and finally an analysis of challenges that lie ahead.

9.2 The LEAD Data and Service Architecture

Every aspect of the LEAD project is dominated by data: capturing it, storing it, moving it, cataloging it, transforming it, and visualizing it.

The data products used in LEAD experiments arrive from a variety of sources (Figure 9.1). These include surface observations of temperature, wind, and precipitation from Meteorological Aviation Weather Reports (METAR); upper air soundings data on temperature, pressure, and humidity from balloon-borne instruments; Doppler data from NEXt generation RADars (NEXRAD); image data from Geostationary Operational Environmental Satellites (GOES); and North American Meso (NAM) forecast model data from the National Center for Environmental Prediction (NCEP). These data products are cataloged and stored in servers based on Thematic Realtime Environmental Distributed Data Services (THREDDS) [108] and can be accessed using the OPeNDAP and Common Data Model (CDM) protocols. The local THREDDS catalog at each site provides the basic metadata about products that reside at that site. If a user knows what to look for, there are tools to locate a site and download a specific data set.
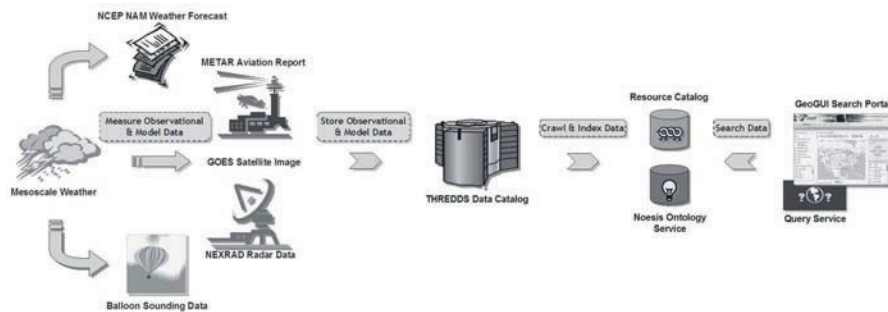

Figure 9.1: Collection and management of observational data in LEAD.

The philosophy of LEAD is to provide the ability for users to discover data based on queries about their content rather than name and location. This is analogous to being able to use a search engine to find information rather than having to know all possible URLs and filenames of the files that contain what you seek. Where it differs from a Web search engine is that LEAD queries are based on metadata that conform to a metadata schema specifically designed for LEAD. Hence queries do not return unrelated hits.

To start the search for data, the user interfaces with the LEAD portal. The portal contains several tools, including the *Geo Reference Interface* (GeoGUI), which allows the user to select a rectangular region of the map, a date range, and other attributes of the data. This forms the user query, which is sent to the Query service. Another Web service, called the *Resource Catalog* [385], keeps track of many important LEAD resources. Among others, it contains an index of the contents of all the THREDDS catalogs it knows about. This index is built by crawling the THREDDS catalogs and capturing metadata in the same way a Web search engine crawler indexes Web page content. A

third service that is important to the LEAD data architecture is the Noesis Ontology service. The Query service uses the Ontology service to map the terms in a query to those that conform to the LEAD schema vocabulary and uses this to interrogate the Resource Catalog for the sought-after data.

The LEAD architecture dictates that every data object selected upon searching shall be saved on the LEAD Grid for the users' future use. A unique ID is created for the data objects and they are archived for future use. Subsequently, a name resolver service is used to materialize the data when requested. LEAD is built on the basis of service oriented architecture (SOA) based on Web services. It is organized into three layers. At the bottom level are fundamental services that provide access capability to Grid resources. These include the Grid Resource Allocation Manager (GRAM) and the GridFTP file transfer service provided by Globus [144], security services for authentication and authorization, and data location and access services such as the Data Replication Service (DRS) and Open Grid Service Architecture's Data Access and Integration service (OGSA-DAI) [233].

The middle tier of services provide data and metadata management for users, notification services, and workflow execution and monitoring capability. The myLEAD [359] service is a flexible, personalized data-management tool that is used to record metadata about data products generated and used during scientific investigations and education activities. MyLEAD helps tie multiple components of the SOA together. As a user runs an experiment, resulting generated data are stored on the LEAD Grid and cataloged by myLEAD in the user's space. Notification messages generated during the course of workflow execution are captured as metadata and stored as provenance for the experimental run. The notification system is based on WS-Eventing [62] and allows mediation between WS-Notification (used by Globus) and the WS-Eventing standard.

At the top level of the SOA stack are the application services that form the building blocks for the scientific investigation and wrap scientific tasks such as FORTRAN executables. These Web services are composed into workflows for execution. LEAD workflows and application services are described in greater detail in the following section.

9.3 LEAD Workflow

Workflows in LEAD model the scientific experiment being simulated by the meteorologist. The workflow framework used to compose and execute these experiments needs to support adaptive computation, instrument control, dynamic resource provisioning, and user intervention in order to meet the requirements described in Section 9.1. These properties are explored in greater detail below:

- Workflows are driven by external events. For example, an event from a data-mining agent monitoring a collection of instruments for significant

patterns must be able to trigger a storm prediction. When such a pattern is detected, the miner may send a signal to a specific workflow associated with the particular storm configuration. This should instantiate the necessary workflow or redirect a running workflow to adapt to the changing conditions. External events may also be triggered by changes in resource availability that may significantly alter the number of possible computations in an ensemble run or change the degree to which adaptive refinement may take place.

- Workflows may be long-running. While tornadoes come and go in a matter of hours, hurricanes are tracked over a period of days. A researcher may preemptively launch an experimental workflow to be triggered by an external condition that may take weeks to occur. Therefore, the execution engine for the workflow must be robust and capable of storing the workflow state in persistent storage for long periods of time, and activating it in a timely manner upon the occurrence of the event.
- Workflows should exhibit fault tolerance. In addition to handling event streams, the workflow system should also deal with exceptions that may occur during the workflow execution. Application services in LEAD workflows run FORTRAN programs, which may fail due to, for example, a parameter misconfiguration. In such a case, there should be a proviso to approximate the incorrect parameter or, if possible, identify an alternate application that can execute with the specified configuration and continue with the workflow execution.
- Workflows should be recoverable. Related to exception handling is the ability of the workflow to adaptively recover from a fatal error or a drastic change in requirements. This may mean rolling back to a previous state in the workflow. This capability would also enable users to interact with a running workflow and to dynamically fork a new execution path starting from an intermediate state of the workflow.
- Workflows must be user-friendly. The workflow templates must be composable by the scientist so that they may be easily instantiated by members of the research and educational community having different levels of expertise.

As part of a LEAD experiment, users build a workflow through the XBaya graphical composer [382], which represents application interactions as a flow diagram as shown in Figure 9.2. Each node in this flow graph is an application service that accepts certain data products and configuration files as parameters and generates output data products that may potentially be used as input by other services in the workflow. Edges connecting nodes in the workflow graph represent the flow of the output data of one service to the input of another, forming a virtual data-flow graph. The application service is capable of fetching the input data products using the unique ID assigned to them by the data services. When an application is launched by its corresponding application service, the service monitors the execution of the application and

publishes a notification on its status to an event channel. This event stream is subscribed by myLEAD and other monitoring tools, such as the XBaya composer (which doubles as a workflow monitor) and the Karma provenance service [384]. Users can follow the progress of the workflow by watching the arriving notifications. When the application completes its task, the output data products it produces are registered with the data services by the controlling application services and logged in the user's myLEAD space. The unique ID assigned to the data is passed as input to other services connected to the completed service. Since each data product is saved and cataloged within myLEAD, a workflow can be reexecuted starting at any step in its execution trace. LEAD users also have the option of using Kepler (see Chapter 8) as the composition tool. Plug-ins developed for Kepler [344] allow composition of workflows from LEAD application services, and this is suitable for orchestrating short workflows through Kepler's graphical interface.
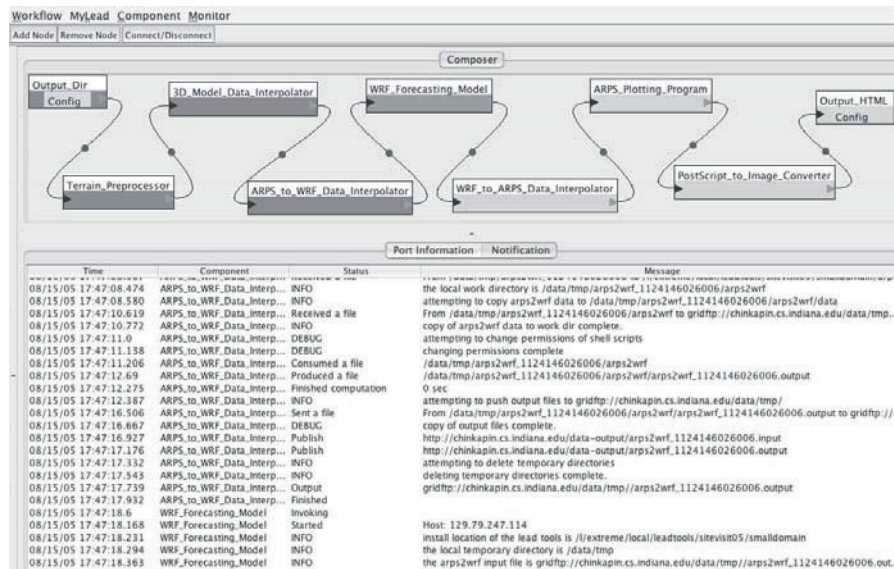

Figure 9.2: A basic simulation workflow showing event output.

Figure 9.2 shows an example workflow that has been executed using the LEAD system. It simulated the devastating hurricane Katrina that occurred in the United States in the summer of 2005. The final output products of the simulation include the visualizations shown in Figure 9.3
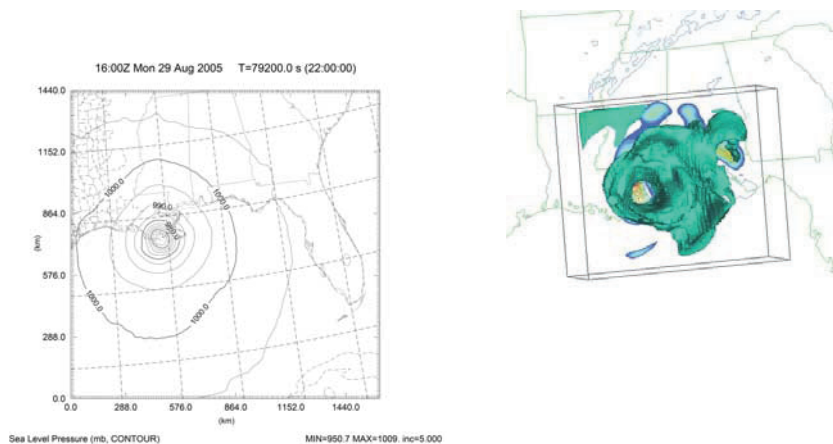
Figure 9.3: On the left is a Katrina simulation of the sea level as the storm approaches. On the right is a 3-D view of simulation data using the Unidata Integrated Data Visualizer (IDV) [306].

9.3.1 Wrapping Tasks as Application Services

The complex weather forecasting applications used in LEAD are first *wrapped* as Web services. *Wrapping* an application as a Web service refers to the process of creating a web service interface to invoke and manage an application. This service layer is referred to as an application service. All clients and end users interact with the application through its application service. When an application service is invoked with a set of input parameters, it launches the underlying application with those parameters and returns the output results as part of the service invocation response. The use of application services allows LEAD scientists to leverage the benefits of an SOA and easily compose, monitor, and run complex weather forecasting workflows from the convenience of the LEAD portal [156].   Although writing an application service wrapper for a given application is not difficult for a Web services specialist, it forms a high barrier of entry for most scientists. The *Generic Service Toolkit* [232] makes this task much easier by allowing scientists to provide a high-level description of the application from the LEAD portal and by automatically generating a service for it. This description is in the form of an XML document called the *ServiceMap* document and includes the input and output parameters of the application, the security restrictions for accessing the application, and the soft-state lifetime-management policies of the application service. The Generic Service Toolkit automatically maps these specifications

to elements within a *Web Services Description Language* (WSDL) document
that it creates for the application service. The *Abstract* WSDL or AWSDL
(*abstract* because the WSDL does not refer to a service instance yet) and the
ServiceMap document form a template for creating a service instance and are
registered with the Resource Catalog in support of subsequent instantiations.

Once the AWSDLs for all application services required for a workflow
are available with the Resource Catalog, scientists can proceed to compose
the weather forecasting workflows from the portal using the graphical XBaya
workflow composer. It should be noted that to compose workflows from ap-
plication services, running instances of the services are not required and the
service templates suffice. However, to execute a workflow, all the application
services in the workflow need to be running and accessible by the workflow
engine that executes the workflow.

9.3.2 Sample LEAD Workflow

A typical ensemble weather forecasting workflow used within LEAD is shown
in Figure 9.4 and illustrates the complexity and dynamic nature of such work-
flows. There are four logical stages to the workflow when seen from a meteoro-
logical perspective: preprocessing of static and terrain files for the geographical
region, analysis and mining of current observational weather data, running the
forecast model, and visualizing the prediction. These four stages and the 15
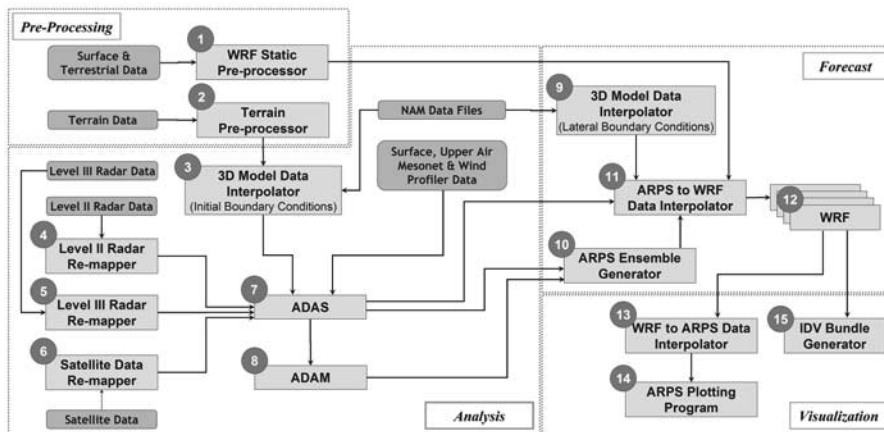services involved in the ensemble workflow are discussed below.

Figure 9.4: A Typical LEAD workflow.

- *Preprocessing.* Service 1, the terrain preprocessor, stage extracts static
  terrain data and service 2, the static preprocessor, extracts the surface

data, such as soil and vegetation type, within the forecast prediction region and pre-processes them into a format compatible with the Advanced Regional Prediction System (ARPS) [486].

- *Analysis.* Static data from preprocessing are interpolated with current NAM forecast model data into a 3D grid for the prediction region is done by service 3, the 3D model interpolator. Dynamic observational data from radars are processed by services 4 and 5, the Level II and Level III Radar data remappers and for satellites by service 6, the satellite data remapper. All these data products are assimilated into the ARPS Data Analysis System, or ADAS in service 7. ADAS performs a 3D gridded analysis of the current atmosphere by combining the observed information from radars, wind profilers, satellites, surface observation networks, and aircraft with a background field created by the 3D model data interpolator. This analysis is performed hourly and examined by a data miner looking for storm signatures in service 8, the Algorithm Development and Mining, or ADAM. When a storm is detected in a certain region, it triggers the subsequent ensemble forecast with pertinent configuration information.

- *Forecast.* The output from the data-mining tool is used in service 10, the ARPS Ensemble Generator, to build the configuration required for the ensemble forecast run. Static terrain data, the ADAS analysis output, and the configuration information are ingested and transformed by service 11, the ARPS to WRF interpolator, into the Weather Research and Forecasting (WRF) model input files. The 3D model data interpolation takes place once again in service 9, the 3D model interpolator, with current weather data and lateral boundary conditions. All of these are used to launch multiple simultaneous runs of service 12,the WRF forecast model, each tuned with slightly different physics to increase the forecast accuracy. The number of parallel ensemble runs can range in the hundreds. The WRF runs perform storm, mesoscale, and synoptic weather prediction that can be used to study convection, baroclinic waves, boundary layer turbulence, and real-time weather phenomena.

- *Visualization.* Visualization and postprocessing tools require that the WRF forecast output to be converted back into the ARPS data format. this is done by service 13, the WRF to ARPS Interpolator. The output is used in service 14, ARPS plot, to automatically generate contour and vector plots of 2D cross sections and vertical profiles. Users can also interactively view the output in 3D using the Integrated Data Viewer (IDV) client tool. An IDV bundle of all relevant data from the forecast is created for this purpose by service 15, IDV bundle generator.

Once composed within the XBaya workflow composer, this flow diagram can be translated into a Business Process Execution Language (BPEL) [24] document, which is executed by a BPEL engine. The composer can also compile the same graphical workflow into a Jython script that can be run as a

stand-alone workflow script. As mentioned earlier, the workflows can also be composed and executed by Kepler.

9.3.3 Configuring Workflow and Application Service Parameters

Large meteorological applications have large and complex parameter sets that are encoded as FORTRAN *namelist* input files. Services such as ADAS and WRF may have several hundred parameters, only a few of which users may wish to modify frequently. Depending on the user's expertise and requirements, a different subset of parameters may need to be modified. To efficiently support changes to a subset of parameters, default values are assigned to the parameters of an application service. When invoking the service, a document containing only the changes relative to the defaults is sent as the parameter.

LEAD users have been divided into four categories based on their domain skill level and the flexibility they require in reconfiguring the research applications. *Category I* users are modelers and application scientists who primarily conduct research on improving a model's capability. These users intend to change the application source code and run their modified applications in the LEAD environment. *Category II* users are atmospheric scientists, graduate students, and operational weather forecasting personnel, who will compose and launch workflows from available applications services. These users will experiment with different sets of input conditions for workflows. *Category III* users are primarily educational users who will perform simulations to understand and learn atmospheric phenomena and will run pre-composed workflows with minimal if any changes in their input configuration. *Category IV* users are casual browsers who will only browse through and visualize completed workflow results.

The majority of the LEAD users and much of the general atmospheric community are Category II users who rarely change the application source code and are content to run the executables in different modes by changing the configuration parameters in the FORTRAN namelist files. These changes in parameters force changes in input observational data and resource requirements at run time.

The namelist parameters in LEAD applications can be classified into five sets:

- The first set of parameters is a *mandatory set* of user-provided parameters that are present in most input files of meteorological services. Examples of such parameters include forecast domain *size*, its *location*, and the *resolution* of the forecasting grid, among others. These parameters play an important role in determining the resource requirements for the workflow execution.
- The second set of parameters is an *optional set* of user-supplied parameters, most of which are service-specific configurations. Default values of these parameters are provided by the application developers, and users

may view and modify them. A model's physics is an example of these para-
meters. The number of optional parameters presented to the user varies
with each category of users. Advanced users familiar with the application
are presented with a broad range of parameters, while novice users have
a minimal set of parameters to modify.

- The third set of parameters relates to *file handling*. FORTRAN applica-
  tions read in input data filenames and locations from namelist files, and
  the applications can only read files locally available in the compute ma-
  chine. After the required input data files for a service are staged on the
  compute servers, these input parameters are modified to reflect the data
  file locations and names.
- The fourth set of parameters are used to assist with *resource scheduling*.
  An MPI-enabled application, for example, may indicate the processor dis-
  tribution in the X and Y directions of the forecasting grid to make the
  computations optimally faster. These parameters are configured after the
  user has selected the forecast domain and the workflow has been allocated
  resources it can use.
- The final set of parameters are those that are always *defaulted* but non-
  etheless need to be supplied to the application.

The various sets of parameters present certain challenges to the LEAD
workflow system. First, the user-editable and cross-cutting parameters have
to be extracted from the workflow dependency graph after the user creates or
modifies a workflow. Care has to be taken to keep the cross-cutting parameters
consistent; otherwise the workflow may produce incorrect results or not run
at all. Second, each user category has to be presented with a different set
of modifiable parameters. User interfaces have to be dynamically generated
based on user category and the workflow graph. Third, these parameters have
to be modified at multiple stages of workflow creation and execution, and
propagate through to the different layers of the SOA. Finally, given the rapid
evolution of different versions of an application, the parameter schema has to
adapt to the changing application parameter set.

When a service provider defines an application service, in addition to the
ServiceMap document, they need to register a set of defaulted namelist files
for each user category that is stored in the Resource Catalog. As we shall
see in the next section, the *Experiment Builder* portlet in the LEAD portal
provides a rich interface for different categories of users to easily specify the
input parameters and data required to run the workflow.

9.3.4 Executing LEAD Workflows

There are several steps that take place before a composed workflow can be
executed by the workflow engine: The parameters for the application ser-
vices need to be configured, the resources required by the services need to
be provisioned, and the services themselves need to be instantiated if needed.

When a user selects a workflow to launch, the *Workflow Configuration Service* (WCS) extracts the cross-cutting parameter dependencies by contacting an *Input Parameter Library* and identifies user-modifiable input parameters by analyzing the workflow dependency graph. WCS then downloads the template namelist files relevant to the user category from the Resource Catalog, assimilates input-parameters that need to be configured by the user, and presents the parameters through a portlet interface generated dynamically. Once the user has modified and verified the parameters of the workflow, the updated template parameter files are merged into a single input parameter file for each application service and are stored in the user's myLEAD space for that workflow (experiment). These parameters form metadata in myLEAD that can be used to search for experiments.

After the input parameters and data are specified, the resource requirements for the applications in the workflow have to be determined. The need for "faster than real time" prediction by the workflow challenges the responsiveness of resource allocation to the dynamic behavior of Grid resources during the workflow's life cycle. As seen earlier, an ensemble workflow can have anywhere from a few to hundreds of services being simultaneously invoked as it progresses. Unique constraints such as large data transfers, real-time data streams, huge computational demands, strict deadlines for workflow completion, the need to steer external radars to collect new data, and responsiveness to weather phenomena drive the need for an adaptive *Resource Provisioning Service* (RPS) that can coordinate across different types of resources to meet soft real-time guarantees. The service needs to dynamically analyze the behavior of applications and workflows to predict resource requirements and track the availability of computational, network, and data resources on the Grid to schedule resource coallocations. Performance and reliability metrics may be used to establish a simple performance contract for a workflow and enable on-demand execution and guaranteed completion of workflows within a specified time range.

Currently, resources are statically allocated "by hand" within LEAD. We are developing a dynamic resource allocation and scheduling strategy as illustrated in Figure 9.5. In step 1, the *Experiment Builder* portlet in the LEAD portal provides the WCS with the selected workflow, its parameters (previously configured and saved in the myLEAD space), and the location of input data products. Next, the WCS contacts an *Application Performance Modeling* service to obtain a performance model for each application in the workflow (step 2). The WCS then determines the resource requirements for each application based on the input configurations and data sets provided by the user. Once the application resource requirements are established, the WCS requests that the RPS allocate the required resources (step 3). Based on the resource requirements and availability, RPS reserves resources for each application in the workflow. Running application instances register their *Concrete* WSDL (CWSDL) with the Resource Catalog, and the WCS can determine if application services required by the workflow are already created (step 4). If so, the

WCS *reconfigures* them to use the new set of resources reserved for them by RPS. This is done by updating the resource requirement namelist parameters for that application. If the required application services were not available, WCS requests the *generic application factory, or GFac* (discussed in Section 9.3.5), to create an instance of the application service (step 5). GFac instantiates and returns the CWSDL for the newly created application service to the WCS, which then *configures* the service (step 6). After the necessary application service instances for the workflow have been selected and configured, the WCS returns their CWSDLs to the Experiment Builder portlet (step 7). The portlet uses the CWSDLs, the application namelist parameters, and the input data products to request that the Workflow Engine execute the workflow (step 8).
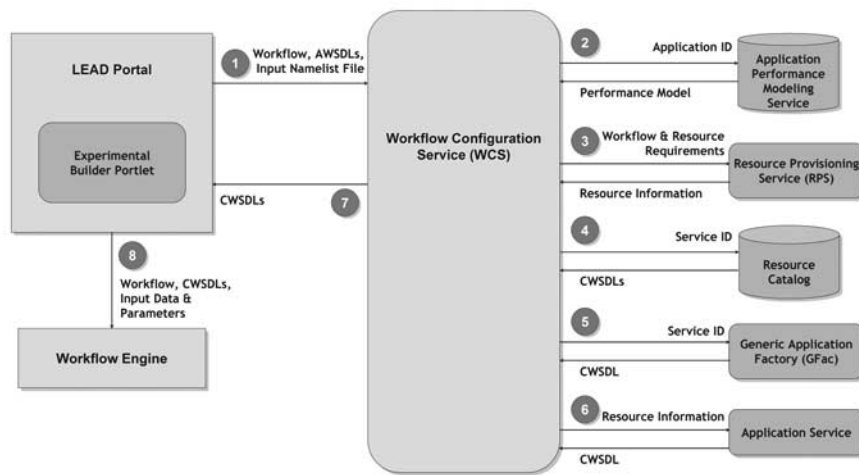
Figure 9.5: The architecture of workflow scheduling and execution.

9.3.5 Creating Application Services On-Demand

When an application service is invoked with a set of input parameters, it invokes the underlying application with those input parameters and returns the output results. By wrapping applications as application services, scientists can easily compose, monitor, and run complex workflows consisting of scientific applications. However, these workflows require their constituent application services to be *available* at the time of workflow execution. At the same time, in large scientific communities such as LEAD, it is unrealistic to keep a large number of persistent application services that entail a significant commitment of resources and support infrastructure. However, it is possible to support a

small number of persistent *generic application factory services* (GFacs) [232] that can can create instances of any application service on-demand (just in time) during a workflow execution in a way that is completely transparent to the users. This provides *highly available* application services without actually requiring them to be persistent. Before GFac creates an application service instance on a host, it first starts a *generic service instance* on that host by calling a generic service binary that is preinstalled on that host. The installation of the generic service binary is a one-time process executed on potential application service hosts. GFac then provides the generic service instance with the *ServiceMap* configuration document for the application service retrieved from the Resource Catalog. Using service ports defined in the ServiceMap document, the generic service instance configures itself to *become* the application service. The generic service instance (now application service) then generates its CWSDL and registers it with the Resource Catalog. Figure 9.6 illustrates the process above. In step 1, the WCS sends a message to GFac containing the fully qualified name of the application service. In step 2, GFac gets the ServiceMap document for the application service from the Resource Catalog. In step 3, GFac creates a generic service instance on the remote host using Globus GRAM [102]. In step 4, the generic service instance configures itself using the ServiceMap document to become the application service instance, generates its CWSDL, and registers it with the Resource Catalog. In step 5, GFac obtains the application service instance's CWSDL from the Resource Catalog, and returns it to WCS in step 6. In step 7, the workflow engine uses the CWSDL passed to it by the WCS to invoke the application service instance directly.
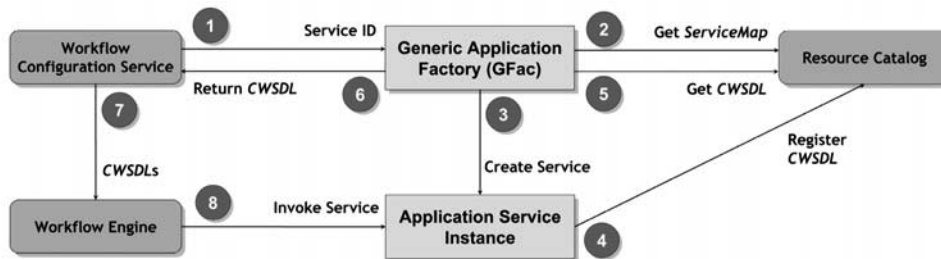

Figure 9.6: Interaction with the factory service.

The application services created by GFac can be *reconfigured* at runtime (i.e., during a workflow execution). This is done through the configure operation, which accepts a Web service call with details of the resources on which the application service should run its application. The details are provided as an XML document called a Resource Specification Document (RSD), which contains information such as

- the host on which to run the application,
- the end point reference to the job scheduler,
- the batch queue for running the application,
- the number of processes to start,
- a list of the nodes on this cluster for running the application,
- the number of processors per node for running the application,
- the maximum wall clock time for running the application,
- the maximum CPU time for running the application, and
- the maximum memory in kilobytes allowed per process.

Once an application service instance receives an RSD, it reconfigures itself accordingly and returns a new CWSDL to the client that contains the *Resource Specification* that the application service will use to run its application. The client can then use the new WSDL to invoke the run operation on the application service instance that invokes the application and return the results to the client. It is important to note that the application service supports multiple simultaneous configurations, and different clients can configure the same application service instance differently. Each client will receive a different CWSDL that it can use to run the application according to its own Resource Specification. This allows the same application service instance to be used simultaneously not only indifferent workflows with varying resource requirements but also within a dynamic workflow with constantly changing resource requirements.

9.4 Conclusions

Workflows in the LEAD project have several characteristics that set them apart from many other e-Science workflow problems. First, they are driven by natural events such as severe storms. Second, because storms such as tornadoes and hurricanes are so destructive, it is essential that the forecasts that are the output of the workflows be extremely accurate and that they be produced prior to the storm's impact on human life and property. Finally, LEAD must have workflows that are extremely adaptive. Resource demands can change as the storm changes. There is also a natural feedback that takes place between the workflow services and the instruments that gather data: As a simulation becomes more specific about the nature of an emerging storm, future generations of radars can be automatically targeted to gather more detailed data for the simulation to use to increase prediction accuracy.

LEAD also shares many characteristics with other large-scale e-Science workflow systems. LEAD is based on a service-oriented architecture that is becoming a standard model in e-Science. Yet, LEAD workflows are still composed of community FORTRAN applications that must run in parallel on supercomputers. These applications have enormously complex parameters and large numbers of input files that are difficult to manage. Consequently, simple

data-flow concepts can only be applied at a very high level of abstraction, and the underlying workflow system must manage a great deal of complexity involving resource allocation, application configuration, and parameter management. Security is also a critical component of every project that deals with expensive resources organized into a Grid. LEAD has adopted Grid standards to solve this problem. Finally, like any other modern science, LEAD is very data-intensive. Every aspect of the workflow generates data products that must be cataloged with the LEAD data subsystem. Metadata must be created and made searchable; data provenance must be tracked and cataloged; and quality input and derived data should be maintained. This chapter has not addressed these issues, but several other papers consider these problems in greater depth [122, 123, 200, 360]. While the LEAD project is only in its third year, the team has learned a great deal. Many ideas that seemed practical in theory failed in practice and had to be replaced by more robust models.

9.5 Acknowledgments

10

SCEC CyberShake Workflows—Automating Probabilistic Seismic Hazard Analysis Calculations

Philip Maechling, Ewa Deelman, Li Zhao, Robert Graves,
Gaurang Mehta, Nitin Gupta, John Mehringer, Carl Kesselman,
Scott Callaghan, David Okaya, Hunter Francoeur, Vipin Gupta,
Yifeng Cui, Karan Vahi, Thomas Jordan, and Edward Field

10.1 Introduction to SCEC CyberShake Workflows

The Southern California Earthquake Center (SCEC) is a community of more than 400 scientists from over 54 research organizations that conducts geophysical research in order to develop a physics-based understanding of earthquake processes and to reduce the hazard from earthquakes in the Southern California region [377].

SCEC researchers are integrating physics-based models into a scientific framework for seismic hazard analysis and risk management. This research requires both structural geological models, such as fault models and three-dimensional Earth density models, and a variety of earthquake simulation programs, such as earthquake wave-propagation simulation codes and dynamic fault-rupture simulation applications. The goal of this model-oriented approach to earthquake science is to transform seismology into a predictive science with forecasting capabilities similar to those of climate modeling and weather forecasting.

SCEC research has several common characteristics. The science is collaborative—a wide variety of organizations and disciplines work together. The science is integrative—techniques and approaches from different disciplines are combined in new ways. The science is physics-based—the scientists are continuously trying to incorporate more physics into their models and to ensure that their simulations are consistent with physical laws. The science is model-driven—theoretical results are incorporated into predictive computational models. The science is validated—predictive model results are compared with observation and with each other for validation.

The output data for many SCEC earthquake simulations are predicted ground motions for a specific earthquake. For example, a researcher can model a "scenario" earthquake on the San Andreas Fault and predict the ground motions that will be produced in Los Angeles if that earthquake actually

occurs. While ground motion predictions for a particular earthquake are of significant interest, they are not a solid basis for understanding the earthquake hazards in an area.

To characterize the earthquake hazards in a region, seismologists and engineers utilize a technique called Probabilistic Seismic Hazard Analysis (PSHA). PSHA attempts to quantify the peak ground motions from *all possible earthquakes* that might affect a particular site and to establish the probabilities that the site will experience a given ground motion level over a particular time frame. An example of a PSHA hazard curve at a specific site in Los Angeles is shown in Figure 10.1. Because Los Angeles has widely vary-
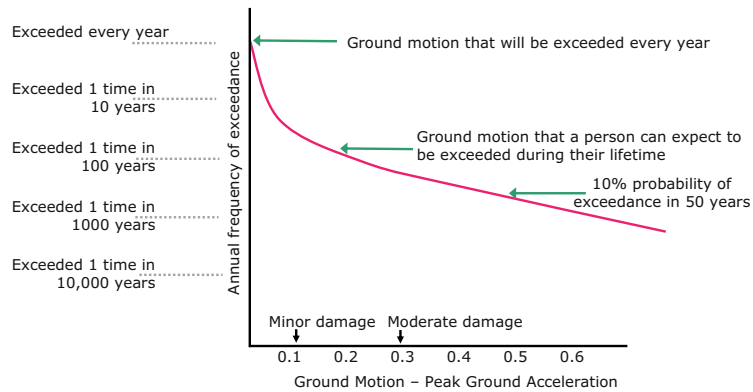

Figure 10.1: Probabilistic Seismic Hazard Curve for the site of Los Angeles City Hall. This curve predicts that this site will experience a Peak Ground Accelaration of 0.5 G about every 500 years.

ing geological regions (mountains, deserts, and sedimentary basins), hazard curves for sites fairly close together can differ significantly. PSHA information is used by city planners and building engineers to estimate seismic hazards prior to the construction of significant buildings, and PSHA results are often the basis for building codes in a region.

Probabilistic seismic hazard curves can be combined into probabilistic seismic hazard maps [278]. To construct a hazard map, one of the two variables used in the curve (either the ground motion level or the probability of exceedance) is fixed, and then color variations indicate how the other parameter varies by location on the map. A set of hazard curves, typically from a set of regularly spaced sites, can be combined into a hazard map by interpolating the site-specific data values and plotting the resulting contours. In the United States, the United States Geological Survey (USGS), as well as several state agencies, publish hazard maps. An example PSHA map, produced by the USGS, and the California Geological Survey (CGS) is shown in Figure

10.2. This map fixes the probability of exceedance at 10% in 50 years, and the color variations indicate predicted levels of peak accelerations, with the darker-colored regions predicted to experience stronger ground motions than the lighter-colored regions.
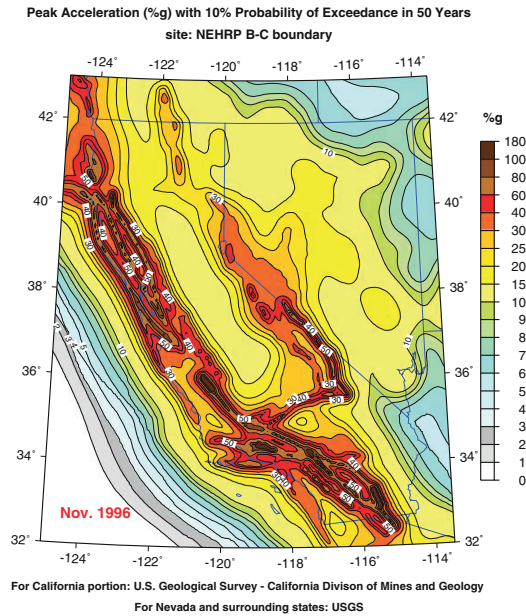

Figure 10.2: This USGS and CGS PSHA map for California and Nevada is based on a large number of PSHA hazard curves. This map fixes the probability of exceedance at 10% in 50 years, and uses color variations to indicate expected peak ground-motion levels throughout the mapped region.

Because of the significant role PSHA information has in public safety, improvements in PSHA techniques are of great interest to seismologists, public safety officials, building engineers, and emergency management groups. PSHA researchers recognize that current PSHA techniques have not fully integrated recent advances in earthquake simulation capabilities. As a result, researchers working on the SCEC Community Modeling Environment Project (SCEC/CME) [226, 378] recently initiated the CyberShake Project to develop new techniques for calculating PSHA seismic hazard curves. The goal of the CyberShake Project is to utilize earthquake wave-propagation simulations to produce the ground motion estimates used in PSHA hazard curves.

The geoscientists and computer scientists working on CyberShake have successfully calculated probabilistic seismic hazard curves for several sites in the Los Angeles area using peak ground-motion values produced by earthquake wave-propagation simulations. This new class of PSHA hazard curves has the potential to transform probabilistic seismic hazard analysis because the earthquake wave-propagation simulations used to produce these new curves generate more physically realistic peak ground-motion values than the techniques used to calculate peak ground motions used in earlier hazard curve calculations.

We refer to all the steps in the CyberShake hazard curve calculation process, including preparation, simulation, postprocessing, and analysis, as the CyberShake computational pathway. The CyberShake computational pathway can be divided into two main computational phases; (1) a high performance, MPI-based, finite-difference earthquake wave-propagation simulation phase; and (2) a postprocessing phase, in which thousands of serial data-analysis jobs must be executed.

We model the CyberShake computational pathway as a scientific workflow to be executed within the SCEC Grid-based computing environment. In the following sections, we describe the CyberShake computational pathway and our efforts to convert this conceptual sequential processing into an executable scientific workflow. We outline issues related to the modeling of computations as workflows and describe where we gained significant benefits from workflow technology.

10.2 The SCEC Hardware and Software Computing Environment

The CyberShake scientific workflows were implemented within the distributed SCEC computing environment that was developed as a part of the SCEC/CME Project [276]. The SCEC/CME computing environment uses a Grid-based architecture that allows us to share heterogeneous computing resources with other collaborating organizations in a consistent and secure manner. The SCEC/CME computing environment is composed of the local SCEC computer resources, including a variety of Linux and Solaris servers, the University of Southern California (USC) Center for High Performance Computing and Communications (USC HPCC) [432]—a large academic Linux cluster— and the National Science Foundation (NSF) TeraGrid [413], a collection of national academic supercomputing facilities.

The SCEC, USC HPCC, and TeraGrid sites are linked into an extensible Grid-based computing environment through the NSF National Middleware Initiative software stack [277]. Grid security is managed using Grid Security Infrastructure (GSI) [463]. Certificate policy was negotiated between the three organizations, allowing acceptance of each other's host and user Grid-security certificates.

The SCEC computing environment provides both computational cycles and significant data storage. Disk storage in excess of 10 TB is available at all sites, including SCEC's local cluster. In addition, the TeraGrid facilities provide more than 100 TB of tape-based data storage for a variety of SCEC data collections.

The SCEC/CME computational system has implemented a workflow software layer based on the Virtual Data Toolkit (VDT) [440] . The Virtual Data Toolkit, in turn, includes the Virtual Data System (VDS) which includes Chimera [148] and Pegasus (Chapter 23). VDT also includes data management tools such as the Replica Location Service (RLS) [88]. An overview of the Grid-based hardware and software used in the CyberShake calculations is shown in Figure 10.3.
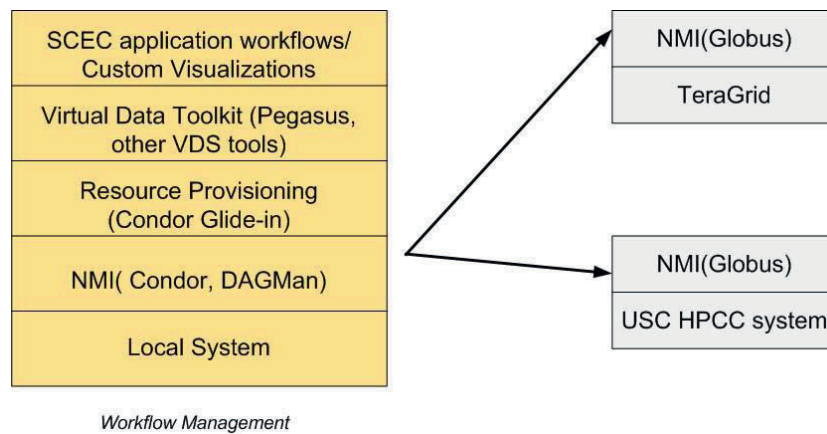

Figure 10.3: The SCEC/CME workflow system software stack, based on the Virtual Data Toolkit, provides SCEC workflows with secure access to a distributed, heterogeneous, Grid-based computing environment.

10.3 SCEC Probabilistic Seismic Hazard Analysis Research

Prior to the start of the CyberShake Project, SCEC researchers outlined a conceptual and computational framework for probabilistic seismic hazard analysis (PSHA) (shown in Figure 10.4). The two primary computational elements in this PSHA model are Earthquake Rupture Forecasts (ERFs) and Intensity Measure Relationships (IMRs).

An ERF is a program that, given a specific region, can produce a list of earthquakes that may occur in the region in the future. An ERF will

also provide a description of each earthquake, including the magnitude of the earthquake, the fault surface that will be involved, and the probability that the earthquake will occur. The list of earthquakes produced by an ERF is based on the active faults in the region, the sizes of the faults, the historical earthquake record in the region, known slip rates for the faults, and other geological and geophysical information.

An IMR can be thought of as an algorithm that defines how earthquake waves decay with distance. Given a specific earthquake and a site some distance away, an IMR will indicate the level of ground motion that will be produced at the site by the earthquake. In more technical terms, an IMR gives the conditional probability that an intensity measure (some function of ground shaking found by engineers to correlate with damage) will be exceeded at a site given the occurrence of a specified earthquake rupture.

Currently, PSHA research uses empirically derived attenuation relationships as IMRs in PSHA. Recently, well-validated 3D wave-propagation simulations have been developed, and the PSHA community has great interest in replacing attenuation-relationship—based IMRs with waveform-based IMRs.

The SCEC CyberShake Project is, we believe, the first project to develop an IMR based on 3D wave-propagation simulations rather than on attenuation relationships. Waveform-based IMRs have not been implemented previously because they require levels of computational, data management, and data analysis that exceed the capabilities of most research groups working in the field.

One of the SCEC/CME Project's working groups has developed a component-based software suite, called OpenSHA [140], that implements standard PSHA models, such as ERFs and IMRs, within a common framework. OpenSHA is a stable and robust suite of software that allows researchers to combine PSHA components in ways never before possible. The CyberShake work uses OpenSHA tools both to produce input data and as to analyze the CyberShake results. OpenSHA implementations of ERFs are used to create the list of ruptures for each CyberShake site. OpenSHA is also used to generate attenuation-relationship—based hazard curves against which we evaluate the CyberShake hazard curves.

10.4 Computational Requirements of CyberShake

SCEC geophysical computing has traditionally been done without using scientific workflow technology. Thus it was not a given that the CyberShake Project needed scientific workflow tools. However, as the scale of the CyberShake computational and data management challenge emerged, we began to recognize that traditional computing methods may not be sufficient.

In order to implement the CyberShake 3D waveform-based IMR, a large number of earthquakes must be simulated. For sites near Los Angeles, current ERFs produce a list of over 20,000 earthquakes within 200 km. Applying
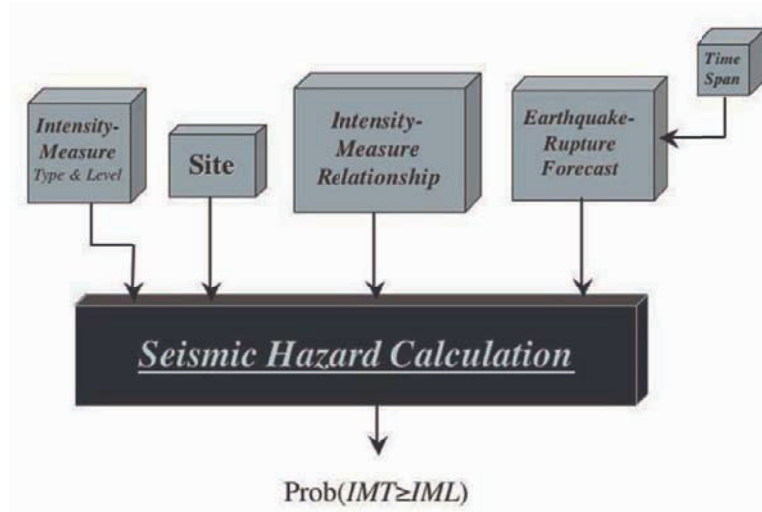
Figure 10.4: The SCEC/CME conceptual model for probabilistic seismic hazard analysis (PSHA) identifies Intensity Measure Relationships (IMRs) and Earthquake Rupture Forecasts (ERFs) as two major computational components involved in PSHA. The CyberShake Project is implementing a new type of IMR.

an attenuation relationship to 20,000 earthquakes is a fairly modest computational challenge, within the capabilities of a desktop computer. However, running state-of-the-art wave-propagation simulations for 20,000 earthquakes is prohibitively expensive in CPU-hours and wall-clock time. The exact computational time required to run an earthquake wave-propagation simulation varies by the size of the volume, the length of time the wave-propagation is simulated, and the frequencies supported by the simulation. Earthquake simulations of approximately the required size and resolution, such as SCEC's Pathway 2 TeraShake simulation [327], require approximately 15,000 CPU-hours and approximately 3 days of wall-clock time. Thus, for the 20,000 ERF ruptures, it would require 300 million CPU-hours and well over 100 years to complete all the simulations needed to calculate a PSHA hazard curve.

While these processing requirements are well beyond the scale of the computer resources available to SCEC, we have not yet represented the full scale of the problem. The numbers underestimate the required calculation because the ERF list of 20,000 earthquakes does not represent the full list of earthquakes that must be simulated. An ERF indicates only the fault surface and the magnitude of the earthquakes that are likely to occur. This is sufficient when using an attenuation relationship. However, when using waveform modeling, one must consider how the earthquake rupture occurs. For example, if the earthquake rupture starts at the bottom of the fault and propagates up-

ward toward the surface, the ground motions at the surface will be larger than if the earthquake starts near the surface and propagates downward into the ground. For a given fault, there are many ways that earthquakes can occur. Each possible, or somewhat likely, earthquake variation must be simulated in order to properly perform the PSHA analysis.

To capture the possible differences between earthquakes in the PSHA analysis, one or more variations of each earthquake mentioned in the ERF must be simulated. For small earthquakes (e.g., magnitude 5.0 or smaller), typically only one variation will be considered. But for large faults there are many ways the fault may rupture, and a reasonable number of rupture variations must be simulated. There is no widely accepted approach for identifying all reasonable rupture variations; however, some basic heuristics have been developed for creating a reasonable number of them. When the heuristics are applied to the ERF list for the Los Angeles area sites, the total number of earthquakes that must be simulated to create a probabilistic seismic hazard curve is over 100,000. At 15,000 CPU-hours per simulation, a fully probabilistic hazard curve calculation would require approximately 1,500,000,000 CPU-hours.

The computation time is not the only challenge; there are also significant data management issues. Each rupture variation will produce two seismograms (horizontal components only), which, depending on the data storage format, may result in one or more seismogram files. These seismogram files and their associated metadata must be managed to support the analysis of the results.

The key to reducing the computational demands of CyberShake PSHA hazard curve calculations to a manageable level was the introduction of a nonintuitive scientific technique for calculating synthetic seismograms called reciprocity. Typically, synthetic seismograms are created through what are termed "forward calculations." Motions are introduced in a volume at the point of the earthquake and the resulting waves are propagated throughout the volume. An alternative method for calculating synthetic seismograms, called reciprocity, can be used [495]. A reciprocity-based approach places a unit force at the site of interest. Then the waves from this force are propagated throughout the volume to "illuminate the volume." The response of the volume to the unit force is saved as strain Green's Tensors (SGTs). Given the SGT data for a volume, it is very computationally inexpensive to calculate a synthetic seismogram for an earthquake located anywhere within the "illuminated" volume using a technique called representation theory or, more informally, seismogram synthesis.

Using a reciprocity-based approach, the computational estimate for calculating a probabilistic seismic hazard curve for a single site is approximately 25,000 CPU-hours. This includes the two unit-force SGT simulations, and the reciprocity-based seismogram synthesis for 100,000 earthquakes. This reciprocity-based technique brings the computational cost of a waveform-based PSHA hazard curve within reach of SCEC computing resources.

There is, as might be expected, a trade-off involved in using a reciprocity-based approach. Reciprocity-based calculations only produce seismograms for

one site and, consequently, only one hazard curve. Since each hazard curve requires approximately 25,000 CPU-hours, producing a small 50 km × 50 km hazard map that requires 625 hazard curves will require approximately 15,625,000 CPU-hours using this approach. The estimates indicate that even using a reciprocity-based approach, it is still prohibitively computationally expensive to produce a waveform-based PSHA hazard map.

10.5 SCEC Workflow Solutions to Key Workflow Requirements

Scientific workflows may be modeled, in general terms, as a set of tasks with data dependencies between them. Scientific workflow tools must then meet three essential requirements: (1) user definition of the tasks and their data dependencies; (2) an execution engine for running the tasks in an appropriate order; and (3) tools for managing the data and metadata that are input and output by the tasks in the workflow.

The SCEC workflow system satisfies the first essential requirement (user definition of workflow tasks and data dependencies) by allowing the user to describe workflows in an abstract form called an abstract Directed Acyclic Graph (DAG). An abstract workflow captures the programmatic and data dependencies in the workflow, but it also imposes some limitations on the workflow, such as no looping. An abstract workflow describes both the program names and filenames in logical, not physical, terms. For example, when an abstract workflow refers to a file, it uses a file ID rather than a physical path to the file. Later, programs in the workflow system will convert the file ID to a physical file pathname. In this workflow definition stage, the SCEC workflow system uses the Pegasus planner (Chapter 23) and Condor's DAGMan (Chapter 22) for mapping and executing the workflows.

To convert an abstract workflow to an executable form, Pegasus (Chapter 23) requires a collection of appropriate configuration files or catalogs, such as those describing the available computer resources (the Site Catalog) and a list of executable programs (the Transformation Catalog). Once this information and the abstract DAG are available, Pegasus can be invoked to do the conversion. Once the executable DAG is available, it can be submitted to the Condor DAGMan (Chapter 22) for execution.

The SCEC workflow system satisfies the second essential workflow requirement (an execution engine for running the tasks) with a series of Globus and Condor tools. Globus GRAM [102] is used as an interface to local resource schedulers. Condor-G [152] manages the remote job submissions by interacting with the GRAM job managers. Condor's DAGMan ensures that the jobs expressed in the DAG are executed in the correct order.

The SCEC workflow system satisfies the third essential workflow requirement (data and metadata management) by using the Replica Location Service (RLS) [88] software to maintain a mapping between logical and physical

file names. Logical File Names (LFNs) are basically ID numbers assigned to files used in SCEC workflows. Physical File Names (PFNs) used in SCEC workflows are typically GridFTP accessible URL's [9]. Metadata are managed through the use of the Metadata Catalog Service (MCS) [386]. The RLS and MCS systems are modular and Grid-enabled. We also utilize a second file preservation system, the Storage Resource Broker (SRB) [41], for long-term storage of valuable data sets.

10.6 Benefits of Modeling CyberShake as Workflows

Implementing the CyberShake workflow on top of a Grid-based architecture provides distributed computing capabilities and the ability to add or remove computing resources from the environment without significant changes to software. The Grid layer provides secure management of job submission and data transfers. The Grid architecture also provides standardized service interfaces to security, job management, resource monitoring, and communication for a heterogeneous environment. This allows our workflows to utilize the standardized interfaces in a heterogeneous computing environment.

As we define our workflow, Pegasus allows us to express the workflow at a high level of abstraction. When the user expresses the workflow and its dependencies, either using VDL (Chapter 17, or in an XML DAG format (DAX), the workflow is specified by referring to logical programs (transformations) and logical files. A significant amount of information can be omitted at the workflow specification stage. For example, the computers and the location of the files to be used are not needed at the workflow specification stage. These details are provided by Pegasus as the abstract workflow is converted to an executable workflow. In addition, Pegasus will augment the workflow with implied but unspecified processing steps. Thus that it can execute within a distributed, Grid-based computing environment. Processing steps such as directory creation, registration of created files into the RLS, and file transfers to and from the program execution hosts are automatically added into the workflow by the Pegasus planner.

Condor DAGMan can analyze the dependencies in a workflow and can run jobs in parallel if there are no dependencies between them. This capability is particularly valuable in a distributed Grid-based environment where there are multiple computing resources available for job execution.

The Condor-G and DAGMan job management tools provide other significant capabilities, such as failure recovery. Condor supports retries of individual failed jobs and provides *rescue DAGs* in cases where the workflow cannot progress any further. The rescue DAG represents the portions of the workflow that have not yet executed, and the DAG can be modified and resubmitted at a later time.

The SCEC workflow system utilizes the common data management practice of separating the logical filename from the physical filename. This tech-

nique helps in two main ways. First, references to the file are not tied to the physical location of the file. When the file is moved, workflow references to the file do not need to be changed (only the mappings in the RLS do). Second, this technique supports copies of files, or file replicas. For each file, multiple versions can be maintained, and the workflow system has the opportunity to select the most appropriate copy.

10.7 Cost of Using the SCEC Workflow System

While the SCEC workflow offers a number of clear benefits, it also imposes a number of requirements, or costs, on system developers and users. These costs are distinct from the costs of personnel or hardware.

First, establishing and maintaining a widely distributed, Grid-based computing environment requires a significant amount of work, involving issues such as security agreements, certificate exchange, software version coordination, installation, operations, and maintenance. A Grid-based environment provides an outstanding foundation on which to build a workflow system, but it also requires significant investment in system and software maintenance.

The SCEC workflow system requires a significant amount of configuration before a workflow can be executed. Pegasus's ability to work at a high level of abstraction is implemented by utilizing data stores that map between abstractions and actual computing resources. This means that before a workflow can be executed, a series of data stores must be developed and populated. For example, computing resources are defined in a site catalog that defines the available computing resources and describes their capabilities. This needs to be done by hand or with the use of information systems deployed on the resources. Also, each executable program or script used in a workflow (along with its runtime environment information) must be defined in a Transformation Catalog.

Also, all files to be used in workflows must be registered into the RLS and staged at a URL that is accessible by a GridFTP server. This creates a fairly sharp distinction between files "in the system" and files "not in the system." The need to register files in RLS before using them in a workflow puts a burden on users who want to create new files by hand or want to import files into the system. While the data management tools such as RLS provide interfaces for registering files, it has been necessary for us to write user-oriented tools to help users with the data registration tasks.

The SCEC workflow system is designed to execute programs with file-oriented inputs and outputs. Programs that support the standard "Unix" computing model work well within the SCEC workflow system. These programs have common characteristics such as file or piped inputs, quiet execution unless there are problems, zero return on success, and nonzero return on problems. The SCEC workflow system is not designed to execute programs with GUIs or with interactive user inputs.

The SCEC workflow system imposes specific requirements on the programs that will be used in the workflow. To integrate with the data management tools, programs used in workflows should not use hardcoded input or output filenames. The workflow system will dynamically assign LFNs to files as they are created. Many of the SCEC programs used hardcoded filenames. In some cases, we modified the programs so that both input and output filenames could be specified as input parameters at runtime. If this modification was difficult, we developed wrapper scripts that would accept arbitrary input and output filenames. The wrapper script would then rename the files to the hardcoded filenames, call the SCEC programs, and then rename the output file to the file name assigned by the workflow system.

One additional requirement for using the SCEC workflow system is the need to create an abstract workflow (the DAX) before the workflow can be run. In order to create a DAX, the user is faced with a couple of options: (a) use VDL to describe the workflow and then use Chimera to convert the VDL to a DAX; or (b) write code that can construct a DAX directly. Because the SCEC CyberShake workflows were fairly static, we chose to develop a DAX generator program and output our DAXs directly. The other option, using VDL, may be the more general solution. Both of these approaches require training and investment of time by users. Often users are not willing to invest significant training time until the benefit to their science is obvious. In the future, we hope that technologies such as Wings and CAT (Chapter 16) can make it easier to create the large and complex abstract workflows we need.

10.8 From Computational Pathway to Abstract Workflow

A CyberShake hazard curve calculation can be described algorithmically in seven steps, as shown in Table 10.1. We refer to this sequence of seven steps as the CyberShake computational pathway. Each processing step has specific computational and workflow implications.

We began our modeling of CyberShake as a workflow by assembling our programs end-to-end and identifying the data dependencies between them. Figure 10.5 shows the programs involved in the CyberShake hazard curve calculation and their data dependencies.

Our intention was to model our CyberShake computational pathway as an abstract workflow, then model the abstract workflow as a DAX, and then use our workflow tools to convert our DAX into an executable workflow and run it until a hazard curve was completed. However, our workflow eventually was reduced to a small portion of this processing chain.

CyberShake Step 1: Select a Site

Probabilistic seismic hazard curves are site-specific, and thus a natural and meaningful unit of work on the CyberShake Project is a *site*. We perform a

Processing Step Number	CyberShake Simulation Algorithm Description
1	Select a site for which a hazard curve is of interest.
2	Use an earthquake rupture forecast (ERF) to identify all probable ruptures within 200 km of the site of interest.
3	For each rupture, convert the rupture description from the ERF into a suite of rupture variations with slip-time history.
4	Calculate Strain Green's Tensors (SGTs) for the two horizontal components for a volume containing all the ruptures and save the volume data.
5	Using a reciprocity-based approach, calculate synthetic seismograms for each rupture variation.
6	Calculate the peak intensity measure of interest, such as peak spectral acceleration, for each synthetic seismogram.
7	Using the peak intensity measures for each rupture and the probabilities of the rupture, calculate a probabilistic hazard curve.

Table 10.1: Steps in the CyberShake salculations.

series of calculations, and at the end we can calculate one or more hazard curves for one particular site.

Sites selected for our initial CyberShake hazard curve calculations must be in a region for which both a 3D velocity model and an earthquake rupture forecast have been defined. These items are available for most parts of Southern California. Also, to facilitate the comparison with other types of IMRs, we selected sites for which hazard curves currently exist. The selection of sites is currently manual.

CyberShake Step 2: Identify Probable Ruptures

Given a particular site, an ERF is used to create a list of all probable ruptures (and the magnitude and probability of each rupture) within 200 km of the site. Table 10.2 shows six of the initial CyberShake sites and the number of ruptures that an ERF identified within 200 km of each site.

Site Name	Number of Ruptures in ERF within 200 km of Site
USC	24,421
Pasadena	24,870
Downtown Los Angeles	24,620
Port of Long Beach	24,484
Santa Ana Business District	25,363
Whittier Narrows Golf Course	25,056

Table 10.2: Initial CyberShake sites.

Steps in the Cybershake activity showing Software Components, Data Flow, Workflow and size of various files and elements
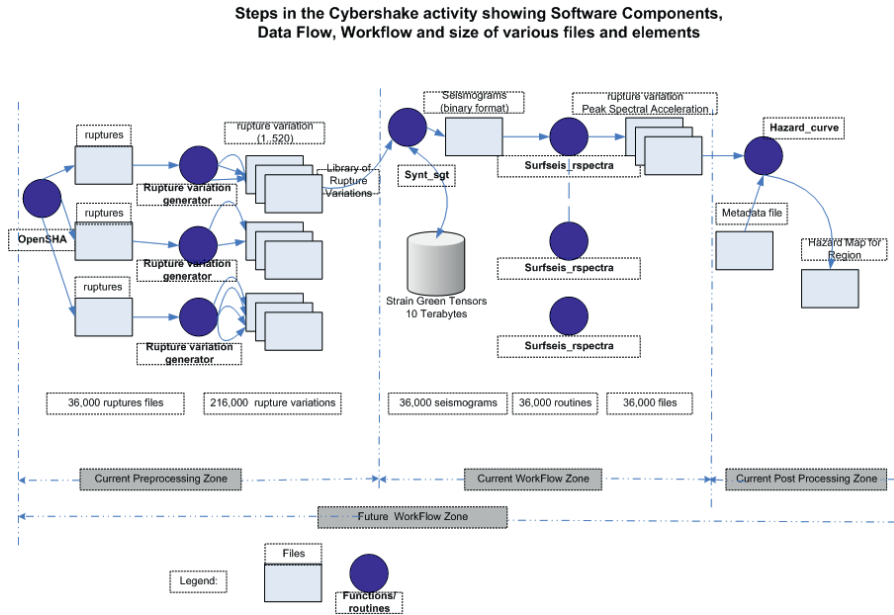
Figure 10.5: The CyberShake computational pathway is an end-to-end computation of a CyberShake Hazard Curve.

In this stage in the CyberShake processing, an OpenSHA implementation of an ERF is used. The ERF is the first computational step in our scientific workflow. The OpenSHA ERF is a GUI-based Java program that requires user interactions during execution. The operator uses a series of drop-down menus and text boxes to enter information about the site, such as the location, the cutoff distance, and other configurable parameters. Then the ERF program is run once to create the list of ruptures that might affect the site being considered. We did not want to integrate a GUI-based program into the workflow, and thus we excluded this processing step from the CyberShake workflow.

CyberShake Step 3: Calculate Rupture Variations

Rupture descriptions produced by current ERFs are static descriptions of earthquakes that indicate the fault surface and the magnitude of each earthquake. However, the earthquake wave-propagation simulations used by CyberShake require more detailed information about the ruptures in order to produce realistic seismograms. Also, several variations of each earthquake rupture must be considered. As a general rule, the larger the earthquake in the ERF, the larger the number of rupture variations that will be used in the CyberShake calculation. For each earthquake in the ERF, the CyberShake

system will calculate a series of rupture variations using a heuristic-based method developed by SCEC scientists.

Table 10.3 shows an example of how the CyberShake processing expands the original ERF rupture list into a larger list of rupture variations (for the USC site). The differences between rupture variations include hypocentral locations and slip distributions. Larger earthquakes require more variations because there are presumably more possible hypocentral locations and slip distributions that must be considered.

Table 10.3 shows that the ERF rupture list for this site contains a large number of small earthquakes, which result in many variations. The table also shows that the ERF rupture list contains only a small number of very large earthquakes for this site. However, for each of the very large earthquakes, CyberShake produces a large number of variations. The result is that the CyberShake processing must produce seismograms for over 100,000 ruptures. Other sites have a similar distribution of ruptures by magnitude, so each CyberShake hazard curve calculation must simulate over 100,000 ruptures.

Site USC	Ruptures By Magnitude	Rupture Variations By Magnitude
Magnitude < 5.0	0	0
Magnitude ≥ 5 and < 6.0	20,450	64,320
Magnitude ≥ 6 and < 7.0	2524	14,600
Magnitude ≥ 7.0 and < 8.0	1435	47,066
Magnitude ≥ 8	12	12,864
Totals	**24,421**	**109,806**

Table 10.3: Ruptures and Rupture Variations for the USC site.

In order to produce all the rupture variations needed by CyberShake, we run a serial FORTRAN program called a rupture generator. This program is run only once to create a master list of all possible ruptures in Southern California. Since this program is run only once, and we do not need to run it for each site, we decided to exclude it from our workflow.

CyberShake Step 4: Calculate Strain Green's Tensors

The next step in the CyberShake computational pathway is to calculate strain Green's tensors for the site of interest. A strain tensor quantifies the strain of an object (e.g., the Earth) undergoing a 3D deformation (e.g. the deformation caused by an earthquake). For small deformations, the strain tensor can be described by a strain Green's tensor (SGT).

This part of the CyberShake computational pathway uses three different programs: a regular mesh maker, a velocity mesh maker, and the SGT calculation. These three programs are run to create a large SGT data set. SGT

calculations are the high-performance, MPI-based computing aspect of the CyberShake simulation. Before the SGT code can be run, an input velocity mesh must be generated. This is done in two steps. First, a regular mesh with the appropriate dimensions and grid spacing is created. Then a 3D velocity model program is run to assign properties such as P-wave velocity, S-wave velocity, density, and attenuation to each mesh point. The velocity mesh properties vary by location in the mesh. For example, the P-wave velocity, S-wave velocity, and density values all typically increase with depth.

The current SGT computation is a fourth-order, finite-difference code. One SGT calculation is run for each horizontal component of motion. Thus, two SGT simulations are run per site. The SGT calculations used in the CyberShake simulations require approximately 140 GB of RAM at runtime. On our target clusters, we can utilize approximately 500 MB of RAM per processor. In order to run the SGT successfully, we must divide the 140 GB across approximately 280 processors, or about 140 nodes on dual-processor systems such as the TeraGrid IA-64 or USC HPCC clusters.

Scheduling large MPI-based programs onto a cluster often has interactive aspects that are not easily managed by a workflow system. For example, the CPU-hours allocation available to the workflow should be verified prior to running. Sufficient disk space must be available in the output storage location. In some cases, a specialized queue, or a reservation for a set of computation nodes, is used, in which case the job should be run in a specific queue or at a specific time. Although it is possible to include these constraints into the workflow system, we decided to leave the MPI-based calculations out of the workflow for now since they are run only twice per site. We do plan to make them a part of the abstract workflow in the near future.

The SCEC workflow system has the capability to automatically restart jobs that fail. However, we recognized that special care must be taken when restarting large, multiday, 280-processor jobs. One way to address the restart capability is to model the SGT calculation as a series of smaller steps with checkpoint files. Then a failure would get restarted from the last checkpoint rather than from the beginning. However, to accomplish this we needed to elaborate our definition of the workflow to identify a series of restart-able calculations. This added complexity into our workflow that, in our judgment, did not add sufficient value.

CyberShake Step 5: Synthesize Synthetic Seismograms

The CyberShake reciprocity-based seismogram synthesis processing stage generates thousands or hundreds of thousands of seismograms for a site. To do this, we must run the seismogram synthesis code for each rupture, which amounts to tens of thousands of times. The seismogram synthesis program will generate output files containing the synthetic seismograms. Metadata must be maintained for each output file so that we can associate the seismogram with the ruptures that it represents.

This stage in the workflow must be executed once for every rupture. Seismograms for all the rupture variations are calculated during the same invocation. For ruptures that have a large number of variations (in some cases a rupture may have more than 1000 rupture variations), the runtime for this program can be many hours. In other cases, where there are few variations, the runtime can be minutes. This stage was included in the workflow.

CyberShake Step 6: Calculate Peak Intensity Measure

Once one or more seismograms have been calculated, the next step is to extract a peak ground-motion value from the synthetic seismograms. SCEC scientists have decided that spectral acceleration at 3.0 seconds (SA3.0) is a ground motion intensity measure type that is consistent with the frequency content of the synthetic seismograms generated by the CyberShake workflow. To calculate peak SA3.0 values from our synthetic seismograms, we use codes that can filter the seismograms, differentiate the acceleration, and then calculate peak SA3.0.

The seismogram synthesis stage produces a binary seismogram file that includes all the seismograms for a given rupture, including both hozizontal components for each rupture variation. Thus, the peak SA3.0 calculation program must be executed once for every rupture in the workflow. Our SA3.0 calculation program is invoked once for each rupture and processes all components, and all rupture variations, in the specified file. This stage was also included in the workflow.

CyberShake Step 7: Calculate Hazard Curve

When all the peak SA3.0 values have been calculated, the final step is to calculate a hazard curve. To do this, the peak SA3.0 values for each rupture are read and a geometric average of the horizontal components is calculated. Then, the peak SA3.0 values are associated with the probability of the given rupture. These calculations are done for each rupture, the results are combined, and a hazard curve is calculated.

We excluded this final step from our workflow primarily because it uses a GUI-based OpenSHA program. However, the final processing step raised another important issue, which we refer to as delayed execution of programs. Based on the time required to execute all the jobs that lead up to this last summary stage, the execution time for this final job could be days, or even weeks, after the workflow is submitted. When a job in a workflow has an expected execution time that is days or weeks in the future, there is a reasonable possibility that the computing environment will change between now and then. Currently many of the Grid systems we target in our work do not provide an easy way of programmatically accessing up-to-date system information and thus make it impossible for workflow management systems to make good scheduling decisions over time. This leads to lower reliability that the workflow will execute to successful completion.

By the end of the CyberShake abstract workflow generation process, the workflow consisted of only two steps: a seismogram synthesis program and a peak spectral acceleration program. These two steps are shown in Figure 10.6. In our workflows, we had approximately 25,000 processing nodes for each step in the data flow, and this was before any needed data transfer jobs were added to support the Grid-based execution. The corresponding executable workflows generated by Pegasus contained tens of thousands of tasks.
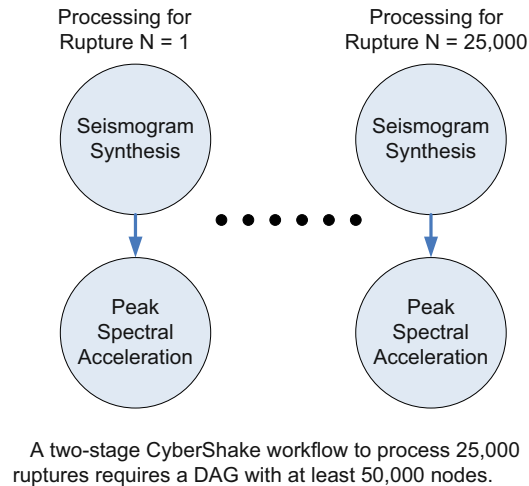

A two-stage CyberShake workflow to process 25,000 ruptures requires a DAG with at least 50,000 nodes.

Figure 10.6: The CyberShake abstract workflow has two processing steps. These two processing steps must be repeated approximately 25,000 times each to produce a single PSHA hazard curve.

10.9 Resource Provisioning in the CyberShake Workflows

Once the CyberShake abstract workflows were developed and we prepared to execute them, a new issue emerged that was related to resource provisioning. In the context of CyberShake workflows, provisioning means reserving computer resources for our use during the workflow execution. We used the Condor glide-in [96] capabilities to provision the computing resources and to overcome the fact that our workflow requirements do not match the scheduling policies of the underlying computing resources.

To understand this issue, we must examine the requirements of the workflow and characteristics of the execution environment. Once our abstract work-

flow was converted to an executable workflow and all the data movement jobs, directory creation jobs, and data registration jobs were added, the total number of jobs in the workflow exceeded 80,000. While some of these are long running, I/O-intensive programs, all of them are single-processor, sequential programs. The only available computing resources that will run this number of jobs within a reasonable amount of time are the high-performance clusters at the TeraGrid and USC HPCC. However, neither of these computational clusters are configured to run a large number of sequential jobs. Instead, they are configured to run a few large parallel jobs. Supercomputer sites implement this policy in a couple of ways.

First, the job submission managers at these sites will typically allow a single user to submit less than 100 jobs at a time. None of the supercomputer sites we use will allow us to send 100,000 jobs to their job submission queues all at one time. This issue can be addressed to some extent by the job submission throttling capabilities of Condor, but the number of jobs we need to schedule represents a real issue for the CyberShake workflows.

Second, the supercomputer facilities give preference to large parallel jobs through the job priorities used by the underlying job scheduling systems. The sites prefer to support the very large jobs that could only run on the large supercomputer clusters. Thus job scheduling algorithms are set up so that large, highly parallel, long running-jobs (that is, supercomputer class jobs) received scheduling priority.

SCEC researchers recognized that the CyberShake computations were supercomputer class computations even though they were not written as MPI-based jobs. Rather than rewrite all the CyberShake programs as parallel codes, our workflow system was able to work around these scheduling policy problems by using provisioning techniques offered by the Condor glide-in system. The Condor tools allow us to run *placeholder* jobs on one or many cluster computation nodes (in our work, we used 50 to 100 placeholders at any one time). Once the placeholder programs are running, we can send CyberShake jobs from our Condor-G job submission host directly to the placeholders for execution. Once a CyberShake job completes on a compute node, Condor-G sends another job to the placeholder.

10.10 CyberShake Workflow Results

The analysis, software development, configuration, testing, and validation work that led up to the first full-scale CyberShake workflows was performed over approximately six months. The first two full-scale CyberShake workflows were run over a period of approximately one month. Subsequently, the computational rate increased dramatically. Eight additional CyberShake curves have now been calculated, at a rate of approximately one a week.

During our first two full-scale CyberShake workflow runs, we executed over 261,000 separate jobs at four different computing centers (SCEC, SDSC,

NCSA, and USC), and we used over 1.8 CPU-years of processing time. Over 80,000 separate files were created and registered into our data management system. We are still collecting statistics on the subsequent eight site calculations, but the numbers are expected to be similar.

The CyberShake workflows made good use of our Grid-based environment. SCEC computers were used as job submission hosts and as the storage location for the resulting seismograms, spectral acceleration, and hazard curve data files. The SCEC workflow system allowed us to create file replicas at two TeraGrid sites and then to divide our workflows across two different supercomputer facilities, with the results ending up back at SCEC. This flexible use of available computing resources underscores the value of specifying workflows in a resource-independent manner. It also underscores the capabilities that can be built on top of a Grid-based infrastructure.

10.11 Conclusions

The first ten CyberShake probabilistic seismic hazard curves are currently under analysis. The CyberShake results are so new that conclusions regarding the scientific benefits of using 3D waveform-based intensity-measure relationships in probabilistic seismic hazard analysis are still pending. Regardless of the final judgment on this new class of PSHA hazard curves, CyberShake represents an important research effort that has provided SCEC scientists with results needed to evaluate this widely anticipated new approach to PSHA.

Our scientific workflow tools provided scalability of calculation through automation. These tools allow us to work at a computational scale that would be very difficult to achieve otherwise. However, we recognize that the computational demands of SCEC science are increasing just as quickly as our computational capabilities.

In order to meet the computational requirements of SCEC science in the near future, we need to improve our workflow automation. We plan to begin by increasing the number of programs executed as a part of the CyberShake workflow. At this point, it appears that the portions of our computational pathway that benefit from modeling as a workflow share two characteristics: *high repetitiveness* and *low interactivity*. We believe that these characteristics may be used to identify which parts of a series of scientific calculations can be most readily expressed as a scientific workflow regardless of the underlying workflow technology.

We believe that scientific workflow tools provide the current best technology for working at the computational scale needed to perform SCEC's transformative seismic hazard analysis research. If SCEC research goals are required only one or two hazard curves, it may have been faster to calculate them without the use of a workflow system. However, since SCEC researchers wanted to calculate hundreds or thousands of these hazard curves, we needed a system that would allow us to scale up the large CyberShake computational

pathway calculation by one or two orders of magnitude. We believe that as SCEC workflow tools evolve and improve, they will make this level of scientific processing and data management possible.

Acknowledgments

Part II

Workflow Representation and Common Structure

Control- Versus Data-Driven Workflows

Matthew Shields

11.1 Introduction

Workflow is typically defined as a sequence of operations or tasks needed to manage a business process or computational activity (Chapter 1). The representation of the sequence of operations or tasks is handled in many different ways by different people and varies from simple scripting languages, through graphs represented in textual or graphical form, to mathematical representations such as Petri Nets (Chapter 13) or π-calculus (Chapter 15).

Most groups agree that there are two simple classes of workflow structure into which most of the representations of workflow languages used in this book fall: control and data flows. The two classes are similar in that they specify the interaction between individual activities within the group that comprise the workflow, but they differ in their methods of implementing that interaction.

In control-driven workflows, or control flows, the connections between the activities in a workflow represent a transfer of control from the preceding task to the one that follows. This includes control structures such as sequences, conditionals, and iterations. Data-driven workflows, or data flows, are designed mostly to support data-driven applications. The dependencies represent the flow of data between workflow activities from data producer to data consumer.

There is also a smaller set of hybrid workflow representations based on a combination of control and data flows. These hybrids use both types of dependencies as appropriate but are normally biased toward either data flow or control flow, using the other to better handle certain conditions. For instance, in a data-flow system such as Triana (Chapter 20), there are situations where a downstream task needs to be activated but the upstream task produces no output. In this case, a *trigger* is used to switch the flow of control. In hybrid control-flow systems, such as the CoG Kit's Karajan workflow (Chapter 21), data dependencies can be represented by a *future*, the concept of data that has not yet been produced, which can block the control flow with a data flow.

This chapter will examine the differences, and indeed similarities, between control flow, data flow, and hybrid representations, with examples of each type

and the applications and frameworks that use them. We will start with a discussion of different workflow representations and some common concepts and conclude with some of the pitfalls and some possible solutions to the problems associated with heterogeneous workflow languages in Grid environments.

11.2 Workflow Representations

The data-driven versus control-driven workflow argument has run for as long as workflow techniques have been in use and can be as evangelical as the choice of editor, *Vi* or *Emacs*, or programming language, C++ or Java. Both sides are convinced that the structure they use is the correct one, but there are cases for the use of both workflow representations, and as we edge toward interoperability and a common workflow language, mixed usage. The choice of which is used in any given framework usually comes down to the original application domain that drove the framework development, as we will see when we examine some examples.

11.2.1 Common Workflow Terminology

It is worth mentioning here some of the common workflow terminology that gets used within the various representations and frameworks. Workflow by its definition has a number of common concepts; however, these are often known by different names.

By definition, a workflow is a sequence of *operations* or *tasks* needed to manage a computational activity. These are typically represented graphically as a *node* on a graph or in a script as a *process* or a *job*. In Chapter 12, the author describes component and service-based workflows, so we also have the terms *components* and *services* used as a name for the computational processes in the workflow. Different workflow frameworks also have different names for this concept: in Kepler (Chapter 7) they are called *actors*; in Petri Net theory (Chapter 13), *transitions*; in Virtual Data Language (VDL) (Chapter 17), *procedures*; in Cactus (Chapter 25), *thorns*; in Askalon (Chapter 27), *activities*; in the CoG Kit's Karajan (Chapter 21), *elements*; and in Triana (Chapter 20), *units*. Although all of these terms hide different mechanisms and technologies, the basic concept of an operation or task holds, and we can think of each of these as a "black box" process that performs some computation or other operation.

The connections between operations are also known by different names: *vertices* in a graph, *edges* in Petri Nets, *pipes* in data-flow systems, and *messages* in service-based systems. They all, however, represent an order to the execution of the operations in the workflow. This order may be a *data dependency*, where the product of the first operation in a connection must be available for the execution of the second operation to start, or a *control dependency*, where the flow of execution passes from the first operation to the

second in the connection, or in a more complex case control is passed from one operation to another based upon a control-flow structure such as *if...then* or *while.*

11.2.2 Classifying Workflow Types

A useful way of classifying whether a representation is control flow, data flow, or some hybrid of both is to look at the connections or dependencies between any two given operations or activities in the workflow. If the connection is a data dependency, such as a data file that must be complete and in place before a succeeding activity can execute, or a socket-based data pipeline, then the workflow is data driven and probably data flow. If the dependency is one of timing, such as task a must complete before b can start, then the workflow is more than likely a control flow.

Another way of looking at the difference between control flow and data flow is to examine the main artifact with which each representation concerns itself, or the terms in which the main concept of the representation is defined. In a control-driven workflow system, the main artifact is a process. Control flow concerns itself mainly with the execution and coordination of processes. The workflow representation will be defined in terms of those processes (i.e., execute process a then execute process b). In a data-driven workflow system, it is units of data and data products that become the main artifacts; the processes or activities in the workflow are merely data transformations of some description. Thus the workflow representation will be centered around the data products (i.e., transform input a into product b).

11.2.3 External Workflow Representations

Most workflow tools and frameworks have two forms of representation an internal one that is used to manipulate the workflow inside an editor or execution engine, for instance, and an external one used for storing workflows and communicating them between participants in the workflow "generation to execution" life cycle.

External representations for workflow instances, whether they are based on control flow or data flow, are often very similar, and in a large number of cases these external forms of representation are stored as XML documents. One of the most common forms is that of a directed acyclic or cyclic graph (DAG or DCG) with nodes and vertices. Whether nodes represent processes and activities or data and data products depends largely on the type of workflow and also the problem domain and methodology.

Petri Nets are another popular representation medium for workflow and can model workflow by representing data as *tokens* and processes as *transitions.* Other representations include scripting languages that model the relationships between tasks as a series of ordered function calls and Unified

Modeling Language (UML) diagrams that use the standard diagrams and representations to model the relationships.

In the Triana workflow language (Chapter 20), a predominantly data-flow language, the external representation is an XML-based DCG. The main artifacts are processes, so the nodes in the graph represent processes, and since we are dealing with a data-flow model, the vertices in this case represent data dependencies or transfers. The XML representation specifies the processes in a WSDL-like format and the vertices as a series of parent–child relationships.

SSDL-SC protocol (Chapter 15) expresses its workflow as a sequence of messages exchanged between participants in the workflow. The order of these messages, the participants in the exchange, and the direction in which the message travel define the workflow and hence the interaction between the services. The external representation as the framework's name, Soap Service Description Language suggests, takes the form of a series of SOAP messages, together with XML, that specify the participant services and the message interactions between them.

The Java CoG Kit's Karajan workflow language (Chapter 21) is an example of a hybrid control flow. The main artifacts are Grid processes and file transfers. The external representation is a parallel extensible scripting language with both XML and native representations. The script specifies the process and file transfers and the order in which they are executed. It includes support for parallel execution and control constructs such as looping.

Petri Nets are another popular representation medium for workflow. Grid Workflow Definition Language (GWorkflowDL) (Chapter 13) is based on Petri Net Markup Language (PNML), an XML dialect for representing Petri Nets. PNML can describe Petri Nets together with information about their graphical layout; GWorkflowDL provides extensions to relate transitions with real services or components and tokens with concrete data. Petri Nets can model both control and data flows since both data and process artifacts are represented with equal weight. Control constructs such as loops and conditionals are supported implicitly by the language, so the correct classification is probably as a hybrid control flow.

11.3 Control-Driven Workflows

In a typical control-driven workflow, the workflow or program consists of a sequence of operations. An operator reads inputs and writes outputs into common store such as a file system. In the simplest case, the operators run sequentially, with the control dependency in the workflow defining the successor once the predecessor has completed.

Control-driven workflow originated in the scripting community, where fine-grained small applications such as Unix processes can be chained together with some shell script or similar high-level language "glue" to form more complex programs. Each process is executed from the script in turn with

control passing from the script to the individual process and back to the script again upon completion. Movement of data is typically handled in this situation by a dedicated file or data transfer process that is called between two compute processes.

Control flow can simulate data flow with data transfers and monitors. In the CoG kit's Karajan, there is a concept called *futures* that allows a data dependency to be established for data that have not yet been produced by an operation. A *future* will cause the execution of certain parts of the workflow to be suspended until the data are available.

11.3.1 Control Structures

Most control-flow languages provide support not only for simple flows of control between components or services in the workflow but also for more complex control interactions such as loops and conditionals. Sometimes this support is implicit, as is the case with Petri Nets, and sometimes explicit, as in languages such as Karajan from the CoG kit.

It is obvious that users of workflow systems will often want more than the simple control constructs available to them. The ability to branch workflow based on conditions and loop over subsections of the workflow repeatedly is important for all but the simplest of applications. The argument is not whether these facilities should exist but how to represent them in the workflow language and to what degree the language should support them. For instance, is a single simple loop construct enough, or should the language support all loop types (i.e. *while*, *for...next*, *repeat...until*)? In the case of conditional behavior, the problem is determining whether the incoming value and the conditional value are equivalent. For simple cases where we are comparing integers or simple strings, checking the condition is straightforward and unambiguous. The problem comes when we have to compare complex, structured scientific data in scientific workflows. This type of data often needs domain-specific knowledge in order to perform comparisons. If the condition is coded in the language, then the implementation of the comparison must be coded in the execution engine. The result is that we end up with complex domain-specific information encoded in the framework itself, or we limit conditionals to simple comparisons.

To take this argument to its extreme conclusion, we could include support for all programming constructs and make the language Turing complete. However, at this stage we have to ask ourselves why we have written another high-level programming language rather than use an existing one and develop a graphical front end.

The whole ethos of workflows is power and simplicity. Workflow systems must be capable of performing all the functions a user requires; otherwise users just won't use them. But the same systems should be simple to use, hiding complexity where appropriate. I would argue that although control constructs are necessary, extending the workflow language itself to cover all possibilities

is against the principles of workflow. As we will see in the next section, there are alternatives.

11.4 Data-Driven Workflows

In a typical data-driven workflow, the workflow or program is a graph of operators with the vertices specifying data dependencies. An operator consumes data or tokens and produces data or tokens. All operators can run concurrently, with unfulfilled data dependencies causing an operator to block until its dependencies are completed. Data-driven workflows originated in applications where digital processing pipelines are common; for example, image processing and signal processing. These fields are inherently datacentric and often realtime, where processing pipelines are connected to measuring devices.

Most data-flow representations are very simple in nature, and unlike their control-flow counterparts, most contain nothing apart from component or service descriptions and the data dependencies between them; control constructs such as loops are generally not included. The SSDL workflow representation consists of services, with communication via messages. The dependencies between services are messages or patterns of messages, just another form of data, so this representation is a true data flow. While the SSDL-SC protocol does support "choices," in effect conditional branching, there is no loop construct.

In Triana's workflow language, there are no control constructs at all; the dependencies between tasks are data dependencies, ensuring the data producer has finished before the consumer may start. It has some control functionality in that a control dependency can be defined between two tasks where there is no data relationship; however, this is a simulated control, as the behavior is implemented as a control "message" passing from sender to receiver triggering, the transfer of control. Looping and conditional behavior is performed through the use of specific components; a branch component with two or more output connections will output data on different connections, depending upon some condition. Loops are handled by making a circular connection in the workflow and having a conditional component break the loop upon a finishing condition, outputting to continue normal workflow execution.

The benefit of both of these solutions to control behavior in data flows is that the language representations remain simple. The downside is that the potential for running the workflow on different systems is reduced since the other system must have access not only to the workflow but to the components or services that perform the control operations.

11.5 Toward a Common Workflow Language

A major goal for both the scientific and business workflow communities is common workflow languages, or at the very least a degree of interoperability

between workflow tools. Frameworks and tools need to be able to interoperate, and specifically for scientific workflows, the execution of a workflow within a Grid environment would benefit from being independent of the tool that created the workflow. Current proprietary solutions mean that it is not possible to share workflows across groups using different tools or execute on Grids where those tools are not installed.

The GGF Workflow Management Research Group [464] is examining various workflow languages with a view toward coming up with a common agreed standard. Any common workflow language will almost certainly have to include elements of both control flows and data flows and will probably start as a superset of the current main workflow languages used in the tools in this book. A mixed solution such as this, containing both data-flow and control constructs, would at least provide a metalanguage into which the other representations could be translated for sharing or execution and would begin the progression toward a common workflow language. The super setwould have to be pruned, as to include every extension or optimization, such as Karajan's, would make the language enormous.

As outlined in Section 11.3.1, adding every programming construct that might ever be needed to a workflow language representation will, eventually, turn what should be a relatively simple domain-specific language into a high-level general-purpose programming language. Designers of workflow languages should bear this in mind as they consider whether to add a new feature to their particular tool. There are alternatives such as designing components or services for performing given control tasks. If these are designed clearly, then they should be easy to replicate on other systems that want to execute the workflow. If the workflow is service-based, then common services that perform these tasks would make the execution even easier.

It is clear that both control- and data-flow techniques are needed for scientific workflow languages. Limiting the language to one or the other limits the usefulness of the tools built to use the language. It is also clear that constantly extending the language to include every programming construct will bloat the language and increase the complexity of the engines needed to execute it. Simple hybrid data-flow languages with limited control constructs will stand the best chance of being interoperable with the most tools and frameworks but still contain enough functionality to be able to represent real scientific workflows.

Component Architectures and Services: From Application Construction to Scientific Workflows

Dennis Gannon

12.1 Introduction

The idea of building computer applications by composing them out of reusable software components is a concept that emerged in the 1970s and 1980s as developers began to realize that the complexity of software was evolving so rapidly that a different approach was needed if actual software development was going to keep pace with the demands placed upon it.[1] This fact had already been realized by hardware designers. By the mid 1970s, it was standard practice to build digital systems by composing them from standard, well-tested integrated circuits that encapsulated sophisticated, powerful subsystems that we easily reused in thousands of applications. By the 1990s, even the designers of integrated circuits such as microprocessors were building them by composing them from standard cell libraries that provided components such as registers and floating-point units that could be arranged on the chip and easily integrated to form a full processor. Now, multiple processor cores can be assembled on a single chip as components of larger systems.

Unfortunately, the world of software has been much slower to adopt component techniques. There are many reasons for this. Part of the problem lies with the 1970s software design practices that dictated that every application was built by deciding upon a central family of data structures and then adapting algorithms to work on those data structures. This implied that code for the algorithms was intimately tied to the design of a few global application-specific data structures, and reuse was difficult. In some subdisciplines, the data structures were obvious and mature, well-tested libraries became the reusable components of software. The best example of this is numerical linear algebra, where there were obvious data structures (arrays) for matrices.

Object-oriented design made a substantial contribution to software reuse because it forced designers to think in terms of encapsulation and interfaces

---

[1] The first reference to the concept of software components is thought to have been by M. D. McIlroy in 1968 [290].

rather than algorithms that crawl exposed data structures. For example, in 1975, a programmer who needed to maintain a linked list would create the data structure and write the routines to insert and delete items. By 2000, the standard approach had evolved to using the generic list package available in your language of choice. Programming languages such as Java, C#, C++, and Python now have very large and impressive class libraries that provide an extensive set of "components" for the application designer. The richness of this library has enabled the programmer to accomplish much more with less work than at any time in the past. For example, building portable interactive graphics applications, a networked application that uses advanced security, or an application that is deeply integrated with a relational database would have required a substantial development and testing team twenty years ago. Today, an application that needs all three of these capabilities may only require a relatively modest effort by a single programmer.

It took a while for these modern libraries to achieve their current degree of success. Object-oriented design was originally thought to be the solution to the software reusability "problem," but it only got us part of the way. While the core OO concepts such as encapsulation, inheritance, and polymorphism are elegant and powerful, they do not guarantee that a class built for one application can be easily reused in another. To build truly reusable software, one must design the software as part of a component architecture that defines rules and contracts for deployment and reuse.

In the following sections of this chapter, we will explore several different definitions of software component architectures and how they have been used in scientific computing. We shall describe how this concept relates to the current model for Web services and how scientific workflow systems can be seen as an instance of software construction by component composition.

12.2 Component Architectures: General Concepts

The exact definitions of software component and software component architecture have not been formally established and agreed upon. However, a definition of a software component by Szyperski and Pfister is frequently cited and provides an excellent starting point:

> A *software component* is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. [402, 403]

By contractually specified interfaces, we mean that a component is an encapsulation of software functionality that is accessed by invoking an interface method or by sending a typed message. The precise interface language and type signature of the interface method or message schema is part of the contract. The other part of the contract is the behavior of the component when

it was invoked. For example, is the interface method invocation a procedure call that always returns the result of the component's action? Or does it return immediately with the component's response delivered by a subsequent callback? What are the rules for component failure? How does the component communicate exceptions?

By context dependencies, we refer to the conditions that must be satisfied by the host environment in order to operate properly. For example, does the component require a specific version of the Java Virtual Machine? Must the host operating system provide that certain libraries be installed for the component to operate?

A *software component architecture* is the *framework* of contracts and context dependencies that provide the ecosystem in which a family of software components may be composed into an application. This framework often takes the form of a runtime environment or application container that satisfies all the context dependencies for the target family of components. The individual components are deployable software modules that can be loaded into this framework and composed by a third party to build applications. A *component instance* is the specific realization of the component as a runtime entity in an application.

12.2.1 Composition and the Inversion of Control Principle

A critical feature of component frameworks is that applications can be built by composing components and, because the components are designed to follow a specific set of behavior rules, the composed application works as expected. For example, an important feature of component frameworks that differs from many standard programming models is the use of a design pattern called Inversion of Control (IOC) [275]. This idea is central to the way in which we think about component composition. In the simplest terms, think of two components implemented as Java classes. Call one class User and the other class Provider. Suppose each instance of the User class needs an instance of the Provider class to operate. The standard way to do this is

```
class User {
    Provider p;
    public void initializeUser(){
        p = new Provider();
        }
    }
```

The problem with this is that it makes the implementation of User completely dependent upon the implementation of Provider because we assume Provider is a class and that it has a null constructor. Inversion of Control states that the specific instance of the Provider should be instantiated elsewhere and "injected" into the state of the User. For example,

```
interface Provider;

Class User{
    Provider p;
    public void setProvider(Provider p){
         this.p = p;
         }
    }
```

allows a "third party" to create an instance of User and an instance of anything that satisfies the Provider interface and compose them with a call of the form

```
    User u;
    Provider p;
    ...
    u.setProvider(p);
```

In its purest form, IOC also implies that a component instance has a life cycle and environment that are completely managed by the framework. Everything the component needs is supplied by the framework. One aspect of this idea, as argued by Fowler [275], involves dependency injection, which is the concept that an application invokes a service but the instantiation of the component that implements this service is determined by the framework at runtime. In other words, the dependency of one component instance upon another is injected into the system at the latest possible moment.

Another type of behavior rule that many component systems enforce is a standard way for a framework to learn about a component at runtime. This type of component introspection is what allows a framework to discover that a component actually implements an interface required by an application.

The earliest component frameworks with many of these properties included Microsoft COM [61], the Java bean model. More recently, the complexity of the Enterprise Java Bean framework [298] has spawned other frameworks, such as Spring [392], to simplify its programming model. Pico [356] and the Apache Avalon [34], which is a server-side framework for Apache, are also important component frameworks based on some form of IOC.

12.2.2 Web Services as Software Components

If we consider Szyperski's definition of a software component, it is important to ask whether a Web service fits this definition. The standard definition of a Web service instance is as a network endpoint that accepts messages (and optionally returns results) in a manner specified by a Web Services Description Language (WSDL) document. The Web service in its abstract form (without a specific network endpoint binding) describes a software component. The form of context dependency is usually based on the selected WS profile that the Web service supports. For example, it is common to consider Web services that

support WSDL 2.0, SOAP 1.2, WS-Security, WS-Addressing, and WS-BPEL as a standard component framework. However, it should be noted that there are two models of SOAP interaction: request/response, which corresponds to a remote procedure call (RPC) style where an operation takes arguments and returns a result, and doc-literal messaging, where the Web service takes an XML document as input (and optionally) returns an XML document as a result.

12.3 Models of Composition

The relationship of software component models to the hardware systems that inspired them has also had a large impact on the way component frameworks allow users to compose components. There are two general models, each having multiple variations.

12.3.1 Direct Composition

If we think of a component literally the same way we think of an integrated circuit, we can envision it as having two basic types of interfaces: input ports and output ports. Data and requests for service flow into the input ports and results flow out of the output ports. A typical "graphical programming" environment will allow users to select components from a pallet and arrange them into component graphs where output ports of one component are connected to the input port of another. As illustrated in Figure 12.1, each component is represented by an icon with input and output ports clearly identified. Placing the icon for a component on the pallet represents an instance of the component. Dragging a mouse from an output to an input represents the IOC action to link these two instances together. The types of graphs that can be built using this approach are a function of the semantics of the component model.

In some systems, the graphs are limited to directed acyclic graphs (DAGS) or even to trees. In the most general case, the graphs can be cyclic, with components that have more than one input port and output ports that can be connected to more than one input. In this general case, the model of composition seems, at first, obvious; the output ports of one component can be connected to the input ports of another component as long as they have the same type signature. In this way, users can build an application as a full *data-flow* graph. This is an extremely attractive model for application scientists, and many of them would like to build applications using this concept.

Unfortunately, having an elegant picture of the graph of connectivity does not fully explain the semantics of the component interaction. There are two standard cases to consider:

1. Components that have *functional* or *method* interfaces
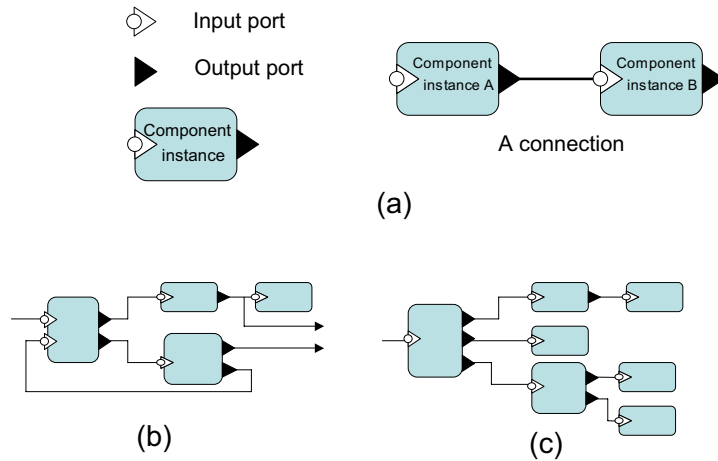2. Components that have interfaces based on sending and receiving one-way messages

Figure 12.1: The basic forms of direct composition. a) An icon representing a component with one input port and one output port, b) a general cyclic graph, and c) a tree as a special case of a directed acyclic graph.

The easiest of these to map to this graphical representation is the one-way message-based component model because the act of sending a message provides the data-driven component of the data-flow graphical metaphor. However, there are several other issues that must be addressed to make the picture semantically consistent. Let us assume that the framework uses an IOC pattern that allows the component to be designed so that it can be viewed as an idle process waiting for a message to be received at one or more of its input ports. The first issue that must be addressed is the meaning of two or more input ports. Do the semantics of the component allow it to respond to any input on either port? Or is there a *data-flow* rule that requires and input message on all ports prior to causing the action of the component to start? If it does follow this data-flow model, what happens if the component receives two inputs on one port but none on another? This implies that each input port must maintain a queue or have the ability to block upstream output ports from sending data until the current data at an input port have been consumed.

Unfortunately, having queues is not sufficient to make the full data-flow model work. A more difficult problem that every data-flow system must deal with is matching the correct inputs with each other. For example, suppose there are two input ports for a component. It is usually the case that if an input is received on port 1, the semantically matching input on port 2 is already there or will be the next to arrive. However, if there are many loops and possible branch conditions, it may be possible for the values that arrive at port 2 to arrive out of order. This requires a mechanism that uniquely labels messages according to the "iteration" with which they are associated. The easiest way out of this problem is to limit components to have a single input port or to eliminate cycles and restrict the composition to a DAG structure.

The next issue that must be resolved in such a component model is the meaning of *output* ports. In most of these systems, an output port is represented as a channel endpoint that the component writer can use to push out typed messages. Typically, the IOC pattern for the component model allows this channel to be connected to one or more input ports on other components. If more than one input port is connected to an output port, it is the job of the component framework to duplicate the message for delivery to each input.

There are several significant examples of this style of composition in a component framework. Ptolemy II [130] is a toolkit for the modeling and design of digital and analog systems and for the construction of embedded systems. The components in Ptolemy II are called actors and are composed together to form data-flow graphs that can support a variety of semantic behaviors. (The Kepler framework described in this book is built on top of Ptolemy.)

Mapping Composition Graphs to Components with
Functional Interfaces

Many component frameworks are designed with functional *procedure-call* interfaces, and users of these systems also demand some form of graphical or scripted composition tool in order to use them.

There are substantial semantic barriers to mapping a graphical compositional model onto software components that have procedure-call semantics for their external interfaces. The first of these involves the meaning of *input* and *output* ports. Suppose a component supports an interface of type F with a method of type signature $T\ m(S)$ (meaning it takes an argument of type S and returns a value of type T). Then it is natural to represent that interface as an *input* port, which expects to receive messages of the form *m(S-type-value)*. The problem is that this is not a one-way message because a value is returned to the caller. There is a standard solution for embedding such a component into a data-flow style of message-oriented system. We can automatically generate or compile a message-style proxy component, as illustrated in Figure 12.2, that gathers inputs, invokes the interface method, and converts the returned result to a message.
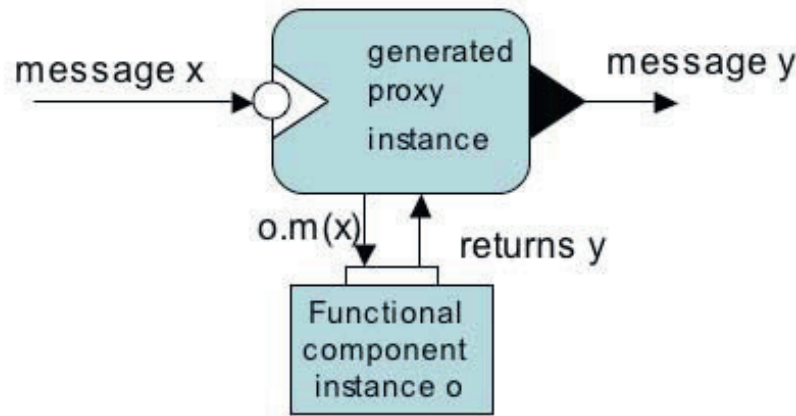
Figure 12.2: A proxy component can be automatically constructed from a procedure-call–based component that allows it to be inserted into a data-flow system.

This approach is used in Kepler, Triana, and Taverna (described in this volume) to integrate RPC-style Web services into a composition tool based on data flow concepts.

In the case where all the software components are Web services that understand WS-Addressing, there is another solution. WS-Addressing gives us the ability to pass a message to a Web service with a special tag in the header that says reply to another. This allows us to specify that the output of a component should be routed to a third party and a proxy need not be defined.

Graphs that are not data-flow oriented. There are other approaches for direct composition that are used rather than data flow. For example, Petri Nets provide a similar compositional model and a rich semantic platform for building a component architecture. Another approach is to base the composition on a realization of the Unified Modeling Language (UML).

The Common Object Request Broker Architecture (CORBA) Component Model [98] has components that have facets that correspond directly to input and output ports. In the domain of scientific computing, the CORBA Component model inspired the Common Component Architecture (CCA) [30, 84]. In the CCA model, each component communicates with other components by a system of ports. Ports are defined by a type system, which is expressed in Scientific Interface Definition Language (SIDL). SIDL provides a simple way

to describe a method interface in terms of the data types common in scientific computing. There are two types of CCA ports:

1. Provides ports are the services offered by the component. Each Provides port implements an interface defined in IDL.
2. Uses ports are component features that implement the use of a service provided by another component. They are bound to the stubs that a component uses to invoke the services of another port provided by another component. Uses ports are also defined by IDL.

A Uses port on one component can be connected to the Provides port of another component as long as they implement the same SIDL interface. The connection is made by the framework operation "connect" at runtime. When a component wants to use a service connected to one of its Uses ports, it executes a framework "getPort" operation. This provides an implementation of the port or blocks the invocation until one is available. When the component is finished, it issues a "releasePort" operation. This feature allows components to be replaced at runtime when not in use.

A Provides port is actually an interface of typed methods with typed results, and hence it is not a true data-flow model. However, CCA may be used in a way that emulates the single-input-port style data-flow by making all method calls have return type *void* and viewing an invocation of the Uses port as a push of data arguments to any connected Provides port. Several implementations of the basic CCA model exist, and they cover a wide spectrum of semantics. SciRun II [492] is one implementation of CCA that is designed for both distributed and concurrent execution and is used for high-end visualization. XCAT3 [245] is an implementation of CCA where the components have Provides ports that are implemented as Web services. Both SciRun II and XCAT3 support an actor style of data-flow graph similar to Ptolemy II, Kepler, and Triana.

But the standard model of CCA usage is not to emulate data flow. The types of graphs that typical CCA applications support are component control-flow call graphs, as illustrated in graph c of Figure 12.1. The emphasis in CCA is to provide a collection of language-neutral libraries of *SPMD parallel* components that can be composed at runtime and that can execute as efficiently as any parallel library. The standard CCA application has a root "driver" component. This driver component uses its Uses ports to invoke services provided by other components. The interaction is based entirely on a single thread of control. When a component invokes a method on a Uses port, control passes to the method implementing the interface on the Provides port of the component that is connected. This component may invoke methods on its own Uses ports, and the control is passed to the next connected component.

In most of the component systems we have described, concurrency is supported by the fact that one-way messages enable each component to run its own thread of control. When a component sends a message to another component, it may not need to stop and wait for a reply. The standard CCA model

exploits parallelism in a completely orthogonal manner. A Single Program
Multiple Data parallel program is one where the data for the computation
have been divided into some number, say N, of pieces. Rather than running
the program with the entire data set, one copy of the program with one piece
of the data is executed on each of N processors. Because most problems cannot
be easily divided into N pieces that can be solved independently, the program
has to be modified so that information that is part of one part of the solution
can be shared with other parts. This is done with a standard message-passing
library such as MPI.

A standard model CCA SPMD parallel component is one that runs in par-
allel on a distributed memory cluster computer. It uses MPI message passing
to share data needed to complete its work, but this message passing is not
visible from outside the component. As illustrated in Figure 12.3, the CCA
program using this model is a sequential composition of these parallel com-
ponents.
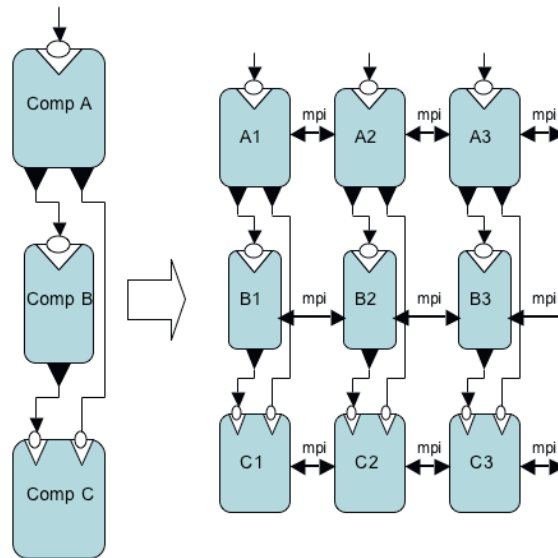
Figure 12.3: A standard model CCA program is a sequential composition of
parallel SPMD components. All of the message passing in the computation is
contained within the components. With a sufficiently powerful library of such
components, programmers can build applications with little need to write
explicit message-passing code.

These are not the only examples of component architectures for scientific
computing. Important early examples include the Model and CODE frame-
works [312]. In visualization applications, the most important early example

is the AVS system [266]. Webflow [51] was an early component model for distributed systems for scientific applications, and, more recently, the Discover project [50] considers the problem in the context of Grid systems.

12.3.2 Bus-Based Composition

Another model for component composition is based on a different metaphor from hardware design: Software components can be designed so that they can be "plugged into a message bus." The concept is simple and elegant. The message bus is supplied by the component framework and is responsible for delivering addressed messages to components. It does this by simply broadcasting each message to each component. The components listen to the message stream. The components that have a need to respond to messages of a particular type or address can take a copy of that message and respond by placing a new message on the bus. Many desktop graphical user interface systems work on this model. Sun's JXTA [64] is a good example of this model in the distributed system case.

There are several different ways such a bus-based system can be organized. One approach is to give each component a unique identifier that represents its *address* on the bus. A message that is tagged with that address is delivered to that component and no other. This approach makes it difficult for more than one component to receive a message unless a copy is created for each, but it does make it possible to build a family of components that are easily
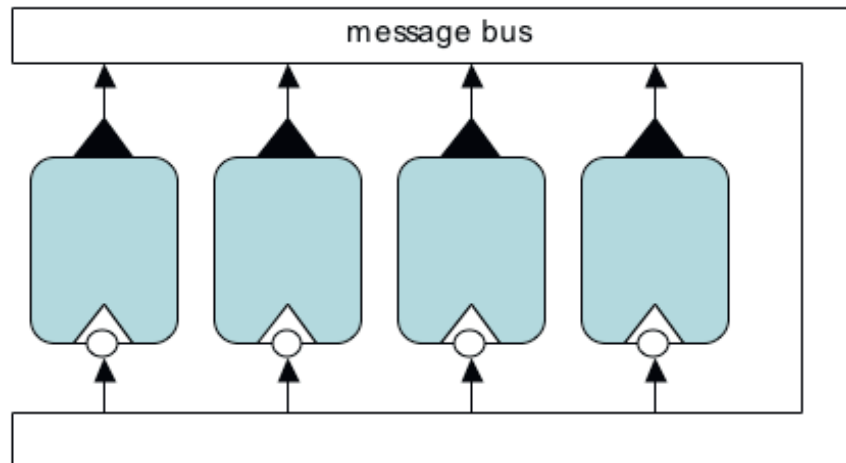
Figure 12.4: Bus-based composition configures components as agents which listen to messages broadcast over a message bus.

assembled into applications that can be easily scripted. For example, suppose you have three components *A*, *B*, and *C*. Each is capable of reading a data file, transforming it in some manner, and writing a new data file as a result. One invokes the component by sending it a message with the address of the input data file and the address of another component. When it completes, it posts a message on the bus addressed to the other component. The application can then be "programmed" with a fourth component *S*, which runs a simple script which uses two primitives: one to send a message on the message bus, *outputPort.send(address, return-address, message)*, and one to wait for a message to be delivered, *message = inputPort.Read()*. A Python-like script to couple the three components together in a linear chain would look like:

```
outputPort.send("A", "S", "input_file_1_url")
outputFromA = inputPort.Read()
outputPort.send("B", "S", "ouputFromA)
outputFromB = inputPort.Read()
outputPort.send("C", "S", outputFromB)
outputFromC = inputPort.Read()
```

Using the full power of a full scripting language, one can easily build component applications of arbitrary complexity.

Publish–Subscribe Composition

A more common approach to the design of message-based component systems is based on a *publish–subscribe* (pub–sub) notification system. In these systems, each message has a *topic*, which is often just a string, but it may also be a more structured object such as a path hierarchy. For example, a topic may be *userAction.mouseMove* or *userAction.ButtonPress*. Once connected to the framework bus, a component may then subscribe to events by topic or topics. For example, a component may subscribe to *timerEvents* to receive all events with this topic, or if a hierarchy is supported, a subscription to the topic *userAction.** would deliver all messages that begin with the *userAction* prefix.

Application construction in a publish–subscribe component framework is usually based on a more implicit form of control. One can add a component to an application because there is a need to respond to an event. For example, one component may occasionally publish an event signaling a special situation such as a resource going offline. Responding to that event may require a special action that must be taken by another component. In this example, this may be a component that alerts an operator and finds a substitute resource.

12.4 Stateful and Stateless Components

A frequently debated issue regarding software component systems involves when it is appropriate for a component instance to have state that is visible

and persists between client invocations. By state, we mean invoking the component instance may cause its behavior to change on subsequent invocations. For example, suppose a component has an internal variable *int x;* that can be accessed with a method *int incrementAndReturn( )* that does *x=x+1; return x;*. This has several problems. First, a client component may need to know the history of prior calls to this component in order for the returned result to have meaning. However, this is not always the case. If x is initialized to zero, then the value returned is some measure of the number of previous invocations, or it can be considered a "unique key" provided to the client. Is this a problem?

To answer this question, consider Web services as components. A service is a stateless entity because it provides an abstract capability defined by a document such as an abstract WSDL specification. A service instance is a concrete binding of a service to a specific network endpoint. A service may be provided by multiple service instances through a resolution mechanism that resolves, at request time, a service URI to one of the instances that implement the service. Therefore, if a client made multiple requests to a service, it would not know which instance it was talking to from one invocation to the next. Consequently, having state in the service instances would not be possible unless that state was somehow shared between multiple instances. In fact, this is a very common situation. For example, consider a service that provides the current temperature at the airport in Bloomington, Indiana. The current state of the temperature is held by an instrument that resides at the airport. This instrument is an example of a *resource* managed by the service. Multiple service instances can interrogate this resource and report the value as the current temperature at the time of invocation.

Another example is a service that is an interface to your bank account. The service may allow you to transfer funds between accounts or simply report the balance in an account. Clearly, we all hope that our bank maintains an accurate accounting of the state of our accounts. And we would insist that the Web service instances that access and update our account do so with the most reliable multiphase transaction protocols. We would not want the state of our transactions to persist in the service instance because that would expose them to fraud or loss. We never want a deposit to be lost because a service instance crashed! If a deposit failed, we would want the transaction to be aborted and the failure reported back to us. There are three important points here:

1. There is long-term state associated with the component, but it is not kept by the component instance. The state is held by the resource.
2. To access this state, we must provide context, such as an account number.
3. The component instance may require an internal state to complete a multiphase commit protocol with the back-end resource, but this state is not visible to the client and it does not persist between invocations.

In the Web services world, the concept of providing context to access the resource state is one that has received considerable attention. WS-Context and WS-Coordination provide protocols for establishing context for ordered

transactions between groups of services. WS-Resource Framework is a family of specs designed to provide a framework for modeling and accessing stateful resources using Web services. This includes mechanisms to describe views on the state, to support management of the state through properties associated with the Web service, and to describe how these mechanisms are extensible to groups of Web services.

12.5 Space and Time and the Limits to the Power of Graphical Expression

The component design metaphor of laying out icons representing software components onto a plane and connecting them together like electronic devices so that they may interact with each other in response to external controls is a powerful concept. As a metaphor, it is also very spatial in nature, and it allows us to see how complex systems can be decomposed into comprehensible units much better than trying to read through the linear source code that actually represents the reality of a large system.

It is often argued that this composition-in-space model is not appropriate for building large systems because a two-dimensional graph of a "real" application would be too hard to read. But component architectures are also naturally hierarchical. Most allow you to wrap up a network of components and encapsulate it in a new component. This allows systems of great complexity to be built from two-dimensional diagrams.

12.5.1 Workflow as Composition in Time

The concepts of a component architecture and workflow systems are obviously closely related. While software component methods are applied to the entire spectrum of software application development, the connection to scientific workflow is very clear. If we take the definition of workflow orchestration to be the activity of organizing the sequences of interdependent tasks that are needed to manage a business or a scientific or engineering process, we can see that this clearly relates to the composition of components in a component framework. Each task is a component, and the composition of one component with another in an output-to-input order is an acknowledgment of a temporal ordering that is based on some type of dependency. A workflow instance represents the active plan to sequence a specific set of tasks to accomplish a single goal. But the workflow template from which instance was derived from can be applied to an entire set of independent enactments that may run concurrently or in a pipelined or data-flow style.

Whereas the composition of software components into a single-program executable address space is the domain of many software component systems, workflow comes from the domain of automating human-scale processes that are scheduled over time. Another difference between direct connection style

component composition and most workflow systems is the way control is managed. Connecting software components into explicit data-flow graphs that are executed within a single system allows the control to be implicit and defined by the local exchanges between components. This type of distributed, asynchronous local control is a defining characteristic of a composition-in-space model. However, if a system is physically distributed and composed of a heterogeneous collection of elements that interoperate over the Internet, then completely distributing control is problematic because it is much more difficult to recover from faults. Hence workflow systems tend to be managed by a central workflow engine that executes a control script (which may have been compiled from a graphical representation). This central control script interprets the component composition graph. It initiates the interaction with each component and waits for its response. When the response is received, the control script can proceed with the next action as determined by the intercomponent dependencies that define the workflow. If the completion of one component interaction enables the invocation of more than one succeeding component, then the control script can invoke them concurrently (either by using a separate thread for each or by making nonblocking requests).

Having a centralized enactment engine that does all of the direct invocation of component services may seem inefficient compared with distributed control. But for most scientific workflows, which may run for very long periods of time, this inefficiency is small compared with the advantage of having a single point that can report on the status of the application and change the course of the workflow enactment if necessary.

12.5.2 Limits to the Power of Graphical Expression

Many of the scientific workflow tools described in this volume are based on providing users with a graphical composition tool. This is an extremely attractive paradigm for programming scientific workflows, and it always raises questions. How powerful is this concept? Is graphical composition of components all that is needed for programming? In computer science terms, we are asking whether these graphical programming systems are Turing complete. In general, the answer to this question is "no." There are many programming activities that are impossible with most graphical systems. For example, most graphical systems are unable to express exception-handling conditions. A more fundamental limitation of many systems is the lack of facilities to create new data types. The fact is that components and services are encapsulation mechanisms and what they encapsulate is either another workflow or component graph or conventional computer code.

A complete component programming system requires not only mechanisms to compose components but also ways to describe their interfaces and behaviors. This may be an Interface Definition Language of some type, or it may be an XML schema. For Web services, it is the Web Services Description Language (WSDL). But to be truly useful, a system must have a way to build

new components and a tool that can convert important legacy applications into components that can be effectively reused.

Petri Nets

Andreas Hoheisel and Martin Alt

In 1962, C.A. Petri introduced in his Ph.D. thesis [351] a formalism for describing distributed processes by extending state machines with a notion of concurrency. Due to the simple and intuitive, but at the same time formal and expressive, nature of his formalism, Petri Nets became an established tool for modelling and analyzing distributed processes in business as well as the IT sector. This chapter gives a brief introduction to the theory of Petri Nets and shows how Petri Nets can be applied for effective workflow management with regard to the choreography, orchestration, and enactment of e-Science applications. While choreography deals with the abstract modelling of applications, orchestration deals with the mapping onto concrete software components and the infrastructure. During the enactment of e-Science applications, runtime issues, such as synchronization, persistence, transaction safety, and fault management, are examined within the workflow formalism.

13.1 Introduction

E-Science applications are usually composed of several distributed services that are part of a specific process. The user or application developer has to decide which services should be used in the application and has to specify the data and control flow between them. We will use the term *workflow* to refer to the automation of both control flows and data flows of the application.

In order to simplify the composition of workflows, it is mandatory to describe an application workflow in a simple, intuitive way. This section gives a brief overview and classification of common approaches for describing workflows and compares these approaches with the notion of Petri Nets.

Existing workflow description languages can be grouped roughly into two classes: *Script-like* workflow descriptions specify the workflow by means of a textual "programming language" that often possesses complex semantics and an extensive syntax, while *graph-based* workflow description languages specify the workflow with only a few basic graph elements. Examples of script-based

workflow descriptions are GridAnt [446] and Karajan (refer to Chapter 21). These languages explicitly contain a set of specific workflow constructs, such as *sequence* or *while/do*, in order to build up the workflow. Purely graph-based workflow descriptions have been proposed (e.g., for Symphony [265] or Condor's DAGMan tool [97]) that are mostly based on directed acyclic graphs (DAGs). Compared with script-based descriptions, DAGs are easier to use and more intuitive for the unskilled user: Communications between different services are represented as arcs going from one service to another. However, as DAGs are acyclic, they offer only a limited expressiveness, so that it is often hard to describe complex workflows (e.g., loops cannot be expressed directly).

Another commonly used script-based approach to describe workflows is the *Business Process Execution Language* (*BPEL*) and its recent version for Web Services (BPEL4WS) that builds on IBM's WSFL (Web Services Flow Language) and Microsoft's XLANG (Web Services for Business Process Design). BPEL is described in more detail in Chapter 14. In comparison with Petri Nets, BPEL has two main disadvantages. First, BPEL possesses complex and rather informal semantics, which makes it more difficult to use formal analysis methods and to model workflows, especially for the unskilled end user. Second, it has a limited expressiveness (in the sense of suitability); i.e., it does not directly support some workflow patterns, such as arbitrary cycles [436].

13.1.1 Petri Nets

A Petri Net is one of several mathematical representations of discrete distributed systems. As a modelling language, it graphically depicts the structure of a distributed system as a directed bipartite graph with annotations. As such, a Petri Net has place nodes, transition nodes, and directed arcs connecting places with transitions [467]. If one abstracts from capacity constraints, Petri Nets are Turing complete.

There exist several different types of Petri Nets. A common classification is based on a survey by [47], who distinguishes between three levels of Petri Nets:

- **Level 1:** Petri Nets characterized by places that can represent Boolean values; i.e., a place is marked by at most one unstructured token. Examples of level 1 nets are Condition/Event (C/E) systems, Elementary Net (EN) systems, and State Machines (SMs).
- **Level 2:** Petri Nets characterized by places that can represent integer values; i.e., a place is marked by a number of unstructured tokens. Examples of level 2 nets are Place/Transition (P/T) Nets, ordinary Petri Nets (PNs), and Free Choice Nets.
- **Level 3:** Petri Nets characterized by places that can represent high-level values; i.e., a place is marked by a multiset of structured tokens. Examples of level 3 nets are Colored Petri Nets (CPNs) and High-Level Petri Nets (HLPNs).

In order to model workflows in e-Science, it is very useful to relate the Petri Net tokens with the real data that are passed from the previous to the following activity. The tokens of a level 1 or level 2 net are unstructured (not distinguishable), so they do not carry any information besides their existence and number. These nets are used to describe basic control and data flows but are not suitable to model the data themselves. The tokens of a level 3 net, however, can be used directly in order to store the exit status (control data) or to model the input and output data (real data) of the previous activity, which are then evaluated by a following activity or the condition of a transition. In the following, we will introduce the basic Place/Transition Net (level 1 net) and two commonly used extensions called Stochastic Petri Net and High-Level Petri Net (level 2 net).

13.1.2 Place/Transition Net (P/T Net)

The basic Petri Net—also known as a Place/Transition Net or P/T Net— consists of *places* (p, denoted by circles), *transitions* (t, represented by thick vertical lines or rectangles), and directed *edges* (arcs) connecting places and transitions or transitions with places, but not places and places or transitions and transitions. An edge from a place p to a transition t is called an *incoming edge* of t, and p is called an *input place*. Outgoing edges and *output places* are defined accordingly. Each place can hold a number of indistinguishable *tokens*. The maximum number of tokens on a place is denoted by its *capacity*. A distribution of tokens over the places of a net is called *marking*, which represents the current state of the workflow. A transition is *enabled* if there is a token present at each of its input places and if all output places have not reached their capacity. Enabled transitions can *fire*, consuming one token from each of the input places and producing a new token on each of the output places. Consecutive markings are obtained by firing transitions. In P/T Nets, each edge may be labeled with a *weight* that expresses how many tokens flow through them at each occurrence of the transitions involved .

It should be noted that the Petri Net state transformation is local in the sense that it involves only the places connected to a transition by input and/or output arcs. This is one of the key features of Petri Nets, which allows the easy description of distributed systems [279]. The execution of P/T Nets may be nondeterministic since multiple transitions can be enabled at the same time. If every transition in a Petri Net has exactly one input place and exactly one output place, the net is in effect a state machine.

Developers often use P/T Nets to model the dynamic behavior of complex systems. The places are related with certain Boolean state information (e.g., open, close, done, failed), and the state is regarded "true" if the corresponding place contains a token. In level 2 nets, the tokens themselves do not carry any additional information, so they model the *existence* of data or specific side effects. P/T Nets are a good choice if you want to model the data and control the flow of applications but not the data itself. In case the data flow explicitly

ready to deliver   *ready to consume*

produce   *deliver*   3   2   *remove*   *consume*
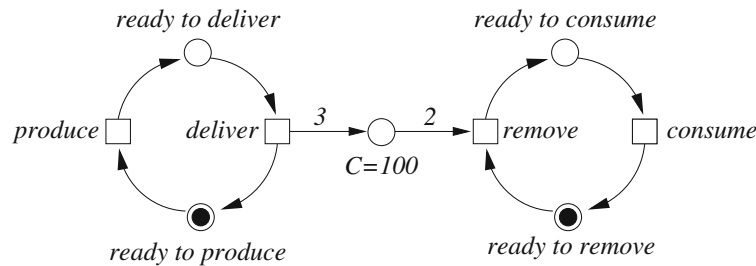
C=100

ready to produce   *ready to remove*

Figure 13.1: Place/Transition Net that models a producer/consumer system with unstructured tokens [369].

depends on the contents of the data, we recommend the use of High-Level Petri Nets instead (refer to Section 13.1.4).

Figure 13.1 shows a P/T Net representation of a producer/consumer system. The producer and consumer are connected via a buffer that has a capacity of 100 tokens. At each iteration, the producer puts three tokens into the buffer while the consumer removes two of them in a concurrent process.

13.1.3 Stochastic Petri Net (SPN)

Stochastic Petri Nets (SPNs) associate a *firing delay*, represented as a *random distribution function*, with each transition. Different types of transitions can be classified depending on their associated delay; for instance, immediate transitions (no delay), exponential transitions (the delay is an exponential distribution), and deterministic transitions (the delay is fixed).

Stochastic Petri Nets are mostly used to statistically analyze running systems (e.g., for probabilistic performance measures) and less to describe single workflows. SPN performance evaluation is the modelling of the given system using SPNs and generating the stochastic process that governs the system's behavior. This stochastic process is then further analyzed using known techniques such as Markov chain models and Semi-Markov chain models. In the context of e-Science frameworks, SPNs are used in complex workflow scheduling problems. Detailed insights into Stochastic Petri Nets can be found in [279].

13.1.4 High-Level Petri Net (HLPN)

One approach to using Petri Nets for the description of distributed workflows in the context of Grid computing is to relate the tokens of a level 3 net with classes and instances of real data by means of High-Level Petri Nets (HLPNs) [18]. HLPNs allow for nondeterministic and deterministic choice simply by connecting several transitions to the same input place and annotating edges with conditions. HLPNs also make the state of the program execution explicit

with tokens flowing through the net that represent the input and output data as well as side effects. In contrast, DAGs only have a single node type, and therefore data flowing through the net cannot be modelled easily. Using the concept of *edge expressions*, a particular service can be assigned to a transition, and *conditions*—also known as transition guards—may be used as an additional control flow. The resulting workflow description can be analyzed for certain properties such as conflicts, deadlocks, and liveness using standard algorithms for HLPNs. High-Level Petri Nets are Turing complete because they overcome the capacity constraints (unbounded places) and therefore can do anything we can define in terms of an algorithm [437].
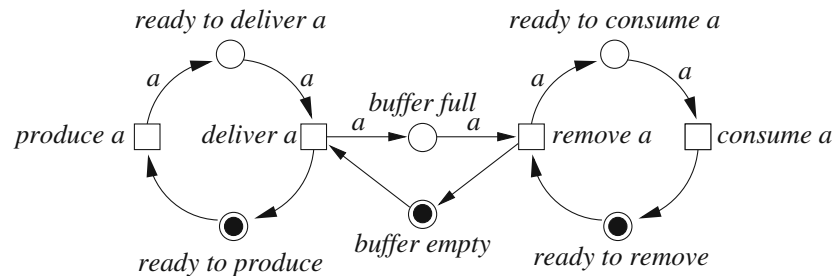
Figure 13.2: High-Level Petri Net that models a producer/consumer system for data objects of type *a* [369].

Figure 13.2 shows an example of an HLPN that models a producer/consumer system similar to that in Figure 13.1 but also capable of modelling data types. This Petri Net specifies the control flow using unstructured tokens (places "*ready to produce*," "*buffer empty*," and "*ready to remove*") and the data flow using tokens of type *a* (places "*ready to deliver a*," "*buffer full*," and "*ready to consume a*").

Sections 13.2, 13.3, and 13.4 are based on the concept of High-Level Petri Nets that is currently being used as the nucleus for workflow management in several projects, such as the K-Wf Grid project [420], the Fraunhofer Resource Grid [150, 193], and Instant-Grid [213].

13.2 Choreography—Using Petri Nets for Modelling Abstract Applications

Choreography—also known as dance composition—is the art of making structures in which movements occur, and it may also refer to the navigation or connection of these movement structures [465]. Translated to the world of distributed e-Science applications, the choreography models and describes the "movement" within applications on an abstract level. This section introduces

the basic theory of Petri Nets and describes how they can be used in order to assist the choreography of distributed e-Science applications.

A workflow description based on graphs does not necessarily mean that a graphical user interface is required in order to compose workflows. Petri Nets are in principle just mathematically well-defined entities that possess the nice feature of having an intuitive visual representation that the user could, but does not necessarily have to, use. In some cases, the user will actually never be confronted with the visual representation of the graph; e.g., when the abstract workflow description is composed automatically or if the Petri Net-based workflow description is the result of an automatic mapping from another workflow description format (e.g., performed by the BPEL2PN tool [191]).

13.2.1 Basics

In this chapter, we focus on High-Level Petri Nets (HLPNs), which were introduced informally in Section 13.1.4. For a formal definition of HLPNs, please refer to [370] or [222]. To model the workflow of a distributed application that consists of a certain number of coupled software components or services is fairly simple:

- *Transitions* represent software components and services or embedded sub-Petri Nets.
- *Places* are placeholders for data tokens or control tokens.
- *Tokens* symbolize real data or control tokens. Control tokens represent the state of the service and its side effects.
- *Edges* (arcs) define the data and control flow between the services and software components.
- *Edge expressions* specify the names of the service parameters. For example, within a service-oriented architecture (SOA) based on Web Services, edge expressions define a mapping between the input and output tokens and their corresponding SOAP message parts.
- *Conditions* (transition guards) define additional preconditions that must be fulfilled before the software component or the service is invoked. Normally, a condition is a function that maps input tokens onto a Boolean value. The transition fires only if all its conditions are "true." Conditions are also used to resolve conflicts and decisions in nondeterministic workflows (see below).

With these few language elements, the Petri Net concept is suitable for modelling the inputs outputs preconditions as well as the side effects for each software component or service, as shown in Figure 13.3.

Figure 13.4 shows three simple examples of how to use Petri Nets for modelling applications. In the first example (Figure 13.4a), the transition represents a single service with two input parameters (x and y) and one output parameter ($result$). The transition possesses a condition that depends on the

input parameters x and y. The result of the service invocation will be placed on the output place.

The second example (Figure 13.4b) shows how to build an if/then/else construct: Each transition represents one branch of the decision. In the sense of the Petri Net theory, the two transitions are in conflict because they compete for the same token, as they share the same input place. This conflict is solved by introducing two disjunctive conditions (*condition* and !*condition*). If *condition* is true, then service $f(x)$ will be invoked; if *condition* is false, then service $g(x)$ will be invoked.

A loop is shown in the third example (Figure 13.4c). The upper place holds the token that represents the data to be passed from each iteration to the next iteration. The token on the lower place contains the number i that is incremented after each iteration $(i + 1)$. If the initial value of this token is $i = 0$, then the service $l(x)$ will be invoked N times.

Further information about how to express common workflow patterns using Petri Nets is available in [305] and [370].
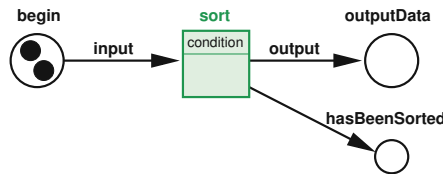
Figure 13.3: Example Petri Net that models the input, output, precondition, and effect (IOPE) of a *sort* transition.
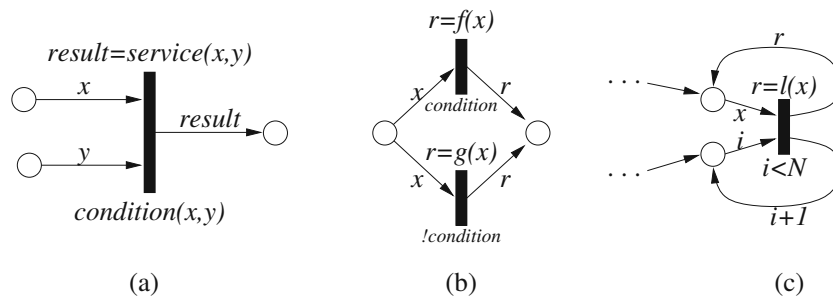
Figure 13.4: HLPNs for single services (a), if/then/else branches (b), and loops (c).

13.2.2 Case Study

In the following, we demonstrate the Petri Net approach using a real-world example from the e-Science community. The *Barnes–Hut (BH)* algorithm [39] is a widely used approach to compute force interactions of bodies (particles) based on their mass and position in space; e.g., in astrophysical simulations. At each timestep, the pairwise interactions of all bodies have to be calculated, which implies a computational complexity of $O(n^2)$ for n bodies. The BH algorithm reduces the complexity to $O(n \cdot \log n)$ by grouping distant particles: For a single particle in the BH algorithm, distant groups of particles are considered as a single object if the ratio between the spatial extent of the particle group and the distance to the group is smaller than a simulation-specific coefficient θ (chosen by the user).

For efficient access to the huge amount of possible groups in a simulation space with a large number of objects, the BH algorithm subdivides the 3D simulation space using a hierarchical *octree* with eight child cubes for each node (or *quadtree* for the 2D case). The tree's leaves contain single particles, and parental nodes represent the particle group of all child nodes and contain the group's center and aggregated mass. The force calculation of a single particle then is performed by a depth-first traversal of the tree. Figure 13.5 depicts an example partition and the resulting quadtree for the 2D case (see [39] for further details and complexity considerations).
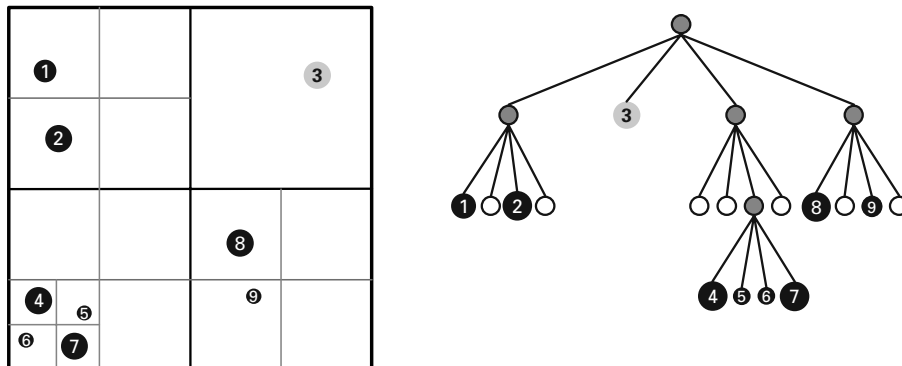
Figure 13.5: Example of a typical e-Science application: The Barnes–Hut algorithm and its octree partition of the simulation space.

We will now show how the workflow of this complex Grid application can be expressed easily as an HLPN.

The computations for one timestep of the algorithm are decomposed into a workflow containing six services, as shown in Fig. 13.6, which correspond to the following steps of the algorithm:

1. *Calculation of the spatial boundary of the simulation space.* In order to build the tree, it is necessary to know the boundaries of the universe to

be simulated. This is done using the service *compBB*, which produces a bounding box *bb* as output. Note that this bounding box is copied to two output places for use by two other services. Also, the array of particles *part* received as input is copied to a third output place, as it is also used by the next service.

2. *Indexing.* In order to group particles that are nearby in space, the particle array must be sorted so that nearby particles are also at nearby positions in the particle array. As a first step for sorting, an index is computed for each particle, based on its spatial location, using service *index*. The result *iPart* is a particle array, where each particle has an index associated with it.

3. *Sorting.* The particles are then sorted in ascending order of the index computed in the previous step using the service *sort*. The resulting sorted particle array *sPart* is used as input for two other services and thus copied to two different output places.

4. *Building the octrees.* This step builds the octree representation of the universe using the service *treebuild*. The resulting tree is used to group particles for efficient access.

5. *Force computation.* In this step, the interaction of each particle with all others is computed by the service *interact*. For each particle in *sPart*, the octree *tree* is traversed and the force effect of the current node is added to the velocity vector of the particle if the node represents a group that is small enough or far enough away. If this criterion is not yet met, then the eight child nodes are processed recursively.

6. *Particle update.* Finally, in the *update* service, for each particle, the current particle's position is updated according to the forces computed in the previous step.

Each of the services can be executed remotely on parallel high-performance Grid servers; e.g., as described in [17].

The workflow for a single timestep described above is executed iteratively to evolve the simulated universe for a user-defined amount of time. The corresponding workflow is shown in Fig. 13.7.
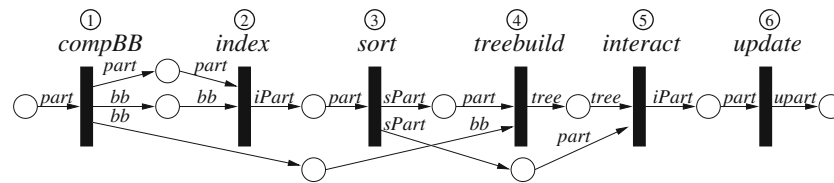
Figure 13.6: This Petri Net specifies the workflow for a single timestep of the Barnes–Hut algorithm.
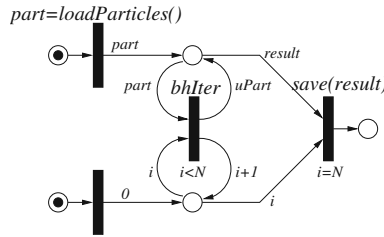
Figure 13.7: This Petri Net specifies the outer loop of the Barnes–Hut algorithm. The transition *bhIter* represents the embedded sub-Petri Net shown in Figure 13.6.

The single-iteration workflow is encapsulated in a composite transition *bhIter*, which is executed in a bounded loop. Before the algorithm starts, initial particle positions and velocities are loaded using the service *loadParticles*. Also, the iteration counter is initialized with 0 using a transition that places a 0 on its output place when executed. Finally the transition *save* is used to save the result after N timesteps.

13.2.3 Workflow Description Languages Based on Petri Nets

The concepts, definitions, and graphical notations of High-Level Petri Nets are standardized within the ISO/IEC 15909-1 standard [220]. Part 2 of this standard (ISO/IEC 15909-2) [221] is currently available as a working draft and specifies a so-called *Petri Net Markup Language* (PNML) [228] in order to establish an XML-based interchange format for exchanging Petri Nets between different Petri Net tools. The PNML is a language for describing a pure Petri Net together with its graphical layout; however, it is not possible with this language to relate transitions with services or software components, or tokens with concrete data, as is required for modelling and controlling real e-Science applications. Therefore, Fraunhofer FIRST developed a *Grid Job Definition Language* (GJobDL) that is based on PNML and possesses additional language elements required in a Grid computing environment. The GJobDL has been used extensively in the Fraunhofer Resource Grid [150] as a general Grid workflow description language since 2002.

Based on the GJobDL, Fraunhofer FIRST and the University of Münster recently developed the *Grid Workflow Description Language* (GWorkflowDL) in the context of the European project "Knowledge-Based Workflow System for Grid Applications" (K-Wf Grid) and the European Network of Excellence "CoreGRID." Besides the XML schema, there are also Java tools for creating, parsing, editing, and analyzing GWorkflowDL documents under development [421].

Figure 13.8 graphically represents the XML schema of the GWorkflowDL. The root element is called `<workflow>`: It contains the optional element
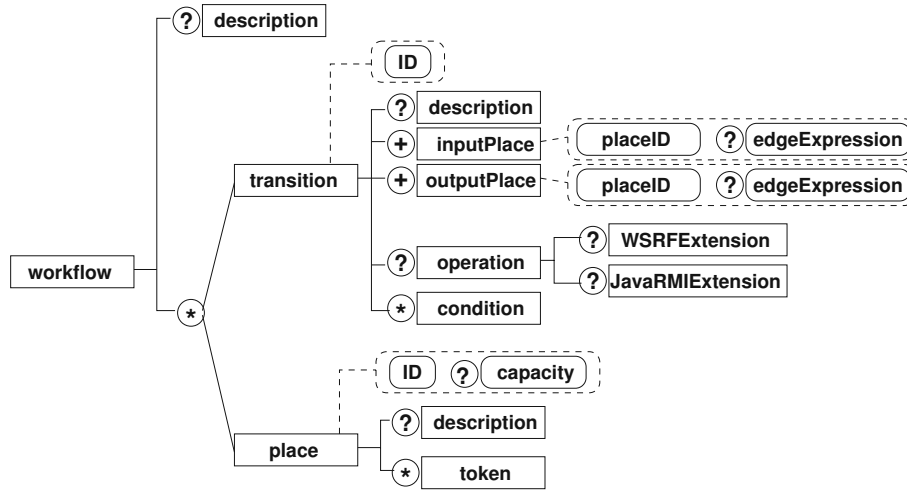
Figure 13.8: Graphical representation of the GWorkflowDL XML schema ("?"
= 0...1 elements, "∗" = 0...n elements, "+" = 1...n elements, rectangle with
dashed line = attribute, rectangle with solid line = element).

`<description>` with a human-readable description of the workflow and several occurrences of the elements `<transition>` and `<place>` that define the
Petri Net of the workflow. The element `<transition>` contains the child element `<operation>`, which may be extended by platform-specific child elements, such as `<WSRFExtension>` and `<JavaRMIExtension>`, representing special mappings of transitions onto particular Grid platforms. The elements
`<inputPlace>` and `<outputPlace>` define the edges of the net. Edge expressions are represented as attribute `edgeExpression` of `InputPlace` and
`OutputPlace` tags.

The XML document listed below specifies a workflow according to the
example represented graphically in Figure 13.3:

```
<workflow xsi:noNamespaceSchemaLocation=
      "http://www.gridworkflow.org/kwfgrid/src/xsd/
      gworkflowdl_0_9.xsd">
  <place ID="begin">
    <token>
      <soap>
        <data1 xsd:type="xsi:string">1 3 17 4 5</data1>
      </soap>
    </token>
    <token>
      <soap>
        <data2 xsd:type="xsi:string">5 13 4 5 100</data2>
```

```
      </soap>
    </token>
  </place>
  <place ID="outputData"/>
  <place ID="hasBeenSorted"/>
  <transition ID="sort">
    <description>sorts strings or numbers</description>
    <inputPlace placeID="begin" edgeExpression="input"/>
    <outputPlace placeID="outputData" edgeExpression="output"/>
    <outputPlace placeID="hasBeenSorted"/>
    <condition>string-length($input/token)&gt;0</condition>
    <operation>
      <WSClassOperation>
        <WSOperation owl="http://kwfgrid.net/services/Sort"
                     selected="true"/>
      </WSClassOperation>
    </operation>
  </transition>
</workflow>
```

13.3 Orchestration—Using Petri Nets for Mapping Abstract Workflows onto Concrete Resources

In the noncomputational world, the term "orchestration" stands for deciding which instruments should play which notes in a piece of music. Orchestration includes, in addition to instrumentation, the handling of groups of instruments and their balance and interaction [466]. If you now replace *instrument* by *resource*, *play note* by *invoke operation*, and *piece of music* by *e-Science application*, then you get a nice definition of the term orchestration in the context of e-Science. This section shows how Petri Nets can be used when mapping abstract workflows onto concrete resources.

Figure 13.9 shows an example of such a mapping. Each workflow may possess a different abstraction level, ranging from an abstract user request to the concrete workflow, which can be invoked directly on the available resources. All these abstraction levels are represented by Petri Nets within a single workflow description language. The mapping itself is done by refining the Petri Net (e.g., replacing a transition by a sub-Petri Net). The example in Figure 13.9 depicts the abstraction levels that are supported within the service-oriented architecture (SOA) of the K-Wf Grid project, as in the following list.

- *Abstract operation.* The user request represents a single abstract operation that still has not been mapped onto potential workflows. The output places of the transition are linked to some metadata, which specify the workflow result (data and side effects) requested by the user.

- *Web Service classes.* The user request is mapped onto an abstract workflow, which consists of operations of Web Service classes. This abstract workflow is independent from the concrete resources and represents the functionality of the workflow. The automation of the composition of abstract workflows is an ongoing research topic. Gubala et al. [183] used matching of ontologically described resources in order to find classes of Web Services that provide the desired output data and side effects.
- *Web Service candidates.* Each Web Service class is mapped onto matching Web Service candidates that are currently available within the distributed e-Science environment. An expert system could assist this matching process [125].
- *Web Service operations.* From each list of matching Web Service candidates, one concrete instance of Web Service operation has to be selected and invoked. This is normally delegated to a scheduler, which optimizes the selection of concrete Web Service instances according to a user-defined policy, such as "fastest" or "cheapest." In state-of-the-art e-Science environments, the scheduling decision is based on performance prediction and detailed monitoring data, such as computational load and network traffic.

It is worth mentioning that one workflow may possess different abstraction levels at the same time— for example, if the refinement of the workflow is done during runtime. In this case, only currently enabled transitions are mapped onto concrete Grid resources, while the rest of the workflow remains abstract. This is mandatory if a consecutive refinement decision depends on an intermediate workflow result. In this case, it is not possible to build the concrete workflow from the beginning—the orchestration is then an iterative or even interactive process.

Figure 13.10 shows how the case study in Section 13.2.2 can be mapped onto a service-oriented architecture (SOA). Each specific service of the Barnes–Hut algorithm is deployed on a dedicated host, and it is up to the workflow enactment service to synchronize the invocation of the remote Web Service operations and to transfer the data from one service to the next, as described in the next section.

13.4 Enactment—Using Petri Nets for Executing and Controlling e-Science Applications

Petri Nets are used not only for modelling coupled and distributed applications but also for executing the workflow directly on underlying middleware. In order to enact a workflow due to its description, a service is required that parses the abstract workflow description, maps it onto real resources (refer to Section 13.3), and coordinates the execution of the corresponding activities. The Workflow Management Coalition [479] uses the term *workflow engine* for such a software service that provides the runtime execution environment for interpreting workflows.
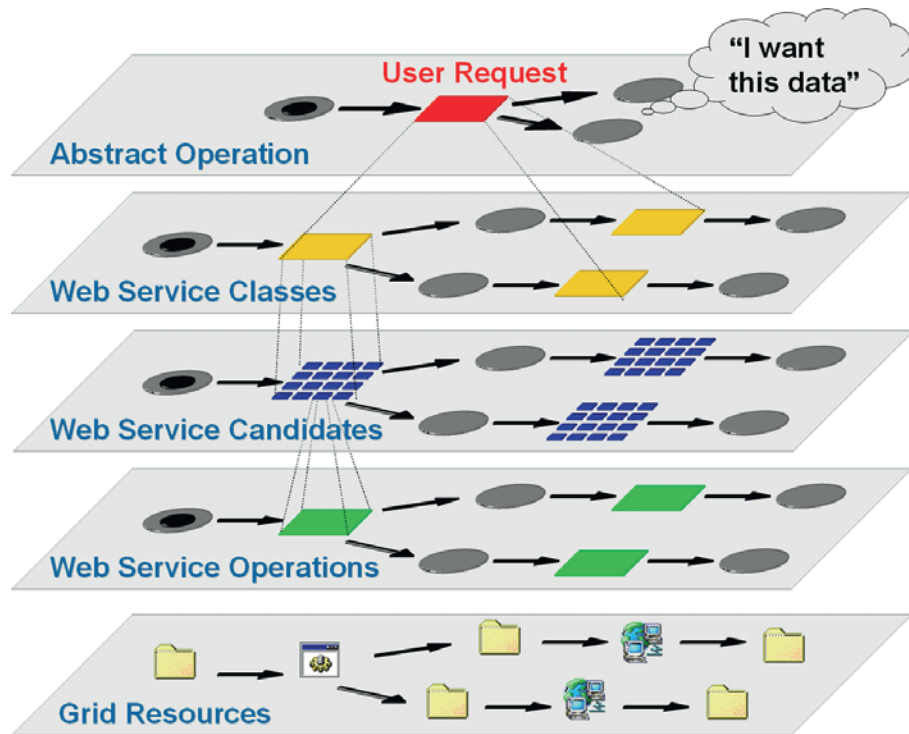
Figure 13.9: Workflow abstraction levels as supported by the K-Wf Grid project. All abstraction levels are represented as Petri Nets within a single workflow description language.

The development of a workflow engine based on Petri Nets is quite easy, as it is a straightforward implementation of the Petri Net rules. Figure 13.11 shows the kernel process of the *Grid Workflow Execution Service* (GWES) developed within the K-Wf Grid project. First, the workflow engine parses, verifies, and analyzes the incoming workflow description. Next, the engine collects all enabled transitions according to the mathematical definition of the term *enabled* (refer to Section 13.1.2). For each enabled transition, a condition checker evaluates the attached conditions (also known as transition guards). If the condition is true and if the transition references a concrete activity, then this activity is started (e.g., invoking a remote Web Service operation). If the activity completes, then the corresponding transition fires; i.e., one token is removed from each input place and the activity results (data, side effects) are placed as new tokens on the output places. If the transition refers to an abstract activity, then the transition has to be refined first as described in Section 13.3. The new marking of the Petri Net enables subsequent transitions
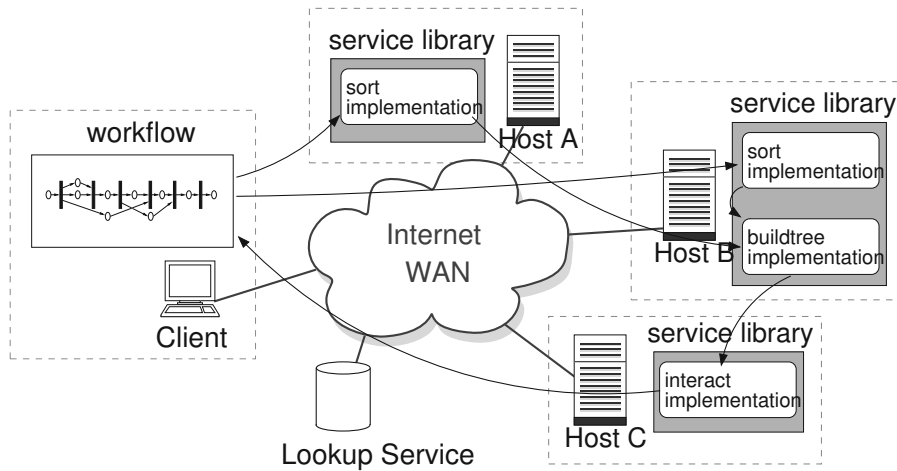
Figure 13.10: Example of the Barnes–Hut algorithm mapped onto a service-oriented architecture (SOA).

and their corresponding activities. If there are no more enabled transitions, nor active activities remaining in the workflow, then the workflow is completed.

A big advantage of Petri Net-based workflow engines is that they can process almost every workflow pattern without modification of the software. The Petri Net concept is very expressive and simple at the same time, and there is no need to implement special functionality for workflow constructs, such as loops, if/then clauses, and synchronization points. All these workflow constructs are supported implicitly by the Petri Net approach, and the workflow engine itself does not have to bother about them if it implements the basic Petri Net rules.

Figure 13.12 shows the Petri Net workflow enactment front end of the Fraunhofer Resource Grid. The right panel gives a graphical representation of the current workflow. The upper left panel depicts the XML document of the workflow description. The lower left panel shows the geographical distribution of the workflow on a map (here with four Fraunhofer Institutes involved in the workflow). The user interface can be used either as a stand-alone application or as a set of Java applets, which communicate with the workflow engine using Web Service technology.

After having introduced the basic execution mechanism of a Petri Net workflow engine, in the following we discuss further runtime issues, such as workflow persistence, transactional workflows, and fault management.

In a nonreliable environment, workflows should be *persistent*; i.e., they should be stored on nonvolatile storage during and after their execution. This is required, for example, to reproduce workflow results or to checkpoint the intermediate workflow state in order to recover a workflow after a system failure. Using the Petri Net approach, it is easy to achieve persistence, as the mark-
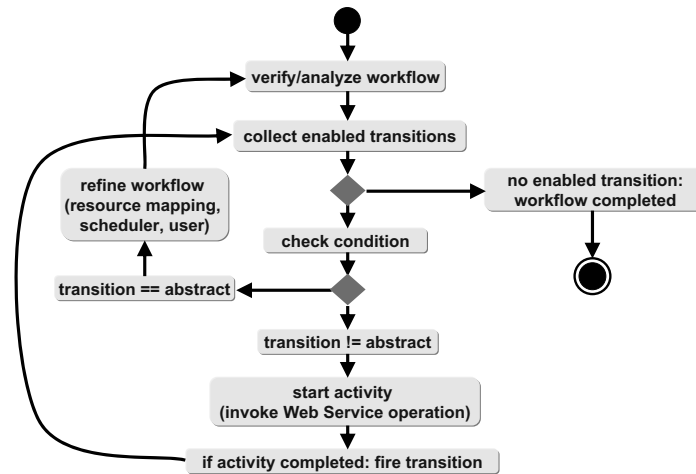
Figure 13.11: The kernel process of a Petri Net-based workflow enactment machine with automatic refinement.

ing of a Petri Net fully describes the state on the workflow level. Therefore it is enough just to store the current workflow description document together with the contents of the tokens in order to get a nonvolatile snapshot of the workflow state. As there is no principal difference between the descriptions of an initial and a running workflow, it is possible to just reload the stored workflow description in order to recover a terminated or aborted workflow.

Transactional workflows are workflows that are either entirely completed or aborted in order to guarantee the integrity of the workflow management system. In general, this can only be achieved if each of the workflow's activities is transactional itself. In traditional database systems, transactions are specified according to the ACID properties (atomicity, consistency, isolation, and durability) [219]. The ACID properties, however, are very difficult to guarantee in a distributed environment with long-running transactions, so here the so-called *compensation transaction* is often used instead, with limited rollback and isolation capabilities [56]. A Petri Net-based workflow engine could support transactional workflows by recording the whole workflow history and implementing a mechanism that calls the underlying compensation mechanisms of the invoked activities in order to roll back the workflow.

Petri Nets are appropriate to support implicit as well as explicit *fault management* mechanisms. Implicit fault management can be inherently included in the middleware and is invoked either by lower-level services regarding fault management of single activities or by higher-level services considering the workflow of the e-Science application. This type of implicit fault management
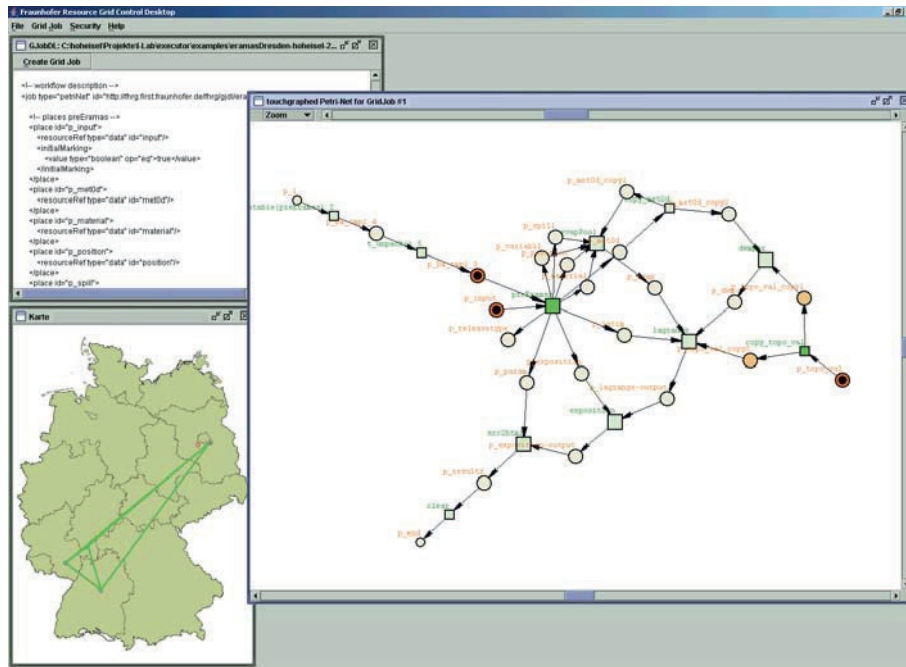
Figure 13.12: The Petri Net-based workflow enactment front end of the Fraunhofer Resource Grid.

can be achieved by Petri Net refinement; e.g., by automatically introducing a sub-Petri Net that restarts the activity if the submission or execution fails. Explicit fault management in our definition refers to user-defined fault management. Within the Petri Net workflow model, the user defines the fault management explicitly by including user-defined fault management tasks in the Petri Net of the application. Hoheisel and Der [194] give more details about how to model and enable fault management using Petri Nets.

13.5 Conclusions

Petri Nets are a well-established approach in computer science for modelling and analyzing distributed processes, whereas many workflow management systems in the e-Science domain use other workflow formalisms, such as BPEL and DAG. The reasons for this are on the one hand the strong influence of industrial groups enforcing their own standards (e.g., BPEL) and on the other hand the wish to keep things very simple (DAG). The Petri Net approach is, nevertheless, a good candidate for becoming a vendor-independent standard for graph-based modelling of e-Science workflows, as it has formal semantics—which offer a profound theory background—and provides advanced analysis

methods. An encouraging alternative is to base the workflow engine on Petri Nets and to map other high-level workflow formalisms (e.g., BPEL, UML) onto Petri Nets just before the workflow enactment. It is worth mentioning that many commercial workflow management systems in the business process domain are based on Petri Nets and that the semantics of UML 2.0 activity diagrams have been strongly influenced by them.

There exist several classes of Petri Nets that are suitable for different purposes. In order to apply the Petri Net approach to the choreography, orchestration, and enactment of real-world e-Science workflows, High-Level Petri Nets (HLPNs) provide an adequate solution. However, we propose to extend the classical definition of HLPN for this purpose. We introduce a special notation for conditions (using the XPath 1.0 standard) to facilitate reactive workflow management in addition to the control and data flows that are explicitly modeled by the edges of the Petri Net. Transitions do not fire instantaneously, as they represent the invocation of real-world software components or services. The content of data tokens represents the real data that are produced by external software components or services. We use edge expressions to link places with specific software component or service parameters.

One drawback of the Petri Net approach is the fact that the graph may become very huge for complex and fine-grained systems. One solution to this problem is the use of hierarchical Petri Nets, where one transition represents a whole sub-Petri Net. The main application area for Petri Nets is in loosely coupled systems that exhibit a certain granularity of components.

Acknowledgments

14

Adapting BPEL to Scientific Workflows

Aleksander Slominski

14.1 Introduction

In this chapter, we examine the degree to which a *de facto* standard business Web services workflow language, Business Process Execution Language for Web Services (BPEL4WS), can be used to compose Grid and scientific workflows. As the Grid application models, such as Open Grid Services Architecture (OGSA) [146], move toward Web services and service-oriented architecture (SOA) [135], supporting Web services is becoming a requirement for a Grid workflow language.

There is a great potential value in leveraging an established workflow language standard from the business domain, as it allows for a productive sharing of workflow definition documents using commercial and open-source tools, leveraging existing training and support, documentation, books, etc. BPEL, even if it is not a primary workflow language in scientific projects, is a very good candidate for a common language for sharing workflows between different projects. (This can be achieved by allowing a workflow to export and import BPEL workflows in scientific projects.) A high-level overview and more details about differences between scientific and business workflows can be found in Chapter 2.

In this chapter, we identify the requirements that we have found to be important for scientific and Grid workflows that are not yet common in business workflows and some that may never become commonplace in business workflows (such as an experimental approach to constructing workflows). To this end, we propose a set of additional capabilities that are needed in Grid workflows and show how they can be implemented with a concrete example.

14.2 Short Overview of BPEL

The following is not meant to be a comprehensive treatment of the BPEL language. Instead, our goal is to highlight key features and describe parts of BPEL

that are particularly important in the context of scientific workflows. Additional information can be easily obtained from many online sources, books, and articles, and the BPEL specification itself is the best resource for all the details.

14.2.1 Origins of BPEL

Business Process Execution Language for Web Services (BPEL4WS), when created in 2002, replaced two workflow languages created earlier by IBM and Microsoft. IBM's Web Services Flow Language (WSFL) had a graph-oriented view on how to describe workflows, and Microsoft's XLANG represented a more block-structured approach. BPEL merged both views and added extensive support for structural handling of errors with try/catch constructs and compensation handlers. The initial 1.0 release of BPEL was followed in 2003 by version 1.1 [24], which clarified and improved several parts of BPEL 1.0. Later that year, BPEL was submitted to the Organization for the Advancement of Structured Information Standards (OASIS), and since 2004 it has been standardized as WS-BPEL 2.0 [315]. The major change in version number and changed name reflect that OASIS WS-BPEL 2.0 will be a major revision and not fully compatible with 1.x versions. In this overview, we will concentrate on BPEL4WS 1.1.

14.2.2 BPEL Capabilities

BPEL4WS is designed from the ground up to work with Web services, and each BPEL workflow is a Web Service as well. This makes BPEL an easy fit into Web services middleware and allows for easy composition of hierarchical workflows: A BPEL workflow is a Web Service that can be used inside another BPEL workflow that may again be used as a Web Service inside yet another BPEL workflow.

BPEL allows one to describe a blueprint of a workflow (called an "abstract BPEL") that highlights important behaviors without specifying all details. The intention is to allow the definition of publicly visible behaviors of a workflow, hiding details that may differ between implementations of a blueprint. This is like an interface or a contract in programming languages. The abstract BPEL is then implemented by a BPEL workflow that has all details filled in (called "executable BPEL").

BPEL mandates support for XPath 1.0 as an expression language to manipulate XML. XML schemas are supported as a type system that is mainly used in Web Service Description Language documents (WSDLs) referenced by BPEL workflows. WS-Addressing and asynchronous conversations are supported with the ability to use message correlations to flexibly relate messages that are part of a workflow execution. Those specifications provide a solid set of

tools to manipulate XML messages, extract and combine parts of XML messages, describe and validate the content of XML messages, and route messages to Web services.

BPEL has a strong set of control structures (loops, conditions, etc.) and good support for catching and handling exceptions (faults) and reversing changes by using compensations. Compensations are particularly important for long-running workflows that need to "undo" changes in case there are unrecoverable errors in services used by a workflow and global consistency must be restored before a workflow is finished. Using traditional transactions may not be an option, as long-running workflows could lead to transactional locks being held for a very long time. BPEL is a control-driven workflow, but modeling data-driven workflows that are translated to BPEL is possible (for more details on the differences between the approaches, see Chapter 11).

14.2.3 Structure of The BPEL Workflow

The overall structure of the BPEL workflow is shown in Figure 14.1. A BPEL workflow definition is inside a `<process>` element. This element is a container for a set of other elements, such as `<partnerLinks>` and `<variables>`, and one activity that is an entry point to a workflow (typically a `<sequence>`).

XML is a very verbose language, so in the interest of keeping examples readable, we will use a simplified notation instead of XML. In this notation, text indentation indicates a level of nesting of an XML element, and XML attributes are simply listed after an element name as `name=value` pairs (possibly on multiple lines). We will also omit details that are not important for a given example, such as "messageType" attributes for variables.

By using this compact notation, the example from Figure 14.1 can be rewritten in a shorter form as shown in Figure 14.2.

14.2.4 The Most Common BPEL Activities

Basic Activities.

BPEL provides a set of simple constructs for sending and receiving messages. A typical BPEL workflow will start with a `<receive>` activity and end with a `<reply>` activity that sends a reply message to whoever sent initial messages that were received. It is easy to send a message to other Web services (they are called partners in BPEL) by using the `<invoke>` activity. There are two versions of `<invoke>`: the one-way version, when only inputVariable is present; and the request–response version, when both inputVariable and outputVariable are present in `<invoke>`.

Data Manipulation.

All messages in BPEL are contained in variables. Variables are passed between BPEL activities. To copy and change the content of variables, `<assign>` activ-

```
<process name="BpelProcessName" targetNamespace="..."
    xmlns="http://schemas.xmlsoap.org/ws/2004/03/business-process/">
  <partnerLinks>
     <partnerLink name="partnerA"
       partnerLinkType="wsdl:partnerALinkType"
       myRole="myRoleInRelationToPartnerA"/>
     ...
  </partnerLinks>
  <variables>
    <variable name="varA" messageType="wsdl:MessageA"/>
    ...
  </variables>
  <!-- this is executable part of workflow -->
  <sequence>
     <receive partnerLink="partnerA" portType="wsdl:partnerALinkType"
       operation="doSomething" variable="varA" />
     <assign>
       <copy>
         <from>$varA.someParameter</from>
         <to>$varB.anotherInfo</to>
       </copy>
     </assign>
     <invoke partnerLink="partnerB" portType="pb:anotherPartnerPT"
       operation="doSomethingElse" inputVariable="varB"
       outputVariable="varC" />
     .... <!-- here something more happens -->
     <reply partnerLink="partnerA" portType="wsdl:partnerALinkType"
        operation="doSomething" variable="results"/>
  </sequence>
</process>
```

Figure 14.1: Outline of a BPEL process in XML.

ity can be used — it supports the XPath language to select and modify XML content (other data-manipulation languages may be used as extensions to BPEL, but only XPath is required).

Structured Activities.

BPEL has a set of structural activities similar to what is available in procedural languages. Loops (`<while>`) and conditions (`<switch>` and `<if>` in BPEL 2.0) are supported. In addition to a block-level construct — `<sequence>` — BPEL also supports starting multiple threads of execution in parallel by using `<flow>`.

```
process  name="BpelProcessName"
  variables $varA, $varB, ...
  partnerLinks "partnerA", "partnerB", ...
  # this is executable part of workflow
  sequence
    receive partnerLink="partnerA"
      operation="doSomething" variable="varA"
    assign copy from $varA.someParameter
      to $varB.anotherInfo
    invoke partnerLink="partnerB"
      operation="doSomethingElse" inputVariable="varB"
      outputVariable="varC"
    # here something more may be added
    reply partnerLink="partnerA"
      operation="doSomething" variable="results"
```

Figure 14.2: Outline of a BPEL process without XML.

Graph-Based-Workflows.

This last capability is a key to supporting graph-based composition. It is easy to start many activities in parallel with `<flow>`, and BPEL allows one to define graph-like dependencies between activities. Each activity (a node in a graph) may have a set of incoming and outgoing links. For an activity to start its execution, all incoming links must be enabled. When an activity is finished, all of its outgoing links will be enabled, and that will enable related incoming links for other activities and so on (additional details can be found in the BPEL specification). This capability allows one to build any graph in BPEL, and the interesting part is that BPEL allows the programmer to mix structured and graph approaches in one workflow.

14.2.5 Limitations of BPEL

BPEL does not have a parallel loop. This is particularly important for scientific code. If the number of iterations is constant, it is possible to use `<flow>` to start multiple activities in parallel, but this approach does not work if the number of iterations depends on an input to a workflow. A parallel loop can be simulated with nonblocking invocations of a Web Service (that is, a BPEL subworkflow), but such invocations are hard to track, and in general establishing communication channels between subworkflows and the main workflow may be difficult (such as detecting when all subworkflows finished execution successfully).

This and some other limitations of BPEL 1.1 (such as limited capabilities of the `<assign>` activity) may be fixed in the upcoming OASIS WS-BPEL 2.0 when work on it is finished.

14.3 Goals and Requirements for Scientific Workflows in Grids

Based on our experience, we identified a set of requirements that are desirable for a scientific workflow language and a workflow execution environment (typically called a "workflow engine") for Grids. Those requirements can be used to evaluate any Grid workflow language, and later we will use them to see how BPEL meets requirements for a scientific workflow language in Grids. However, they will vary in different domains. For example, see Chapter 16, where requirements for semantic workflows are discussed, and Chapter 26, with requirements identified in the SEDNA scientific modeling environment.

Generic Design Goals

Use of Standards. Standards help to increase the reuse of workflows and help share parts of whole workflows. We believe that using an industry standard Web services workflow language is beneficial to scientific workflows. Besides greater reuse and sharing of tooling, it also allows to leverage existing knowhow in tutorials, documentation, and other resources available on the Internet. Only when a standard workflow language does not meet requirement of a scientific workflow (either for a generic or a specific scientific domain) and such a language cannot be extended to meet requirements (or extensions are too complicated) should a new workflow language be created. BPEL is the current *de facto* standard for Web services based workflows in business environments and therefore is a good candidate for a standard-based scientific workflow language for Grids that use Web services.

Integration with Web Architecture. In addition to running workflows, a Grid workflow engine should follow the general design of a Web Architecture [428, 455]. In particular, using URIs simplifies integration of information resources maintained in a workflow engine with portals, scientific notebooks, data management systems, and any other scientific or Grid tools. Using URIs allows to reference workflows (and their parts) already stored in a workflow engine. In particular, this makes it easier to integrate a workflow engine with emerging Semantic Web standards [454] that use URIs to identify everything and makes such semantically enriched information machine-understandable.

Integration with portals. A workflow engine should be easy to integrate into an existing scientific portal. At a minimum, a workflow engine should expose a set of monitoring and administrative operations that can be accessed by portals as Web services. It would also be beneficial if a workflow engine used Semantic Web data standards [454] and was easy to integrate with scientific data management systems such as myGrid [308] and myLEAD [359].

Requirements Specific to Scientific Workflows

Integration with legacy code. In scientific workflows, it should be easy to use components that are not Web services. This requirement can be met by either

directly adding support for specific legacy or special execution capabilities or by taking advantage of WSDL's flexibility. Both choices are common. However, using WSDL as a common abstraction to describe a "service" that is not necessarily a Web Service provides a uniform and elegant abstraction. A service accessed from a workflow can be anything from a "real" SOAP-based Web Service over HTTP to a service that is just an executable running locally. This is advantageous, as it simplifies a workflow language — it needs only to describe the orchestration of services described in WSDLs. Also, using WSDLs makes a workflow description more abstract and resilient to minor changes and allows the service implementation and location to be determined at the moment when the workflow needs to access a WSDL-described service. Apache Web Services Invocation Framework (WSIF [124]) is an example of a runtime environment that allows seamless access to any service that is described in WSDL and available over SOAP/HTTP, SOAP/JMS, as a local Java object, EJB, and even as embedded scripts. The other possibility is to embed actual code that interacts with legacy functionality into BPEL as an extension (for example, the proposed BPELJ [207], which allows one to embed Java code snippets into BPEL).

Experimental flexibility. A scientific workflow language and a workflow runtime environment should support a scientific laboratory notebook paradigm. They should allow a user to construct and develop a workflow incrementally, add and remove steps in a running workflow, modify existing workflow activities, allow repeat execution of workflow parts, modify workflow structure during execution, allow "branching" of a running workflow by cloning its state, and other operations that may come up when creating and running experiments. The exact set of capabilities depends on what is expected by the particular group of users that will be using Grid workflows.

History and provenance. A workflow execution environment for scientific workflows should automatically record the history of a workflow execution. A history log should have enough information to reproduce the workflow execution. That may include, but is not limited to, a time-ordered list of what services were executed (with enough information to uniquely identify the service instances used), what input and output messages were passed, or a record of any modifications to the workflow state. This information should be used to construct a full provenance record by an external service. It is also helpful if a workflow execution environment can use external provenance tracking services.

Reuse and hierarchical composition. To encourage workflow reuse, it is important that workflows be able to be used as parts in bigger workflows. This can be enabled if workflows are Web services themselves that can be part of other workflows. A workflow engine should support such composition by exposing each workflow as a Web or Grid Service.

Support very long-running processes. We expect that some workflows will be used to orchestrate Web and Grid services that may take very long periods of time to complete. Therefore, it is very important that a workflow engine

not only run and store the state of such workflows (so they can survive intermittent failures) but that it will also be easy to find, monitor, and manage such workflows.

Support running a very large number of workflows. In some scientific domains, running experiments involves starting a very large number of short-lived workflows. A workflow engine must provide capabilities to track all workflows started and make it easy to control them.

Grid-Specific Requirements

Accessing Grid resources. As it was mentioned before (in case of the legacy code), it is possible to use WSDL abstraction to hide implementation details of a service. The same approach can be applied to accessing Grid services from a workflow language. In the case where a WSDL abstraction is not used, a workflow language needs to have Grid-specific extensions to interact with specific grid protocols to use Grid resources. Emerging standards such as WSRF [100] provide a promising set of common and reusable WSDL protocol bindings specifically geared for Grids.

Dynamic resources. Support for on-demand creation of resources such as Grid services is essential. In addition to using WSDL abstractions to hide access protocols, one should be able to dynamically create Grid services when they are needed (for example, GFac [232]).

Designed for scalability. Nothing in the language design should prevent a scalable implementation of a workflow engine.

Integration with Grid security One of the most important and fundamental aspects of Grids is a requirement for strong and flexible authentication and authorization. There are many approaches that are popular. Therefore, a workflow language and engine should not mandate one particular security model but be flexible and open so that they can incorporate security capabilities as extensions.

14.4 Illustrative Grid Workflow Example

The LEAD (Linked Environments for Atmospheric Discovery [249]) is a National Science Foundation large information technology research (ITR) project that is creating an integrated, scalable cyberinfrastructure for mesoscale meteorology research and education. Crucial to the success of LEAD is the ability to not only compose services and data sources into applications but make them dynamically adaptive. This requirement is described in LEAD as Workflow Orchestration for On-Demand, Real-Time, Dynamically Adaptive Systems (WOORDS [250]). Some of the desired capabilities include the ability to change configuration rapidly and automatically in response to weather, continually be steered by new data, respond to decision-driven inputs from

users, initiate other processes automatically, and steer remote observing technologies to optimize data collection for the problem at hand. Those goals can be expressed as a more generic capability: Workflows that are driving LEAD applications must be responsive to events and be able to adapt their future execution paths (more details on workflows in LEAD can be found in Chapter 9).

Many typical scientific workflows are long-running and are composed of many steps, such as data acquisition, decoding, processing, and visualization. Those steps may need to be repeated and run in parallel for many hours or days before final results are available.
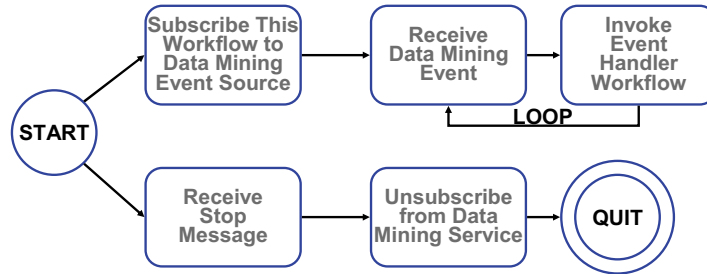
Figure 14.3: Persistent workflow that is monitoring data-mining events.

As an example, we take two workflows that illustrate types of workflows that LEAD plans to use and describe them in a simple scenario. Let us assume that we have a data-mining service that monitors real-time data streams and detects potentially interesting patterns such as the formation of a tornado. When such an interesting condition is detected, the mining service publishes an event to a message bus service (that may support standards such as WS-Eventing or WS-Notification). A user may choose to run a permanent and persistent workflow that subscribes to data-mining events. A simplified graph of such a workflow is shown in Figure 14.3, and in Figure 14.4 we show an outline of a BPEL process for that workflow. The BPEL document has a list of declared variables and a list of partner links. Each partner link represents a Web Service that is either using the workflow or is used by the workflow (or both). BPEL does not specify how the location of the partner is established, and typically this is done statically in a workflow deployment phase. However, more dynamic behavior to determine location of partners is possible (either when a new workflow instance is created or even during workflow execution — this is discussed in more detail later when the workflow life-cycle is described). When an instance of this sample workflow is created and starts running, the first activity executed is `<sequence>`. Then each activity inside `sequence` is executed, beginning with the first assignment. We have used short notation for `<assign>` (`$running = true`) to show that true is assigned to a variable

```
process  name="PersistentMonitoringWorkflowForUserFoo"
 variables $running, $stopMsg, $workflowName, $subscribeMsg, ...
 partnerLinks "WorkflowUser", "EventBus", "DataMining", ...
 sequence
  $running = true
  assign from partnerLink="DataMining"
               endpointReference="workflowEventConsumer"
         to "$subscribeMsg/wse:DeliveryTo/wse:NotifyTo"
  invoke name="SubscribeToEventService" partnerLink="EventBus"
      portType="wse:EventSource" operation="subscribe"
      inputVariable="subsrcibeMsg" outputVariable="subscribeResponse"
  flow # two parallel sequences
   sequence
    receive name="ReceiveStopMessage" partnerLink="WorkflowUser"
            variable="stopMsg"
    $running := false
   sequence name="RunSequence"
    while $running is true do
     sequence
      $workflowName := "EventHandlingWorkflow"
      receive name="ReceiveEv" partnerLink="DataMining"
              variable="event"
      invoke name="StartEventHandlerWf" partnerLink="WorkflowEngine"
             portType="wse:EventSource"
             operation="startNewWorkflowInstance"
             inputVariable="workflowName"
             outputVariable="workflowLocation"
      assign from $workflowLocation
             to partner "EventHandlerWorkflow"
      invoke name="InvokeEventHandlerWorkflow"
             portType="wse:UserWorkflow" operation="processEvent"
             inputVariable="event"
    exit # quit workflow
```

Figure 14.4: Outline of BPEL document describing example workflow.

named "running." The second <assign> in the sequence is used to copy the location ("endpoint reference") of the workflow Web Service (as mentioned before, when a BPEL workflow is started it becomes a Web Service) to the "subscribeMsg" variable. This variable holds the content of a message that is sent to the data-mining service to subscribe for events. Sending the message is accomplished by the <invoke> operation. This is request–response invocation (it has both input and output variables) and is a blocking operation; i.e., further workflow execution of this thread is stopped until a response arrives. The response may be either a response message, in which case its content is copied to the output variable, or it may be a fault message. BPEL has soph-

isticated support for handling faults, but in this example it is not needed and the default behavior works well. By default, if a fault happens, the workflow instance is terminated with an error and the workflow execution environment may notify a user about an abnormal termination of the workflow.

The next activity executed in the sequence is `<flow>`. It splits execution into two parallel threads. The first one will immediately block on `<receive>`. When this workflow Web service receives "stopMsg" then this thread will unblock and set the "running" variable to false. Since this is the last activity in the flow sequence, this thread will be terminated. The other thread started in the flow is more persistent. We have a `<while>` that keeps executing until the "running" variable becomes false. In this loop, the `<receive>` will block until an event is received from the data-mining service. If more than one event is received and the workflow is busy, then events are put into a queue and no event is lost. The next activity in the loop creates a new workflow instance by calling a workflow execution service (workflow engine) to create a workflow instance identified by the "EventHandlingWorkflow" string. When the workflow instance is created, it may be further configured (as explained later in the description of workflow life-cycle), but in this example we just use the new workflow location to invoke it. This invocation is one-way (no output variable), so there is no need to wait for the result of the invocation and the loop can continue. When the "running" variable becomes false (after receiving the stop message in the other thread), the loop will be exited. This is not an optimal solution, as the loop may still be blocked, waiting to receive an event. Unfortunately, BPEL does not have the capability to interrupt blocking waits (still, some BPEL implementations may allow one to configure timeouts for blocking receive/invoke, and, in such a case, a workflow will eventually finish). For simplicity, we could just use `<exit>` in the thread that received a stop message (as shown in Figure 14.3), but in this example we show how multiple threads inside a BPEL workflow instance can communicate by using shared variables (as it is an interesting capability to have in more complex workflows).

When an event is received, the workflow will start other workflows ("Event-HandlingWorkflow") such as the one depicted in Figure 14.5. This event handler workflow may finish quickly (when the event is deemed "uninteresting"), or it may continue running a for long time to determine if anything interesting happens. That may lead to generation of other events that may trigger execution of other workflows and eventually sending of an urgent notification to a user that something like a tornado is happening with a high probability.

In Figure 14.6, we have an example of BPEL code to implement the workflow graph shown in Figure 14.5. As we see in those examples, BPEL is capable of describing complex workflows, but more than a workflow language is needed. An important part of a workflow execution is monitoring. Users should be able to determine the state of the workflows they started. Users may want to know what workflows are waiting for services, what the intermediary results are, etc.
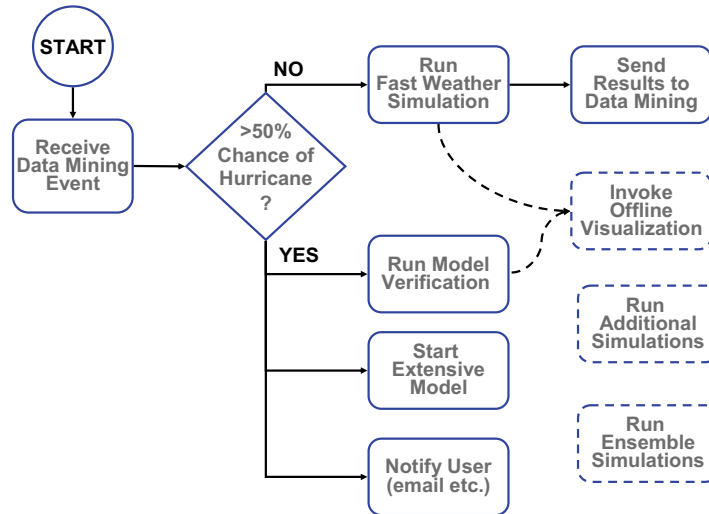
Figure 14.5: Workflow instance launched in response to a data-mining event.

When something interesting is noticed in a workflow, a user should be able not only to steer the workflow execution (start, stop, pause) but also to modify either the state of one particular workflow or a whole group of similar workflows. This is an important requirement for a workflow execution environment in LEAD: Workflows are built incrementally and can be modified by a user even when they are running (we depict some possible modifications in the second workflow by drawing them with dashed lines in Figure 14.5). The user can add new steps or rearrange existing steps to meet new requirements. Workflows are frequently changing, reflecting what the user wants to get done. This experimental flexibility fits well in the scientific lab notebook paradigm mentioned under requirements. For example the user may add a new visualization step to the second workflow or modify the first workflow to launch another experimental workflow on a dedicated resource in response to events under some conditions. This experimental capability is part of a workflow engine and not a workflow language (BPEL) but nonetheless is important for running scientific workflows in Grids.

14.5 Workflow Life-Cycle on an Example of a GPEL Engine

We will now continue to delve into our example to see how aforementioned goals and requirements can be met. To make the description very concrete, we use the Grid Process Execution Language For Scientific Workflows

```
process name="EventHandlingWorkflow"
 sequence
  receive name="ReceiveEvent" partnerLink="WorkflowCaller"
          variable="event"
  if condition $event.probability < 50.0 then
   sequence
    invoke name="WeatherSim" partnerLink="WeatherSimulationExecution"
          portType="fw:FastWeatherSim" operation="runFastCheck"
          inputVariable="event" outputVariable="runResults"
    invoke name="SendResults" partnerLink="DataMiningService"
          portType="dm:DataMining" operation="runDataMining"
          inputVariable="runResults" outputVariable="sendStatus"
  else
   flow  # start 3 parallel activities
    invoke name="ModelVerification" partnerLink="ModelVerification"
          portType="fw:ModelVerification" operation="verify"
          inputVariable="event" outputVariable="verificationResults"
    invoke name="WeatherSim" partnerLink="WeatherSimulationExecution"
          portType="fw:WeatherSim" operation="runExtensiveModel"
          inputVariable="event" outputVariable="runModelResults"
    sequence
     $notifyMsg/userName = "foo"
     $notifyMsg/event = $event
     invoke name="NotifyUser" partnerLink="NotificationService"
           portType="dm:UserNotificationService"
           operation="notifyUser" inputVariable="notifyMsg"
  exit
```

Figure 14.6: Outline of BPEL document describing example event-handling workflow.

(GPEL4SW) environment developed at Indiana University. Following the requirements for standards and reuse, we use BPEL. GPEL4WS consists of two parts. The first part is the GPEL language, defined as a subset of the BPEL 1.1 language. We are gradually expanding the supported subset with the goal of supporting the final version of the OASIS WS-BPEL 2.0 standard in future versions of GPEL. However, as BPEL is still under a standardization process in OASIS, for now we provide a stable set of semantics by freezing the set of BPEL constructs in GPEL namespace.

When compared with BPEL4WS, GPEL4SW adds support for Grid-oriented life-cycle and workflow management operations (those were intentionally left out of the BPEL4WS standardization scope). The GPEL4WS API has a set of standard XML messages that can be used to find capabilities of a workflow engine, deploy workflows, start them, and control their execution. This workflow life-cycle is described in detail in the following sections.

14.5.1 Workflow Composition

There are many tools that can be used to prepare BPEL workflows. They range from simple or advanced XML editors (sometimes with XML schema support to assist in XML creation) to graphical tools that provide an intuitive GUI to compose workflows by connecting Web services in a graphical way by hiding from users the XML text of the BPEL process and generating XML automatically when needed. Because graphical tools operate on a higher level of abstraction, they usually support only a subset of the BPEL language and provide functionality that is specialized for certain groups of users. For example, Sedna (see Chapter 26) provides a convenient GUI to manipulate high-level abstractions such as an indexed flow construct (a representation of a parallel loop construct that is not available in BPEL) and supports visual macros and plug-ins to reuse fragments of BPEL code. In LEAD, we developed XBaya Workflow Composer [382], which provides an intuitive GUI tool to compose Web services and generate BPEL or GPEL workflows. XBaya provides an extensible library of LEAD services and allows a user to drag-and-drop services and connect them together. In addition to workflow composition, XBaya allows monitoring and visualizing workflow execution (for example, visual cues, such as colors, are used to show the state of services during execution).

14.5.2 Workflow Engine Introspection

The way a client discovers the capabilities of a workflow engine differs greatly from one implementation to another. Typically there is no mechanism to discover capabilities of a workflow engine, but the capabilities of a particular workflow runtime installation are known beforehand. In the GPEL4SW API, we specified the discovery process by defining an extensible way to do a workflow engine introspection. This makes it easier for clients to interact with different GPEL implementations and to discover additional capabilities. The discovery is performed by obtaining (typically using HTTP) an introspection XML document. This document describes capabilities of a GPEL engine. For example, one of the capabilities is a location where new documents can be created inside the GPEL engine. When a workflow deployment tool (such as the XBaya Workflow Composer) is deploying a workflow to a GPEL engine, it must first obtain an introspection document to find a location where the deployment documents can be created (see Section 14.5.3 for details). The location of the introspection document can be found in multiple ways. It can be hard-coded into the client software, but a more flexible approach is to allow a user to specify the location of a workflow engine. This location may point to a Web page that contains a link to the actual GPEL introspection document.

14.5.3 Workflow Deployment

Before a workflow can be started it needs first to be deployed. The deployment process defines how to associate Web services (described in WSDLs) and the actual workflow process definition (BPEL/GPEL) together. There may be additional deployment-specific options, such as security (who can start workflows), that must be specified. This process is not standardized in the BPEL specification, as it was declared out of the scope of BPEL. As a consequence, the way the deployment is accomplished in different BPEL engines varies greatly between implementations. This is actually good for Grids, as it allows us to define a deployment process that fits the dynamic requirements of Grid environments.

```
<entry xmlns="http://www.w3.org/2005/Atom">
  <title>GPEL template for Workflow Foo</title>
  <summary>GPEL template for Workflow Foo.</summary>
  <content type="application/x-gpel+xml">
     <template xmlns="http://schemas.gpel.org/2005/grid-process/" />
  </content>
  <link rel="http://schemas.gpel.org/2005/wsdl"
    href="http://gpel.example.org/foo.wsdl"/>
  <link rel="http://schemas.gpel.org/2005/gpel"
    href="http://gpel.example.org/foo.gpel"/>
</entry>
```

Figure 14.7: An example GPEL workflow template.

The deployment process in BPEL engines is implemented by sending a set of XML documents, which includes, at a minimum, a definition of BPEL workflow, but also typically includes WSDL files for all partners and related partner link types. Sometimes, instead of sending documents, only their locations (URLs), are sent during deployment. There are many protocols that can be used in a BPEL engine for deployment, and they range from simple HTTP POST and SOAP over HTTP to specialized binary protocols. A particular BPEL engine may provide a programmatic API to do the deployment, but there may also be no way to do programmatic deployment if the deployment is done from a GUI application or a servlet that uses proprietary mechanisms for deploying workflows.

We believe there is a very simple way to do BPEL workflow deployment and that it may have a chance to be supported in multiple BPEL implementations eventually. It seems that the simplest way to do deployment is to use HTTP POST and send all workflow-related documents to the workflow engine. That is how we defined deployment for GPEL. First, a client application needs to send all documents to a GPEL engine (i.e., WSDL and BPEL/GPEL

process definitions). The documents are stored in the GPEL engine, and each one gets a unique URL. Using URLs simplifies the linking of documents (and is consistent with the requirement of using Web Architecture). When the document is stored in the GPEL engine, it is validated (so no invalid BPEL or GPEL workflow definitions can be executed, and errors should be reported as early as possible). The last step of the deployment is to create a simple XML document that describes how to link different documents into a workflow template (see Figure 14.7). The GPEL workflow template has all the information that is necessary to create workflow instances. The GPEL engine will check that inside the template document there is a link to the workflow document (BPEL or GPEL) and will validate that all required WSDL port types and partner link types are present (actual service bindings and locations can be set later during workflow instance creation). This step finishes deployment.

14.5.4 Workflow Instance Creation

We recommend separating the workflow creation step from actual workflow execution. This is different from what is described in the BPEL specification, where workflow instances are created implicitly when a message marked as "createInstance" is received. Making the process explicit allows for fine-grained control over a workflow instance execution environment. However, both approaches can be supported in one workflow engine.

The separate step of workflow instance creation allows one to set up the workflow instance to use specific Grid or Web Service instances. This is very important in Grid environments where a workflow instance may be part of a bigger application and will run on dedicated Grid resources requiring a special workflow setup for each execution (such as creation of security credentials and allocation of cluster nodes).

The GPEL workflow instance document, similarly to the GPEL template document, is deployed by using HTTP POST and essentially contains a set of links. The most important link in the workflow instance document is to the workflow template that this workflow instance "implements." A user must replace abstract WSDLs (if any) with concrete WSDLs and can replace any WSDL used in deployment with a new version that points to a service instance to use just for this workflow instance.

Workflow Instance State

A workflow when running is stateful, and its state is similar to a state in a typical program: There is a set of threads, and each thread has a set of variables. A BPEL engine needs to maintain a set of variables that are scoped (and in this way similar to local variables in a thread), list of active threads of execution, and what each thread is doing: What activity is executing? Is it blocked waiting for a response? What messages are in outgoing and incoming queues? And so on (see Figure 14.8).
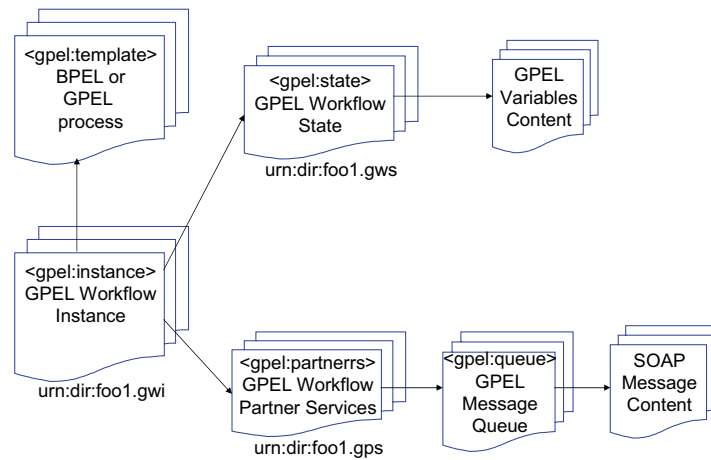
Figure 14.8: GPEL workflow instance state.

Using XML is a very convenient way to expose the workflow state. This not only allows to monitor the state of a workflow instance execution, but a user or an automatic tool (such as a case-based reasoning system or a semantic agent) may modify a running workflow simply by modifying XML documents describing the workflow state. If both the workflow process definitions (BPEL document) and a workflow instance state document are modified by a user, then this is not just a simple modification of variables or what activity the workflow is executing (as in a debugger) but can be a structural change to the workflow (such as adding new activities).

14.5.5 Workflow Execution

At this point, after a workflow composition, deployment, and creation of a workflow instance, we have a running workflow. The workflow execution is the part that is the most important for taking full advantage of Grid resources. A Grid workflow engine must be able to request and create Grid resources on-demand. This can be accomplished by leaving the decision about what service to use to the very moment when the workflow engine needs to send a message to a Grid service. At that point, the service may be created on the best available resource and used by the Grid workflow engine.

Workflow Instance Control

The workflow instance state document contains all information pertaining to a workflow execution. An interesting consequence is that a user is able to go back in time to any previous state of workflow and continue execution from that moment by requesting that the GPEL engine use a previously stored

workflow instance state document. This is particularly useful to allow "cloning" of workflow execution: A user can explore possible execution paths by storing a workflow instance document and creating a workflow instance clone to experiment with an alternative execution path. This capability is limited by the level of support from services used by workflow instances — in particular, services used by the workflow may need to support checkpointing. In a more traditional sense, the workflow state can be monitored to do debugging and, in particular, to request a step-by-step execution of the workflow instance. This is a very useful capability that can be used even by nonprogrammers when a suitable high level user interface is provided. For example, the metaphor of VCR remote start/pause/resume/stop buttons may be used. In our example, the persistent workflow (Figure 14.3) when started will continue running until a stop message is received. At any point, a user can request the workflow engine to pause the workflow execution and then examine the workflow state, make modifications, and either resume or step through the workflow execution.

The state of a workflow execution is not complete without knowing what messages were received and sent to Web services used during a workflow execution. A user should be able to view and modify messages and the location of Web services used in a workflow instance and request resending of a message to a failing service.

In our example, the second workflow that is launched to handle a datamining event is more experimental in its nature. The intention is that a user may tailor a workflow execution to particular needs related to an event received by that workflow instance. As an example, a user may want to steer what the workflow instance is doing or even add new activities to the workflow (such as invoking a visualization service).

14.6 Challenges in Using BPEL in Grids

BPEL meets the generic requirements identified in Section 14.3 quite well: It is becoming a leading standard for Web services workflows and can be well integrated with Web Architecture and with portals. The current limitations of BPEL, such as poor support for running a large number of parallel subworkflows, are either addressed in OASIS WS-BPEL or can be overcome by providing a higher-level language that is then translated into BPEL XML for workflow execution.

Other goals and requirements are independent of the choice of BPEL as a scientific workflow language — they have more to do with actual implementation of a workflow engine. First there are performance goals (such as scalability, clustering, administrative interface, etc.) that are generally desirable and become even more important in Grid environments that require support for dynamic resources and Grid security. As many scientific workflows may take a long time to complete and scientific experiments may require running a large

number of workflows, persistence is a desirable feature of a workflow engine implementation. A scientist should be able to start workflows and not worry that if a machine running the workflow engine is rebooted all work will be lost (and may need to be redone).

Some requirements are specific to scientific workflows. One is supporting history and provenance tracking. The other is experimental flexibility — many scientific workflows may never be "finalized" but need to be incrementally refined and modified during their execution. This capability is particularly important for long-running workflows where restarting (and losing all results) is not a good way to make changes in a workflow.

We hope that we showed that BPEL is a viable choice for a Grid workflow language but a BPEL workflow engine needs additional capabilities to meet requirements common in Grids. To this extent, we have shown, using the GPEL4SW as an example, how to define a set of simple XML documents that can be used to control the life-cycle of a workflow and, in particular, allow monitoring and steering of a running workflow instance. By defining a set of simple XML documents, we hope to increase the chances that such a workflow engine API will be used in different middleware applications (including portals) and that it may be implemented by other scientific workflow engines used in Grids.

The main challenges are in the area of interactions with legacy scientific code and Grid services. Approaches such as WSIF or BPELJ can help make BPEL workflows interact with non-Web services, but only time will tell how well they meet the requirements of scientific workflows. BPEL supports extensibility, so it is possible that in the future some extensions may become *de facto* standards for scientific BPEL in Grids.

Protocol-Based Integration Using SSDL and π-Calculus

Simon Woodman, Savas Parastatidis, and Jim Webber

A "service" has become the contemporary abstraction around which modern distributed applications are designed and built. A service represents a piece of functionality that is exposed on the network. The "message" abstraction is used to create interaction patterns or *protocols* to represent the messaging behavior of a service. In the Web services domain, SOAP is the preferred model for encoding, transferring, and processing such messages.

The SOAP Service Description Language (SSDL) is a SOAP-centric contract description language for Web services. SSDL provides the base concepts on top of which frameworks for describing protocols are built. Such protocol frameworks can capture a range of interaction patterns from simple request–response message exchange patterns to entire multiservice workflows within a composite application.

In this chapter, we will introduce the main features of SSDL and its supported protocol frameworks. We will focus on the Sequential Constraints (SC) SSDL protocol framework for capturing the messaging behavior of Web services acting as part of a composite application or multiparty workflow. The SC SSDL protocol framework can be used to describe multiservice, multi message exchange protocols using notations based on the π-calculus. By building on a formal model, we can make assertions about certain properties (e.g., liveness, lack of starvation, agreed termination, etc.) of workflows involving multiple Web services. We will also provide a use case detailing how SSDL can be used in partnership with Windows Workflow Foundation.

15.1 Introduction

SOAP is the standard message transfer protocol for Web services. However, the default description language for Web services, Web services Description Language (WSDL) [457], does not explicitly target SOAP but instead provides a generic framework for the description of network-exposed software artifacts. WSDL's protocol independence makes describing SOAP message transfers

more complex than if SOAP had been assumed from the outset. WSDL's focus on the "interface" abstraction for describing services makes it difficult to escape the object-oriented or remote procedure call mindset and focus on message orientation as the means through which integration is achieved.

The SOAP Service Description Language (SSDL) [336–338] is an XML-based vocabulary for writing message-oriented contracts for Web services. SSDL focuses on the use of messages combined into protocols (arbitrary message-exchange patterns) to describe a SOAP-based Web service and is intended to provide a natural fit with the SOAP model.[1]

The SOAP processing model [390] in turn provides the fundamental architectural constraints for the Web services stack, as shown in Figure 15.1. While the stack itself is unremarkable, it serves to make the strong point that all Web services must support SOAP and that services interact through the transfer of SOAP messages. That is, in a Web services based environment (which includes workflows composed from Web services) we assume that other communication means, such as Remote Method Invocation (RMI) [400] and Common Object Request Broker Architecture (CORBA) [322], are merely transport protocols for the transfer of SOAP messages. Such protocols are thus out of scope and do not impact the transfer of messages within the Web services domain.

The work presented in this chapter is specifically bounded by the SOAP, metadata, and process choreography layers from the diagram in Figure 15.1. While the SOAP layer provides the fundamental architectural constraints to a service, the process choreography layer orchestrates the workflow at a global (or application) level. However, it is the introduction of SSDL at the metadata level that enables choreographies to enlist SOAP-based Web services and be able to determine in advance whether the message exchanges supported by the chosen services will lead to workflows that complete in consistent, safe states.

The remainder of this chapter shows how SSDL, and particularly the Sequential Constraints SSDL protocol framework, achieve the goal of supporting the description of a contract for services involved in multiparty workflows, where the capabilities of a service in one part of the workflow must be matched by capabilities of other services in that workflow. Section 15.2 defines the basic service-oriented model that is espoused by the SOAP processing model. Section 15.3 introduces SSDL contracts and how they can be extended through protocol frameworks. Section 15.4 provides an in-depth look at the Sequential Constraints (SC) SSDL protocol framework and highlights its relationship to the π-calculus. Section 15.5 presents a use case of how SSDL and the SC protocol framework can be used in a typical multiservice e-Science scenario, while Section 15.6 relates the N-way contract framework that SC provides to emerging Web services middleware technology. Final remarks and conclusions about the utility of SC are provided in Section 15.7.

---

[1] It is assumed that a "Web service" by definition must support SOAP as its native message-transfer protocol.
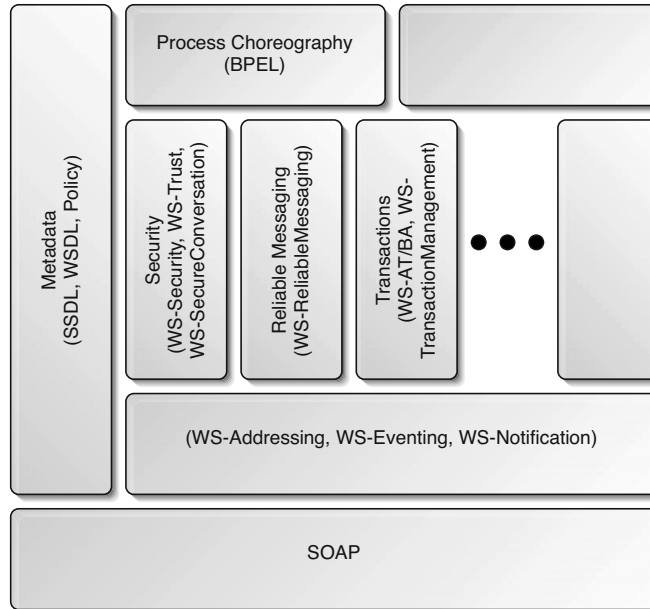
Figure 15.1: The Web services stack (adapted from [292]).

15.2 Service Orientation

While service orientation is not a new architectural paradigm, the advent of Web services has reinvigorated interest in the approach. It is a common misconception that Web services are a form of software magic that automatically corrals an application architect toward a scalable, robust, dependable, and loosely coupled solution. Certainly it is possible to build service-oriented applications using Web services protocols and toolkits to meet such quality-of-service requirements, but, as with any approach and suite of technologies, this is possible only after carefully considering the solution's design and by following the right architectural principles. Furthermore, the use of Web services technologies does not implicitly lead to a service-oriented solution; indeed Web services based distributed applications could be architected according to the principles of other paradigms, such as resource or object orientation.

As researchers and developers have rebranded their work to be in vogue with the latest buzzwords, the terms "service" and "service-oriented architecture" (SOA) have become overloaded. In what follows, we treat a service as the logical manifestation of some application logic that is exposed on the network. Such a service may encapsulate and provide access to any number of physical or logical resources (such as databases, programs, devices, humans, etc.). A service's boundaries are explicit, it is autonomous, it exposes message

schema information, and its compatibility with other services is determined through metadata information such as policies and protocol description contracts [292]. The interaction between services is facilitated through the explicit exchange of messages. We treat the message abstraction as a first-class citizen of service-oriented architectures and we promote message orientation as the paradigm of choice for enabling the composition of services into workflows.

A service such as that shown in Figure 15.2 consists of some resources (e.g., data, programs, or devices), service logic, and a layer responsible for the processing of incoming and outgoing messages. Messages arrive at the service and are acted on by the service logic, utilizing the service's resources (if any) as required. Services may be of any scale, from a single operating system process to enterprise-wide business processes.
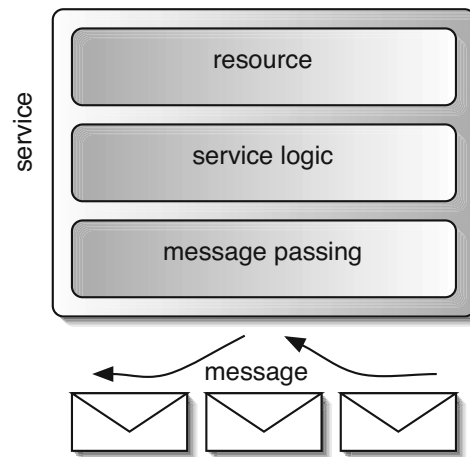
Figure 15.2: The typical structure of a service.

Services may be hosted on devices of arbitrary size (e.g., workstations, databases, printers, phones, personal digital assistants, etc.), providing different types of functionality to a network application. This promotes the concept of a connected world in which no single device and/or service is isolated. Interesting applications and workflows are built through the composition of services and the exchange of messages.

15.2.1 Messages

A message is the unit of communication between services. Service-oriented systems do not expose abstractions such as classes, objects, methods, or remote

procedures. Instead, services bind to messages transferred between them. A number of such message transfers can be logically grouped to form message exchange patterns (e.g., an incoming and a related outgoing message may form a "request–response"). Such multimessage interactions can be grouped to form *protocols* to represent well-defined behaviors.

15.2.2 Protocols, Policies, and Contracts

The messaging behavior of a service in a distributed application is specified by a set of messages and the order in which they are sent and received (i.e., the supported protocols). This is a departure from the traditional object-oriented world, where behavioral semantics are associated with types, exposed through methods, and coupled with particular endpoints.

Protocols and other metadata are usually described in contracts to which services must adhere. A contract is a description of the policy (e.g., quality of service characteristics such as security, support for reliable messaging, etc.), along with a syntactic description of the message structure and protocols that a service supports.

15.3 SSDL Overview

The primary goal of an SSDL contract is to provide the mechanisms for service architects to describe the structure of the SOAP messages that a Web service supports. Once the messages of a Web service have been described, any of the currently available (or future) protocol frameworks can be used to combine the messages into protocols that expose the messaging behavior of that Web service. To that end, SSDL defines an extensible mechanism for various protocol frameworks to be used.

SSDL contracts communicate the supported messaging behavior of a Web service in terms of messages and protocols, so that architects and developers can create systems that can meaningfully participate in conversations between them. SSDL contracts may be dynamically discovered (e.g., from registries or equivalent mechanisms) and the protocol descriptions compared against an application's or workflow's requirements in order to determine whether a multimessage interaction can sensibly take place.

An SSDL contract is defined in a namespace that uniquely identifies it and consists of four major sections, as shown in Figure 15.3.

15.3.1 Schemas

The "schemas" section is used to define the structure of all the elements that will be used for the description of the SOAP messages. Any schema language may be used to define schema elements, though XML schema is the default choice.
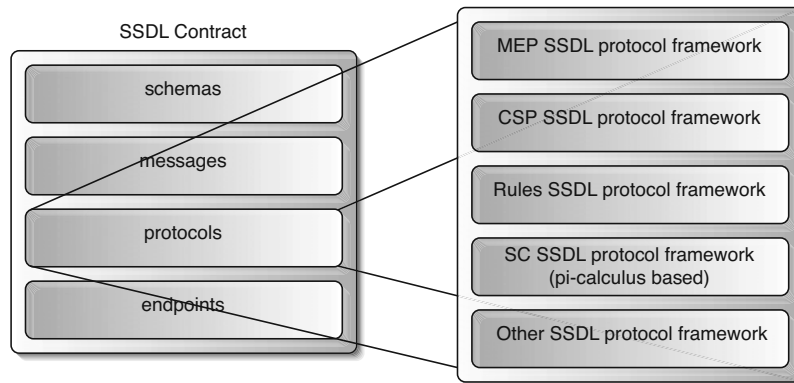
Figure 15.3: The structure of an SSDL contract.

15.3.2 Messages

The "messages" section is where the messages that a Web service supports are declared. There can be many groups of messages defined in different namespaces. However, irrespective of the namespace in which they are defined, the messages included in the SSDL document are all part of the same contract. SOAP messages are described in terms of header and body elements and are named so that protocol frameworks can reference them.

```
1   <ssdl:messages targetNamespace="uri">
2     <ssdl:message name="msg">
3       <ssdl:header ref="elements:header1" mustUnderstand="true" />
4       <ssdl:header ref="elements:header2" role="urn:ssdl:example:role"/>
5       <ssdl:body ref="elements:body1" />
6       <ssdl:body ref="elements:body2" />
7     </ssdl:message>
8
9     <ssdl:fault name="fault">
10      <ssdl:code role="http://www.w3.org/.../role/ultimateReceiver">
11        <ssdl:value>Sender</ssdl:value>
12      </ssdl:code>
13    </ssdl:fault>
14  </ssdl:messages>
```

Figure 15.4: An example of a message and a fault message.

In Figure 15.4, a message `msg` is defined to have two header elements (children of `soap:Header`) and two body (children of `soap:Body`) elements.

Note that while the SOAP processing model permits it, the WS-I Basic Profile 1.0a [35] mandates a single element as a child of `soap:Body`. However, SSDL does not enforce that restriction. Figure 15.4 also demonstrates how a SOAP fault message could be declared.

The header element provides the `mustUnderstand`, `role`, and `relay` attributes, which correspond to the equivalent attributes defined by the SOAP processing model (not all of which are shown in Figure 15.4). This makes it possible and straightforward to describe Web services infrastructure protocols.

15.3.3 Protocols and Endpoints

Once the messages in a contract have been defined, we can move on to describe how they may relate to each other. SSDL provides an extensible mechanism based on the concept of protocol frameworks.

A protocol framework uses messages declared in a contract to describe the simple message-exchange patterns or multimessage interactions that are observed by other services. A protocol framework is an XML-based model for capturing relationships between message exchanges in a workflow and may or may not be supported by an underlying formal model.

It may be possible for the same protocol to be defined in multiple ways using the same or different protocol frameworks. It is up to the designers to choose which protocol framework is best for their needs. Also, it may be possible to translate the description of a service's messaging behavior from one protocol framework to another without losing any semantics, depending on the source and target frameworks.

Some protocol frameworks may be associated with the semantics of a formal model (e.g., CSP, Rules, SC). As a result, it may be possible to use model checkers, such as SPIN [198], Failure Divergence Refinement (FDR) [143], and Mobility Workbench (MWB) [441] to verify the safety (e.g., absence of starvation and agreed termination) and liveness (e.g., eventual termination guarantee) properties of the defined protocols.

The initial release of SSDL comes with four protocol frameworks:

- The *MEP (Message Exchange Pattern) SSDL Protocol Framework* is defined to be a representation of the MEPs defined by the WSDL 2.0 specification [457]. The MEP specification defines the semantics and structure of XML elements representing several message-exchange patterns of two messages at most (excluding faults).
- The *CSP SSDL Protocol Framework* is based on the Communicating Sequential Processes [334] semantics. A protocol is defined in terms of one or more sequential processes that may communicate with each other. Messages that are sent or received represent the events in the described CSP processes [192].
- The *Rules SSDL Protocol Framework* uses preconditions on "send" and "receive" events as the means to describe messaging behaviour. As with

the CSP SSDL Protocol Framework, it is possible to use model checkers to verify that a protocol is free from deadlock and race conditions.
- The *SC (Sequential Constraints) SSDL Protocol Framework* is used to describe multiservice interactions, and its semantics are based on the π-calculus [296]. The next section of this chapter discusses this protocol framework in more detail.

An SSDL contract may also define endpoints, such as WS-Addressing Endpoint References (EPRs), of Web services that are known to support the defined contract. While the schemas, messages, and protocols of a contract (the contract is identified by its namespace) remain constant, the endpoints may change. Also, additional endpoints not defined in the contract may exist.

Note that SSDL says nothing about the scope or context of an interaction. A Web service may support one or more instantiations of a protocol at the same time. If more instantiations are supported, a contextualization mechanism is necessary for messages to be associated with a particular instantiation of the protocol (e.g., WS-Context [318], WS-Security [320], WS-Addressing [456] Reference Parameters, service-specific information, etc.).

A detailed description of the SSDL contract and the MEP, CSP, Rules, and SC SSDL Protocol Frameworks is presented in the technical specifications [247, 334, 335, 474], and a more detailed introduction to SSDL has been published in the literature [338].

15.4 The Sequential Constraint Protocol Framework

The Sequential Constraint (SC) SSDL Protocol Framework provides a machine-readable description that is used to define the protocols that a Web service supports. Such protocols may be a set of request–response interactions or could use several messages involving multiple parties over arbitrary lengths of time. The framework is intended to provide a simple way of specifying such protocols but also has a formal basis to allow properties of the protocols to be determined if required. Protocols in the framework are specified using a sequential technique, specifying the legal set of actions at each stage of the protocol. It is believed that this leads to a description that is easy to understand, as at each step of the protocol the set of actions allowed is explicitly described. The SC SSDL protocol has a formal basis in the π-calculus, a process algebra for describing mobile communicating processes. The formal basis allows multiple protocols described in the SC framework to be validated to ensure compatibility between them.

15.4.1 An Overview of π-Calculus and Its Relationship to SSDL

The π-calculus [296] is an algebra for describing and analyzing the behavior of concurrent systems. A π-calculus system is described in terms of *processes*,

channels, and *names*. Processes are independent of each other and communicate by sending messages along channels that connect them. Both channels and messages are referred to as *name* and are thus indistinguishable from each other.[1] In the following sections, we will tend to use the term *name* to refer to a message and will explicitly state when the *name* is in fact a channel rather than a message.

To send the *name msgB* along a channel named *AtoB* in π-calculus, we use the expression $\overline{AtoB} < msgB >$. A notational convention exists whereby an overbar is placed on a channel that is being used to send messages. To represent the sending of a message in SSDL-SC, we use `ssdl:msgref` with a value of "out" for the `direction` attribute, as shown on line 3 of Figure 15.5.

```
1   <sc:sequence>
2     <msgref ref="msgA" direction="in" sc:participant="serviceA"/>
3     <msgref ref="msgB" direction="out" sc:participant="serviceA"/>
4   </sc:sequence>
5
6   <sc:choice>
7     <msgref ref="msgA" direction="in" sc:participant="serviceA"/>
8     <msgref ref="msgB" direction="in" sc:participant="serviceA"/>
9   </sc:choice>
10
11  <sc:parallel>
12      <msgref ref="msgA" direction="in" sc:participant="serviceA"/>
13      <msgref ref="msgB" direction="in" sc:participant="serviceA"/>
14  </sc:parallel>
```

Figure 15.5: SSDL-SC examples.

To receive the *name msgA* down the channel named *AtoB* in π-calculus, the expression $AtoB(msgA)$ is used. Note the lack of an overbar in the channel name. Line 2 in Figure 15.5 shows the use of `ssdl:msgref` with a value of "in" for the `direction` attribute element to receive a message.

In order to define that certain things must occur in sequence, we use the period operator ".". This can be used, for example, to indicate that one message must be received before another is sent. To represent the fact that $msgA$ must be received before $msgB$ is sent, in π-calculus we would write $AtoB(msgA).\overline{BtoA} < msgB >$ and the SSDL-SC representation is shown on lines 1 to 4 of Figure 15.5.

When there is a choice of actions that can occur at a particular point in the protocol, we use the "+" operator. To test whether a particular name is received, we use the $[x = y]$ notation after the receipt of the name. For

[1] It is the fact that messages and channels are considered equivalent that allows a channel to be sent from one process to another.

instance, where either $msgA$ or $msgB$ is to be received, we would write $AtoB(msg)[msg = msgA] + [msg = msgB]$, and as shown in lines 6 to 9 of Figure 15.5. Following the test (which is known as a "match"), it is possible to execute different behaviors in a manner similar to a `switch` statement in imperative languages.

A special operator in π-calculus exists to define an internal and unobservable action, and it is represented by τ. Such τ actions should be used to describe the action of internally deciding which message to send. To either send $msgA$ or $msgB$, we would write $\tau.\overline{AtoB}<msgA>+\tau.\overline{AtoB}<msgB>$.

Sometimes it is necessary to indicate that actions may occur in parallel, which can be represented by the vertical bar operator "|". This allows us to say that $msgA$ and $msgB$ must both be received but can occur in parallel using $AtoB(msgA)|AtoB(msgB)$ or the SSDL-SC shown on lines 11 to 14 of Figure 15.5.

π-calculus expressions are built up to define named *processes* that correspond to a protocol defined in SSDL-SC; for example, the process $REQ{-}RES$ that defines the server-side view of a request–response interaction: $REQ{-}RES = AtoB(req).\overline{BtoA}<res>.REQ{-}RES$. Once it has received the request and sent the response, it invokes itself so that another request can be received. Other processes can be invoked in a similar manner in π-calculus, and the SSDL-SC element `protocol-ref` has the same semantics. Use of these primitives allows for protocol reuse in SSDL-SC.

The 0 operator in π-calculus signifies explicit termination; for instance, $P.Q.0$ means execute process P, when it completes execute process Q, and then stop. The 0 is often omitted for brevity (simply stating $P.Q$ is equivalent) but where it adds clarity or cannot be implied from the context it is included. The SSDL-SC `nothing` element has semantics similar to the 0 operator.

To define the named participants in SSDL-SC, the `participant` element is used. If the participant is annotated with the `abstract` attribute, it implies that this participant will be bound at runtime. To bind a participant during the protocol, the incoming message that contains the details of the participant is annotated using the `participant-binding-name` and `participant-binding-content` attributes. Participants who are neither specified as abstract nor implicitly bound at the beginning (the service advertising the protocol and the other initial participant) are assumed to be bound in some out-of-band method.

15.4.2 Computation in π-Calculus

Computation in π-calculus is defined by structured operational semantics, or "reaction rules" that describe how a system P can be transformed into P' in one computational step ($P \rightarrow P'$). Every computation step in π-calculus consists of communication between two terms (which may be part of separate processes or the same process). Communication may only occur between

two terms that are unguarded (that is, they are not part of a sequence prefixed by an action yet to occur) and not alternatives to each other. Consider $P = (... + x(b).Q) \,|\, (... + \overline{x}<a>.R)$ when the process is in its initial state P, two parallel processes are executing, and the latter sends the name a along the channel x. The former process receives a along channel x, as the sending and receiving terms are complementary and unguarded (said to form a "redex"). The action of receiving a has the effect of substituting a for b in the process Q, and the transformation $P \to P'$ has occurred, where $P = \{a/b\}\,Q\,|\,R$. The substitution is denoted by $\{a/b\}$ in the process P'. A side effect of this communication occurring is that the alternatives (denoted by ...) have been discarded and any communication that they would have performed has been preempted. We have now performed one computation step in the system, and the system is in a new state.

In many cases, there may be multiple states into which a process can be transformed. For example, in $P = (\overline{x}<a>.Q)\,|\,(x(b).R)\,|\,(x(c).S)$, there are two transformations possible, $P \to P$ or $P \to P'$. In the process P, name a is being sent along the channel x but can only be received by one of the two other parallel compositions. Therefore, after state P, the following states are $P' = Q\,|\,\{a/b\}\,R\,|\,(x(c).S)$, which assumes that the name a is received by the middle composition, causing a substitution of a for b in process R or $P' = Q\,|\,(x(b).R)\,|\,\{a/c\}S$, where a has been received by the other composition and is substituted for c in process S. When examining processes for compatability, each alternative transformation must be evaluated.

When considering computation in the π-calculus, the property that we are interested in proving is a lack of starvation. Starvation describes the situation where one service is expecting to receive a message that another service will never send. It should be noted that this is different from the case where a service breaks its contract by failing to send a message that its contract defines it will send. Starvation within a set of SSDL contracts results in the system becoming deadlocked: The interaction cannot progress, as an action required for progress cannot occur. In some sets of contracts, starvation may only occur under certain race conditions.

In order to validate a set of contracts, it is necessary to apply the reaction rules that were presented earlier recursively; that is, apply them to the state P' that process P has moved into following the previous computation step. When this is followed to its natural conclusion, it can be shown that the system is free of starvation conditions.

When applying the reaction rules recursively, it is necessary to show that when the receiving process is performing a match (on the incoming message), there are no messages sent that do not match one of the conditions. Also, following every reaction, one of the following holds:

1. Another reaction can occur.
2. Every process in the system is either in its initial state or a termination state where no action terms remain.

While this section has served as a very brief introduction to the π-calculus and its relationship to SSDL-SC, it has not been possible to explain all of the complexities and subtleties that would, and do, fill a book in their own right. For a further explanation of such issues, including a formal definition of the structural operational semantics of π-calculus, the reader is directed to [296].

15.5 A Use Case

We use an example from the life sciences application domain to illustrate the value of SSDL in Web services composition. In order to keep our example simple, the services are kept minimal by not exposing the complex functionality typically found in bioinformatics or Grid applications.

Figure 15.6 shows the UML message sequence diagram for three services: user interaction, bioinformatics, and data/computational. The collection of one-way messages each service supports represents the application-specific protocol in place. SSDL can be used to describe such a protocol. If we were to use WSDL to describe the message-exchange patterns of the bioinformatics service, however, we would not be able to capture the relative ordering of the messages or the interactions between multiple parties.

The application service requests a list of the supported algorithms from the bioinformatics service, chooses one, and makes a request for an analysis to start (sending all the necessary information). The bioinformatics service replies with an analysis identifier that can be used by subsequent interactions. The user-interaction service can cancel the running computation at any time. Of course, the request may be rejected for any number of reasons (e.g., because the analysis has reached a critical point or has already been completed). Appropriate messages may be sent back to the user-interaction service to represent different request rejections, but for simplicity reasons we have only included a general one.

The bioinformatics service will contact a data/computational service so that the analysis code could be executed close to an encapsulated bioinformatics database. A job identifier is returned so that the two services can correlate subsequent messages related to specific jobs. Messages containing the results are also sent from the data/computational service to the bioinformatics service and from the bioinformatics to the user-interaction service.

Due to the verbosity of the SSDL contract document, we cannot present it in its entirety here. Figure 15.7 shows part of the protocol exposed by the bioinformatics service. It is assumed that the structure of the messages (XML schema and SOAP body/header elements) and the endpoints are also defined. The protocol captures the relationship between the *AnalysisSubmission* and *AnalysisId* messages as a sequence. It then defines that if the execution is canceled at this point, the cancellation is accepted, but if the *AnalysisStarted* message is sent, then a subsequent *ExecutionCancelRequest* will be rejected. Finally, the *AnalysisCompleted* message defines the end of the protocol. For
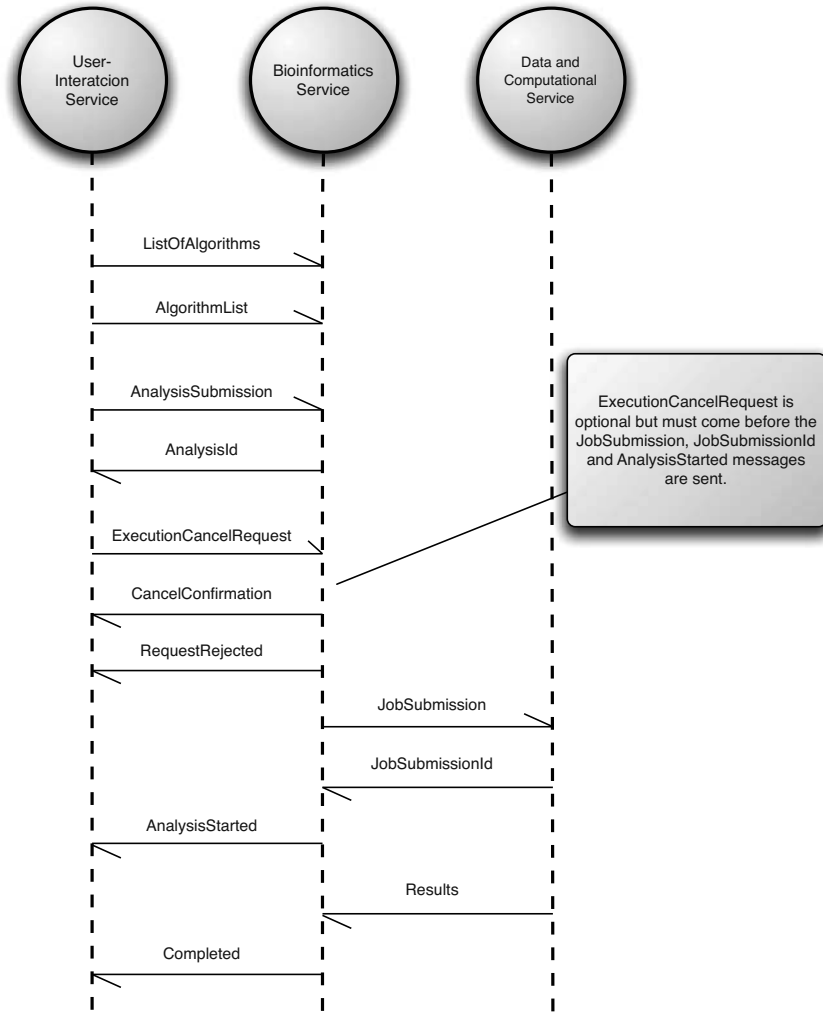
Figure 15.6: UML sequence diagram for a typical bioinformatics example.

reasons of encapsulation, the interactions with the data/computational service have not been shown in this protocol.

The protocol of Figure 15.7 can also be captured using the π-calculus notation, as shown in Figure 15.8.

In order to make the use of SSDL easy, tooling could be created to enable automatic extraction of an SSDL contract from workflow definitions. Service developers can concentrate on the implementation of their service without

```
1   <ssdl:protocol
2     targetNamespace="http://example.org/bioinformaticsService/protocol"
3     xmlns:msgs="http://example.org/bioinformaticsService/messages"
4     xmlns:sc="urn:ssdl:v1:protocol:sc">
5     <sc:sc>
6       <sc:participant name="UserInteractionService"/>
7       <sc:participant name="DataAndComputationalService"/>
8       <sc:protocol name="BioinformaticsProtocol">
9         <sc:sequence>
10          <!-- Algorithms request and response omitted -->
11          <ssdl:msgref ref="msgs:AnalysisSubmission" direction="in"
12            sc:participant="UserInteractionService"/>
13          <ssdl:msgref ref="msgs:AnalysisId" direction="out"
14            sc:participant="UserInteractionService"/>
15          <sc:choice>
16            <sc:sequence>
17              <ssdl:msgref ref="msgs:ExecutionCancelRequest" direction="in"
18                sc:particpant="UserInteractionService"/>
19              <ssdl:msgref ref="msgs:CancelConfirmation" direction="out"
20                sc:participant="UserInteractionService"/>
21            </sc:sequence>
22            <sc:sequence>
23              <ssdl:msgref ref="msgs:AnalysisStarted" direction="out"
24                sc:participant="UserInteractionService"/>
25              <sc:choice>
26                <sc:sequence>
27                  <ssdl:msgref ref="msgs:ExecutionCancelRequest" direction="in"
28                    sc:particpant="UserInteractionService"/>
29                  <ssdl:msgref ref="msgs:RequestRejected" direction="out"
30                    sc:participant="UserInteractionService"/>
31                  <ssdl:msgref ref="msgs:AnalysisCompleted" direction="out"
32                    sc:participant="UserInteractionService"/>
33                </sc:sequence>
34                <sc:sequence>
35                <ssdl:msgref ref="msgs:AnalysisCompleted" direction="out"
36                  sc:participant="UserInteractionService"/>
37                <sc:choice>
38          <sc:sequence>
39            <ssdl:msgref ref="msgs:ExecutionCancelRequest" direction="in"
40                  sc:particpant="UserInteractionService"/>
41                    <ssdl:msgref ref="msgs:RequestRejected" direction="out"
42                      sc:participant="UserInteractionService"/>
43          </sc:sequence>
44                    <sc:nothing/>
45                  </sc:choice>
46                </sc:choice>
47              </sc:sequence>
48            </sc:choice>
49          </sc:sequence>
50        </sc:protocol>
51      </sc:sc>
52  </ssdl:protocol>
```

Figure 15.7: SSDL-SC contract for the bioinformatics service.

```
1   BioService =
2   UIStoBS(asu).BStoUIS<aId>.
3     (
4     UIStoBS(ecr).BStoUIS<cc>.0
5     +
6     τ.BStoUIS<as>.
7       (
8       UIStoBS(ecr).BStoUIS<rr>.BStoUIS<com>.0
9       +
10      τ.BStoUIS<com>.
11        (
12        UIStoBS(ecr).BStoUIS<rr>.0
13        +
14        0
15        )
16      )
17    )
```

Figure 15.8: π-calculus corresponding to the bioinformatics service.

having to worry about SSDL contracts or the π-calculus syntax. As shown in Figure 15.9, workflow authors can concentrate on capturing the service logic using tools with which they are familiar. Message exchanges are represented using explicit send/receive workflow activities. The SSDL contract shown in Figure 15.7 can be generated automatically through analysis of the workflow.

A workflow definition may contain activities private to the service. The SSDL contract presented to services wishing to interact with the bioinformatics one only captures those aspects of the workflow that relate to message exchanges. It is possible to create tooling that will automatically create skeleton workflows from the bioinformatics service's SSDL contract capturing the sequence of the interactions expected. This way, the implementation of the user-interaction service can be simplified a great deal.

15.6 Related Work

In addition to WSDL, WS-BPEL 14 and WS-Choreography [458] have gained some prominence within the Web services community as candidates for describing complex Web service contracts. Both (abstract) WS-BPEL and WS-Choreography layer on top of WSDL contracts and augment those contracts with additional information pertaining to the choreography of the MEPs contained therein.

While there is merit in these approaches, there are also drawbacks. In particular, since both rely on WSDL, the level of complexity is high. The building block for the process or choreography descriptions is not the "message" abstraction, as one might have expected, but instead the "operation"
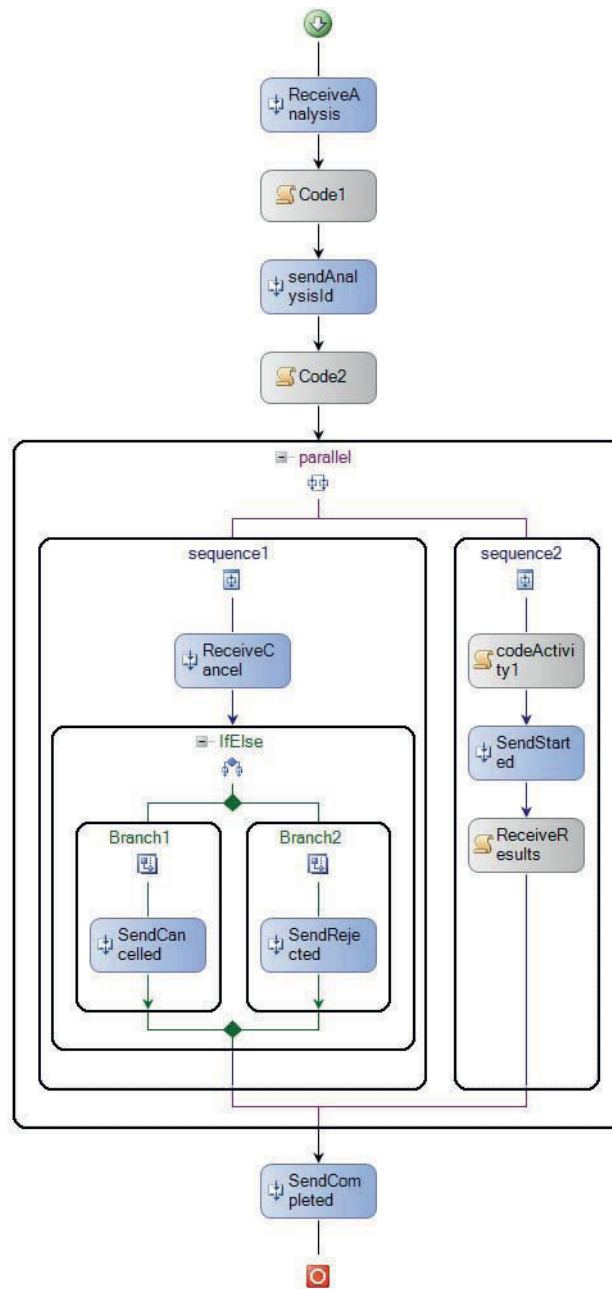
Figure 15.9: The implementation of the bioinformatics service using Microsoft's Windows Workflow Foundation [293].

abstraction. SSDL, on the other hand, allows protocols to be described directly through the correlation of messages. As a result, it should be possible to define both WS-BPEL or WS-Choreography as SSDL protocol frameworks.

15.7 Conclusions

SSDL is a contract language for describing message-oriented, asynchronous interactions between Web services. In addition to its simplicity and SOAP-centricity approach, SSDL is also able to capture rich conversations between Web services without being limited to simple request–response message-exchange patterns as is the case with WSDL.

A novel and powerful aspect of SSDL is that it enables the use of protocol description frameworks that are amenable to formal verification. While this is certainly a luxury for today's simple Web services systems, as the size and number of connected services in a deployment increases, the ability to formally verify that the system as a whole, or individual services, will not starve or race is an extremely useful proposition.

Acknowledgments

Workflow Composition: Semantic Representations for Flexible Automation

Yolanda Gil

16.1 Introduction

Many different kinds of users may need to compose scientific workflows for different purposes. This chapter focuses on the requirements and challenges of scientific workflow composition. They are motivated by our work with two particular application domains: physics-based seismic hazard analysis (Chapter 10) and data-intensive natural language processing [238]. Our research on workflow creation spans fully automated workflow generation (Chapter 23) using artificial intelligence planning techniques for assisted workflow composition [237,276] by combining semantic representations of workflow components with formal properties of correct workflows. Other projects have used similar techniques in different domains to support workflow composition through planning and automated reasoning [286,289,415] and semantic representations (Chapter 19). As workflow representations become more declarative and expressive, they enable significant improvements in automation and assistance for workflow composition and in general for managing and automating complex scientific processes. The chapter starts off motivating and describing important requirements to support the creation of workflows. Based on these requirements, we outline the approaches that we have found effective, including separating levels of abstraction in workflow descriptions, using semantic representations of workflows and their components, and supporting flexible automation through reuse and automatic completion of user specifications of partial workflows. These are all important areas in current and future research in workflow composition.

16.2 The Need for Assisted Workflow Composition

Scientific workflows typically comprise dozens of application components that process large data sets. The data sets are often sliced into smaller sets to be processed concurrently, often resulting in the execution of thousands of jobs.

Figure 16.1 shows a sketch of a partial workflow for machine translation. It illustrates how a data set is first divided into subsets, how these are processed in parallel by the same sequences of jobs, and how the final results are assembled in the final stage.
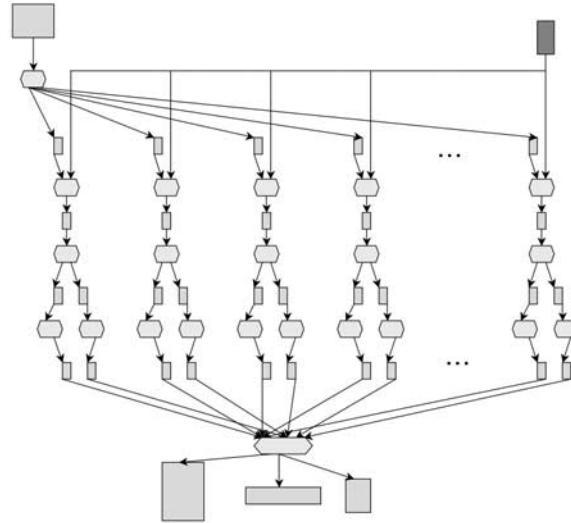
Figure 16.1: Scientific workflows may be complex and often involve parallel processing of data sets. This figure shows an example where a data set is split up in the early stages, its subsets are processed concurrently, and final results are compiled in the later stages.

16.2.1 Unassisted Workflow Composition and Its Limitations

A common approach to creating workflows is to develop ad hoc scripts that handle the iterative nature of sets of jobs and can generate workflow variants through global variables. They also specify the data locations and execution locations necessary for each job. The data have to be moved to the locations specified, and the executables must be set up in the appropriate locations. The scripts also take care of generating the metadata associated with the workflow products, often using naming conventions to differentiate among alternative configurations and executions. As an alternative to scripts, workflows may be created by hand with a text editor and updated with a copy–edit process.

These approaches have severe limitations in terms of usability and scale. Workflows can only be created by users who are very familiar with the application components used in the workflow, the execution environment, and the scripting language. Errors abound, as with any manually managed process,

and users need to be able to understand error conditions and repair failures. Extending the size of the workflows to include new models has cascading effects that have to be managed manually, making it impractical unless the additions were anticipated in advance.

Usability and scale turn out to be crucial requirements for many scientific disciplines. We motivate the requirements for scientific workflow composition with two application domains that we have used in our work and are representative of the requirements we see in other disciplines.

In order to simulate potential earthquakes, a workflow for seismic hazard analysis combines physics-based models including stress models that hypothesize the distribution of cumulated stress over fault systems given continental drift and the stress in other faults, fault rupture models that forecast potential earthquake sources in a fault system, wave propagation models that simulate the propagation of a seismic wave in a 3D Earth volume, site response models that predict how a seismic wave will be amplified at a certain location based on its soil type, and structure deformation models that simulate the effect of seismic waves in a man-made structure such as a building or a bridge. These models can be used today by the scientists who developed them, but ideally the users would include other scientists who want to use, extend, or validate the aggregate models. In addition, the models should be accessible to a wider range of users, such as engineers designing structures supposed to withstand ground motion to a reasonable degree, graduate research assistants doing advanced projects on the sensitivity of the models to certain controlled variations, and scientists in related disciplines.

Natural language researchers are developing data-intensive statistical training techniques to create language models useful for automatic summarization, machine translation, and document Indexing. A wide variety of models can be found to address different aspects of language processing, such as lexical analyzers, stemmers, part-of-speech taggers, syntax-based parsers, semantic parsers, translation rules, and so on. To put together a machine translation system requires assembling an entire suite of such models to process and parse the original language sentence, map it to the target language, and smooth out the output to make it as fluent as possible. Each of the models has to be trained, perhaps on a different body of text, depending on the topic of translation, before the actual translation is done on the original sentence. New models are developed constantly by different groups around the world, and variations of combinations of models are explored by different research groups for different purposes. Because better performance is invariably obtained with larger sets of training data, there is increased interest in workflow environments that exploit high-end computing and large data and storage management facilities. Sharing of data and models is often done informally across research groups. Flexible workflow composition and execution frameworks would support the rapid development and validation of novel approaches and models.

In summary, although it is possible to create and manage workflows of considerable size in an unassisted manner, there are severe challenges and

practical limitations in terms of usability and scalability that can only be addressed by end-to-end workflow systems that assist users with the creation, execution, and management of workflows.

16.2.2 Workflow Composition Scenarios

The following are representative scenarios for workflow composition illustrated in these two application domains. These scenarios motivate the requirements for workflow composition discussed in the next subsection.

Running a Common Kind of Analysis with a New Data Set

A common kind of wave propagation simulation takes a fault rupture and a model of the corresponding Earth volume's characteristics and runs a physics-based anelastic wave propagation model over that volume to generate 2D or 3D seismograms. This kind of analysis is done routinely by Southern California Earthquake Center (SCEC) scientists well versed in such physics-based wave propagation models, but a scientist in Seattle may want to apply the same analysis to data for the Pacific Northwest area. The workflow structure is essentially the same, but the input data to be used are different. The Seattle scientist will not be able to compose the workflow from scratch but could reuse the basic workflow structure. In a machine translation project, the same workflow can be tried out with a new body of text or a new language.

Creating a Variant of a Type of Analysis

A scientist in Santa Barbara who creates in her research a new model of a fault in Southern California would want to test this model with typical wave propagation simulation codes, except replacing the usual Earth volume model by one that incorporates hers. In this case, the scientist from Santa Barbara does not need to compose a workflow from scratch but instead could reuse the commonly used workflow and modify it slightly by substituting one of the components. In a machine translation project, a scientist may try out a new parser her group has developed and investigate its effect on the final translation quality.

Specifying only Critical Aspects of the Analysis

A scientist in Boston may be interested in simulating wave propagation using finite-difference models, but any of the finite-difference models would be acceptable. This illustrates that it is possible to describe categories of workflows based on abstract classes of models. The new workflow would not be composed from scratch, but by selecting one of the instances of the abstract class of models mentioned in the workflow. A machine translation researcher working on improving the fluency of the output will run workflows with a part-of-speech tagger but may have no preference regarding the kind used.

Running a Complex Analysis Composed of Common, Simpler Ones

An engineer in Palo Alto would like to simulate the effect of certain fault ruptures on his design of a freeway overpass at a location close to the San Andreas fault. This may require composing a workflow by combining two workflows: one designed to simulate the effect of certain ground motions on the overpass structure and another one designed to simulate the wave propagation from the fault ruptures to the site of the overpass. In a machine translation project, the output of translation may be used for document summarization, where the overall processing would be obtained by combining the two respective workflows.

Specifying a New Type of Analysis

A scientist may create a new model for wave propagation that runs very efficiently if coupled with certain types of models of an Earth volume. This scientist would have to compose a completely new workflow out of a new set of models by specifying step by step what models are to be used and how they need to be combined. A machine translation researcher may create models that represent a new approach to word-by-word translation and use them to create a new kind of workflow.

16.2.3 Requirements for Workflow Composition

The scenarios above illustrate that workflows have many users and uses that need assistance in creating workflows. From graduate students to experienced scientists, scientists with varied needs and expertise may need to conduct workflow analyses using the same underlying models and data. Engineers or scientists in other disciplines may benefit from using the same models if they are made accessible and easy to use within their own analysis process. The degree of freedom and the amount of assistance and automation required during workflow creation will be very different in each case. But ideally the same underlying mechanisms should be used to manage the workflow composition process.

Some of the scenarios above describe scientific exploration tasks. In those cases, the scientist will always want to specify some aspects of the analysis that are critical to their investigation, leaving it to the system to figure out the rest automatically. The initial specification may include partial descriptions of desired results, application components to be used in the analysis, input data to be used in the computation, or all of the above. This requires an expressive language that supports flexible descriptions of models and data. This may require assisting users to provide a complete and valid initial specification to ensure that all the pieces provided are mutually consistent and that it is possible to create a full workflow from them. Once the initial user specification is provided, it can then be automatically extended to form a complete workflow

that can be executed. This requires a flexible workflow completion mechanism since it will need to work from results back to what is required to generate them, from input data down to typical ways to process them, or from models and their requirements that need to be generated by adding other models to the workflow and so on.

Most of the scenarios above do not require creating workflows from scratch. Although in some cases step-by-step assembly of new workflows from individual components is needed, workflows can often be created by reusing existing workflows with minimal adaptations. This is not surprising, given that scientific exploration often involves repeated analysis with small variants or local modifications. Workflow reuse also encourages the practice of well-established methodologies captured in particular workflow specifications. The more common steps in the workflows used by different scientists to do similar analyses, the more comparable their results will be. This argues for reusing workflow structures as much as possible across research groups and across experiments or analyses. Results that are obtained using well-established methodologies should essentially be the products of well-known and easily identifiable workflows.

Workflow reuse involves two major aspects: retrieval and adaptation. Retrieval involves finding appropriate workflows in a library, which requires that workflow repositories be organized and indexed thematically and hierarchically. Adaptation of workflows has a wide range of complexity. The less sophisticated a user is, the more he or she is likely to reuse entire workflow structures. More advanced users will be familiar with details of the models and may venture to create variants of a previous workflow by adding or replacing components. The simplest kinds of adaptation involve simple substitutions of input data. The workflow composition system should ensure that the new data set is appropriate for the models included in the workflow. This requires that each workflow be described in terms of the types of data for which it is appropriate. More complex kinds of reuse involve substitutions of specific components together with the addition of steps to generate the data needed by the new components. Other steps needed by the old model may no longer be necessary and need to be removed. Supporting this adaptation process requires representing the characteristics and constraints of each model and the ability to use those representations to check the overall consistency of workflows. Checking the consistency and validity of workflows is even more necessary when several existing workflows at a time are reused to create a new workflow.

These scenarios also illustrate the need for describing workflows in terms that are critical for the experiment while ignoring necessary but irrelevant execution details. This necessary detail is needed to execute the workflow and includes components that prepare and transform the data into formats required by the models, move data to the locations where they need to be processed or stored, and perform conversions to different metric or reference systems. A workflow composition system should present workflows to users

at an appropriate level of abstraction. In addition, it should automatically manage any steps in the workflow that do not involve experiment-critical components.

In summary, there are three key requirements for assisting users in workflow composition. First, workflows must be described at different levels of abstraction that support varying degrees of reuse and adaptation. Second, expressive descriptions of workflow components are needed to enable workflow systems to reason about how alternative components are related, the data requirements and products for each component, and any interacting constraints among them. Third, flexible workflow composition approaches are needed that accept partial workflow specifications from users and automatically complete them into executable workflows. The next three sections discuss each of these three requirements in turn.

16.3 From Reusable Templates to Fully Specified Executable Workflows

Representing workflows at appropriate levels of abstraction is key to support reuse and to manage the complexity and details involved in creating scientific workflows. In our work we consider three stages of creation of workflows, illustrated in Figures 16.2, 16.3, and 16.4. Each stage corresponds to a different type of information being added to the workflow, namely:

1. Defining *workflow templates* that are data- and execution-independent specifications of computations. Workflow templates identify the types of components to be invoked and the data flow among them. The nature of the components constrains the type of data that the workflow is designed to process, but the specific data to be used are not described in the template. In this sense, a workflow template is parameterized where its variables are data holders that will be bound to specific data in later stages of the workflow creation process. A workflow template should be shared and reused among users performing the same type of analysis.
2. Creating *workflow instances* that are execution-independent. Workflow instances specify the input data needed for an analysis in addition to the application components to be used and the data flow among them. A workflow instance can be created by selecting a workflow template that describes the desired type of analysis and binding its data descriptions to specific data to be used. While a workflow instance logically identifies the full analysis, it does not include execution details such as the physical replicas or locations to be used. That is, the same workflow instance can be mapped into different executable workflows that generate exactly the same results but use different resources available in alternative execution environments.

3. Creating *executable workflows*. Executable workflows are created by taking workflow instances and assigning actual resources that exist in the execution environment and reassigning them dynamically as the execution unfolds. Executable workflows fully specify the resources available in the execution environment (e.g., physical replicas, sites and hosts, and service instances) that should be used for execution. This mapping process can be automated and ideally is incremental and dynamic. In an incremental mapping scheme, only the initial workflow steps might be assigned to resources, while later steps can wait until the execution of the initial steps is finalized. The mapping should be dynamic so that when an execution failure occurs, the assignment can be reconsidered.

The template shown in Figure 16.2 depicts a rule-pruning workflow for machine translation. This template corresponds to the workflow shown in Figure 16.1. Templates should specify the types of input data that they are able to process. In this case, the template takes as input a plain text corpus and a roster of kernel rules (generated by a different workflow). An instance can be created simply by binding these two inputs to specific data. In Figure 16.3, the input is specified as WSJ-2001 and KR-09-05. The specification of the workflow instance can be quite compact since it consists of a template identifier and a set of bindings for its inputs. This compact specification can be turned into a fully expanded instance, shown on the right-hand side of Figure 16.3 and corresponding to the workflow in Figure 16.1. Executable workflows fully specify what needs to be executed, where, and in what order. They also include data movement steps as required by the computations. In the example shown in the figure, the initial and final stages may be performed in local machines, while the most computationally intensive stages could be executed in a shared resource (e.g., a cluster). The final results as well as some intermediate results may be recorded in shared data repositories. An executable workflow, shown in Figure 16.4, can be automatically generated from the workflow instance by analyzing the execution requirements of each step and mapping them into the particular data storage and computation resources available at the time of execution.

While these different stages make useful distinctions, they are not meant to be rigid. For example, a workflow template may be partially instantiated in that some of its data type placeholders may already be assigned to existing data. A partial instantiation can be used to specify an analysis of a data set against a standard invariant data set. It could also be used to specify parameter settings for some of the components of the analysis. The workflow creation process needs to be flexible across these stages.

Reuse is greatly facilitated by distinguishing these different levels of abstraction, from generic reusable workflow templates, to specific data-processing workflow instances, to execution-dependent workflows. Users can simply retrieve templates that are appropriate to their needs and specify the
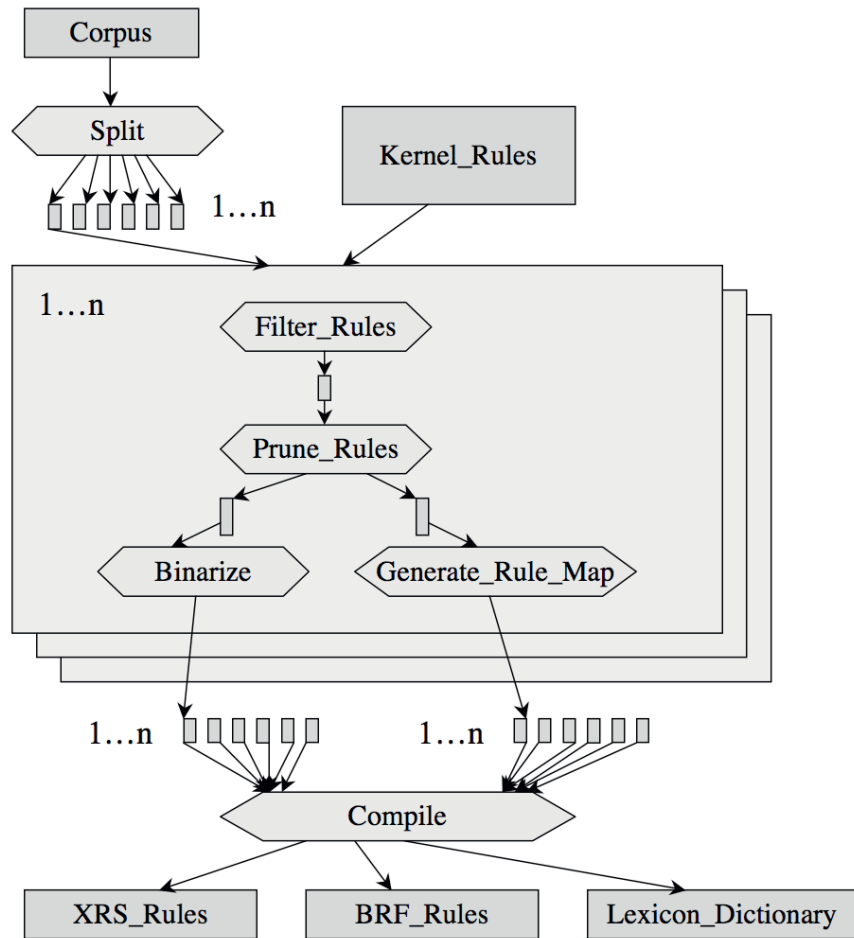
Figure 16.2: A workflow template captures the structure of the workflow in a data-independent and execution-independent representation.

data to be processed. New types of workflows can be created by adapting or merging existing templates.

Validation is also facilitated through this three-stage process. Workflow templates can specify constraints on the types of input data that they can process. In creating workflow instances, users can be required (and guided!) to provide data that satisfy those constraints.
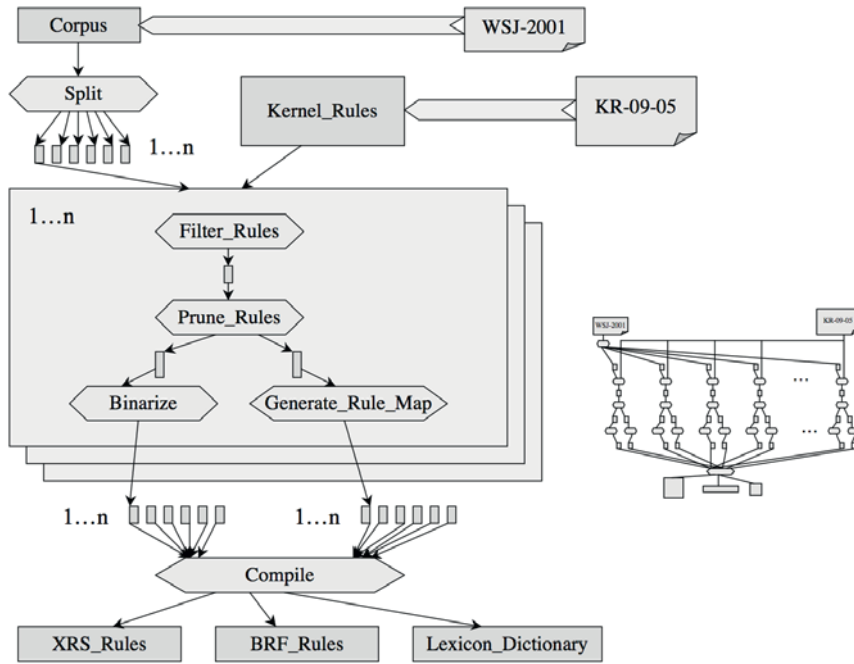
Figure 16.3: A workflow instance specifies the data to be processed by a workflow template in an execution-independent representation.
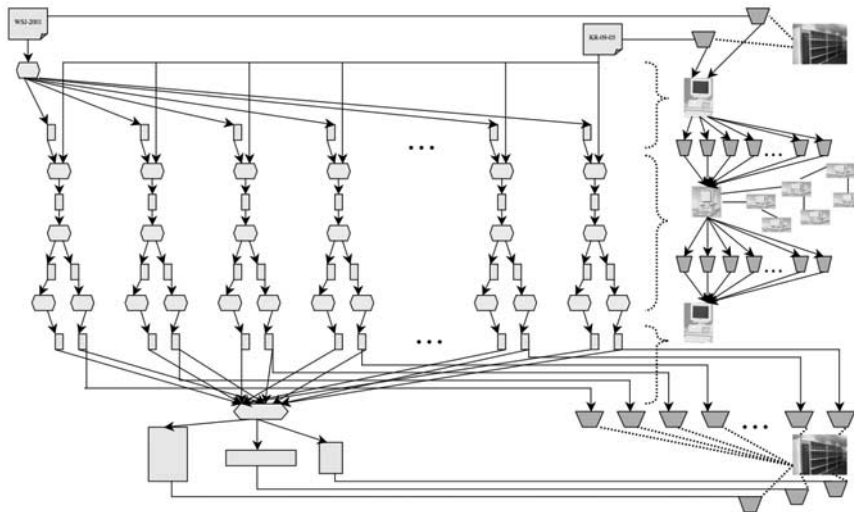
Figure 16.4: An executable workflow specifies where the data are stored and processed and may include additional steps to move data as required for computation and storage.

16.4 Semantic Representations of Workflows to Support Assisted Composition

Figure 16.5 shows an overview of the kinds of ontologies that we use to represent workflows. We use the Web Ontology Language (OWL) [331], a W3C recommendation that now has several widely available editors and reasoners for efficient inference. Application-specific ontologies, shown in the middle of Figure 16.5, include definitions of terms and relations that are useful to describe different data properties in the domain at hand. These domain terms can be organized in classes described in terms of their relations to other classes as well as by their class-specific properties. For example, a class of data objects defined for the machine translation workflow is "Kernel_Rules." This can be a subclass of a more general class, "Translation_Rules." These domain terms are then used to define the classes of data required for each component, as well as the data they create. In our example, the definition of the component "Filter_Rules" would state that one of its inputs is a file of type "Kernel_Rules." Components can be organized by component types, also organized in classes. For example, a generic class "Process_Rules" may be defined as having at least one input that is of type "Translation_Rules." Given this definition, the component "Filter_Rules" belongs to that more general class. Workflow templates and instances can be specified by using these component classes and descriptions. Since the output of "Filter_Rules" is specified to be a set of "Translation_Rules," the file that results from the second component of the template is of that type. All these application-specific ontologies can be specializations of application-independent definitions, shown at the top of Figure 16.5. These generic ontologies can describe the relationships between components, data, and workflows, as well as their properties. At the bottom of the figure, external catalogs can be used as repositories of data, components, and executed workflows. All these catalogs can be indexed by the ontologies, where any metadata attributes would correspond to terms defined in the ontologies.

Semantic workflow representations support workflow composition in several ways. First, the ontology definitions of classes and properties can be used to check that newly created workflows are valid. In our running example, a workflow instance may be noted to be invalid if KR-09-05 is not recognized to be an object of type "Kernel_Rules," which may be either directly stated in or deduced from the metadata attributes of KR-09-05. The consistency of newly created workflow templates can also be checked against the definitions in the ontologies. The second use of semantic workflow representations is for retrieval from workflow libraries. Using ontology-based query languages, a workflow template can be retrieved by providing a description of the features sought. For example, one may search for workflows that take "Kernel_Rules" and include at least one component of type "Process_Rules." With this description, the template in Figure 16.2 would be found.

Semantic workflow representations allow a better integration of data management systems and workflow systems by supporting detailed descriptions of
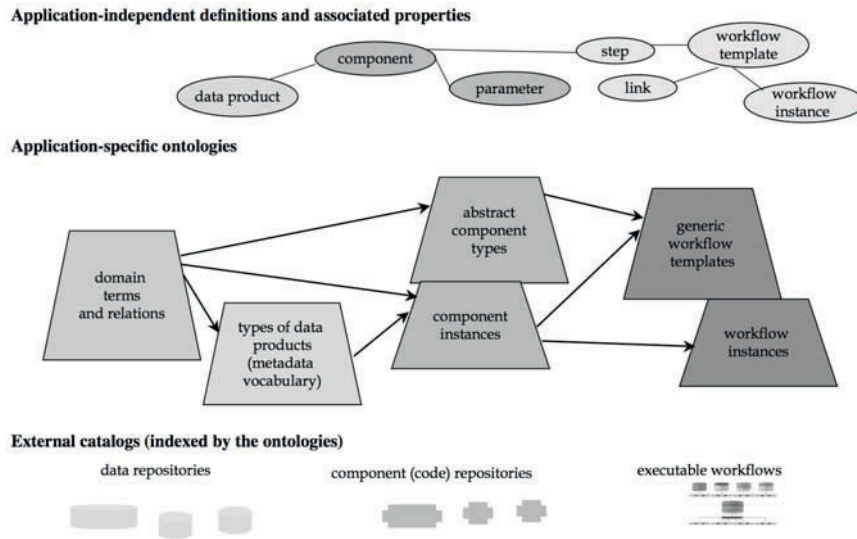
Figure 16.5: Semantic workflow representations include application-specific ontologies organized as specializations of generic application-independent definitions. The ontologies can be used to index repositories of data, components, and executed workflows.

the requirements of a given workflow. Data can be described in a workflow using metadata attributes to more or less detail. A workflow specification could include intensional descriptions that describe the data required or extensional descriptions that refer to specific data sets to be used. These descriptions could be used to retrieve the data by matching the description against a catalog. The descriptions of the data could also prompt the automatic assembly or generation of the data, perhaps by firing off another workflow. Note that these representations can describe the data required or produced by a workflow at different levels of abstraction, including regarding its format and storage. The same data can be stored in alternative formats, such as a database, or perhaps in a set of files, each structured either as a table, an XML structure, or a labeled list. The same data may be replicated in different locations, perhaps with alternative file breakdowns or directory structures. In some cases, data formats have a major influence on the efficiency of the workflow, whereas in other cases data formats do not affect the logical analysis in the workflow and their handling and conversions may be completely abstracted away.

Rich metadata descriptions of intermediate or final workflow results can be automatically created based on the representations of the template and instance that generated them. This supports an important requirement of scientific domains in terms of documenting how any data are generated and

what their properties are. It also supports automatic workflow completion, as we explain in the next section.

In summary, semantic workflow representations can support workflow composition in several ways. The reasoners can use the ontologies and associated definitions to ensure the consistency of user-created workflows. During workflow creation and retrieval, ontology-based query languages can be used to find relevant workflows and components based on their properties.

16.5 Automatic Completion of Workflows

In our work, we have used search and planning algorithms from artificial intelligence to design workflow completion and automatic generation algorithms. The abstraction levels and the semantic representations discussed so far turn out to be useful in supporting a more flexible framework for automatic completion of workflows.

Complete automation is desirable in the stage of creating an executable workflow from a workflow instance. This is possible when the execution requirements of the workflow components are specified, and the system can query the execution environment to find what resources are available for execution. Important challenges include optimizing the completion time of any given workflow, considering resource assignment trade-offs across many workflows, and designing appropriate failure handling and recovery mechanisms.

Automation can also be used to complete underspecified workflow templates. Workflow templates are underspecified when they include abstract computation descriptions to be specialized during workflow instance creation. For example, a step in a workflow template may specify a class of component such as "Gridding." The specific gridding component to be used may depend on the nature of the data processed by the workflow. Workflow templates can also be underspecified in that they may be missing workflow components that perform data formatting, conversions, and other steps that are not considered critical to the analysis done in the workflow. As we mentioned, these are necessary ingredients of a workflow, yet the details of how these steps are performed may be irrelevant to the experimental design and to the scientist. Automatically adding these steps is possible when the format requirements for experiment-critical components are declaratively specified and when the component library includes appropriate components for doing the kinds of data processing required. The kinds of data processing needed may not be known until the data are specified and therefore would not typically be included in a workflow template. Once input data are specified, new data-processing steps can be added during workflow instance creation. Intermediate data products may also need to be converted for consumption of a subsequent step. In some cases, their format can be anticipated from the workflow template definitions and the new steps can be added during workflow instance creation. However, in other cases, the format of intermediate data products will only be known

once they are created during execution, and in those cases the insertion of data-processing steps will need to be interleaved with the execution process.

Full automation of the workflow composition process may be desirable for some kinds of workflows and application domains. Given a description of the desired data products, the task of creating a valid workflow from individual application components can require full-fledged automatic programming capabilities. Automatic workflow generation is manageable in domains where application components can be clearly encapsulated and described with detailed specifications of the component's outputs based on the properties of their input data. These specifications must include criteria for component selection and data selection when several alternatives are appropriate and where the quality of the workflow results may depend on mutually constraining choices. As an alternative to creating new workflows from scratch, fully automatic workflow generation can also be achieved by reusing workflow templates. This approach requires a library of workflow templates that reflects common analysis processes in the domain and is appropriately indexed according to the requirements that will be provided at workflow generation time, be they the desired results, the input data to be analyzed, or the types of components in the workflow.

16.6 Conclusions

The scale and complexity of scientific applications challenge current workflow representations and pose limitations on simple workflow composition tools such as graphical editors or authoring environments with limited user assistance. We have argued that workflow composition environments can greatly benefit from (1) semantic representations of workflows and their components, (2) workflow representations at different levels of abstraction, and (3) flexible automation in completing user-provided partial workflow descriptions. These novel capabilities can have broader implications beyond workflow composition in terms of increased automation and intelligent capabilities for workflow management and execution.

Acknowledgments

Virtual Data Language: A Typed Workflow Notation for Diversely Structured Scientific Data

Yong Zhao, Michael Wilde, and Ian Foster

17.1 Introduction

When constructing workflows that operate on large and complex data sets, the ability to describe the types of both data sets and workflow procedures can be invaluable, enabling discovery of data sets and procedures, type checking and composition of procedure calls, and iteration over composite data sets.

Such typing should in principle be straightforward because of the hierarchical structure of most scientific data sets. For example, in the functional magnetic resonance imaging (fMRI) applications in cognitive neuroscience research that we use for illustrative purposes in this chapter, we find a hierarchical structure of studies, groups, subjects, experimental runs, and images. A typical application might build a new study by applying a program to each image in each run for each subject in each group in a study.

Unfortunately, we find that such clean logical structures are typically represented in terms of messy physical constructs (e.g., metadata encoded in directory and file names) employed in ad hoc ways. For example, the fMRI physical representation with which we work here is a deeply nested directory structure, with ultimately a single 3D image ("volume") represented by two files located in the same directory, distinguished only by filename suffix. The members of a data set are typically distinguished by identifiers embedded in filenames using diverse, ad hoc conventions.

Such nonuniform physical representations make program development, composition, and execution unnecessarily difficult. While we can incorporate knowledge of file system layouts and file formats into application programs and scripts, the resulting code is hard to write and read, cannot easily be adapted to different representations, and is not clearly typed.

We have previously proposed that these concerns be addressed by separating abstract structure and physical representation [149]. (Woolf et al. [477] have recently proposed similar ideas.) We describe here the design, implementation, and evaluation of a notation that achieves this separation.

We call this notation a *virtual data language* (VDL) because its declarative structure allows data sets to be defined prior to their generation and without regard to their location and representation. For example, consider a VDL procedure "foo_run" with the signature "Run Y=foo_run(Run X)" and with an implementation that builds and returns a new run Y by applying a program "foo" to each image in the run supplied as argument X (X and Y being data set variables of type Run). We can then specify via the VDL procedure invocation "run2=foo_run(run1)" that data set "run2" is to be derived from data set "run1." Independence from location and representation is achieved via the use of XML Data Set Typing and Mapping (XDTM) [303] mechanisms, which allow the types of data sets and procedures to be defined abstractly in terms of XML schema. Separate *mapping descriptors* then define how such abstract data structures translate to physical representations. Such descriptors specify, for example, how to access the physical files associated with "run1" and "run2."

VDL's declarative and typed structure makes it easy to build up increasingly powerful procedures by composition. For example, a procedure "Subject Y = foo_subject(Subject X)" might apply the procedure "foo_run" introduced earlier to each run in a supplied subject. The repeated application of such compositional forms can ultimately define large directed acyclic graphs (DAGs) comprising thousands or even millions of calls to "atomic transformations," each of which operates on just one or two image files.

The expansion of data set definitions expressed in VDL into DAGs, and the execution of these DAGs as workflows in uni- or multiprocessor environments, is the task of an underlying *virtual data system* (VDS) [148], which is comprised of workflow translators, planners, and interfaces to enactment engines.

We have applied our techniques to fMRI data analysis problems [439]. We have modeled a variety of data set types (and their corresponding physical representations) and constructed and executed numerous computational procedures and workflows that operate on those data sets. Quantitative studies of code size in a previous paper [496] suggest that VDL and VDS facilitate easier workflow expression and hence may improve productivity.

This chapter describes:

1. the design of a practical workflow notation and system that separate logical and physical representation to allow the construction of complex workflows on messy data using cleanly typed computational procedures;
2. the VDL type system, as well as the interfaces for mapping specification and program invocation; and
3. a demonstration and evaluation of a prototype of the technology via the encoding and execution of large fMRI workflows in a distributed environment.

This chapter is organized as follows. In Section 17.2, we review related work. In Section 17.3, we introduce the XDTM model, and in Sections 17.4 and

17.5 we describe VDL, using a simplified science application for illustration. In Section 17.6, we apply this model to a real example drawn from procedures used to prepare fMRI study data for analysis. In Section 17.7, we describe our prototype implementation, and in Section 17.8 we conclude with an assessment of this approach.

17.2 Related Work

The Data Format Description Language (DFDL) [42], like XDTM, uses XML schema to describe abstract data models that specify data structures independent from their physical representations. DFDL is concerned with describing legacy data files and complex binary formats, while XDTM focuses on describing data that span files and directories. Thus, the two systems can potentially be used together.

In MIX (Mediation of Information using XML) [40], each data source is also treated as an XML source, and its structural information is represented by an XML DTD. Queries are expressed in a high-level declarative XML query language called XMAS (XML Matching and Structuring Language), which allows object fusion and pattern matching and supports construction of new integrated XML objects from existing ones. MIX's query evaluation takes a virtual approach, where XML queries expressed in XMAS are unfolded and rewritten at runtime and sent to corresponding sources.

The IBM virtual XML garden project [208] provides virtual XML views on diverse data sources such as file systems, zip archives, and databases. It supports XML access and processing on these data sources by writing thin, on-demand adapters that wrap arbitrary data structures into a generic abstract XML interface corresponding to the XML Infoset as well as the XPath and XQuery Data Model.

XML Process Description Language (XPDL) [485], BPEL, and WSDL also use XML schema to describe data or message types but assume that data are represented in XML; in contrast, XDTM can describe "messy" real-world data by mapping from a logical XML view to arbitrary physical representations. Ptolemy [130] and Kepler [19] provide a static typing system; Taverna [326] and Triana [91] do not mandate typing. XDTM's ability to map logical types from/to physical representations is not provided by these languages and systems.

When composing programs into workflows, we must often convert logical types and/or physical representations to make data accessible to downstream programs. XPDL employs scripting languages such as JavaScript to select subcomponents of a data type, and BPEL uses XPath expressions in *Assign* statements for data conversion. VDL permits the declarative specification of a rich set of mapping operations on composite data structures and substructures.

Many workflow languages allow sequential, parallel, and recursive patterns but do not directly support iteration. Taverna relies on its workflow engine to run a process multiple times when a collection is passed to a singleton-argument process. Kepler uses a "map" operator to apply a function that operates on singletons to collections. VDL's typing supports flexible iteration over data sets — and also type checking, composition, and selection.

17.3 XDTM Overview

XDTM defines a data set's *logical structure* via a subset of XML schema, which defines primitive scalar data types, such as Boolean, integer, string, float, and date, and also allows for the definition of complex types via the composition of simple and complex types.

A data set's *physical representation* is defined by a *mapping descriptor*, which defines how each element in the data set's logical schema is stored in, and fetched from, physical structures such as directories, files, and database tables. The original XDTM description [303] indicated that a mapping descriptor groups together a set of mapping functions, each associated with an XML schema type, but did not specify exactly how these mapping functions would be defined. In this chapter, we describe an approach to defining and applying these mapping functions.

In order to permit reuse for different data sets, mapping functions may be parameterized for such things as data set locations. Thus, in order to access a data set, we need to know three things: its type schema, its mapping descriptor, and the value(s) of any parameter(s) associated with the mapping descriptor. These three components are grouped to form a *data set handle*.

Note that multiple mappings may be defined for the same logical schema (i.e., for a single logical type). For example, an array of numbers might be physically represented, in different contexts, as a set of relations, a text file, a spreadsheet, or an XML document.

17.4 Physical and Logical Structure: An Example

We use a simple example to illustrate the relationship between physical and logical structure. This example concerns the analysis of data collected from portable cosmic ray detectors. Such detectors are increasingly used in secondary-level physics education through projects such as QuarkNet [36].

As in many scientific experiments, the nature of the data collection process determines the data's physical representation. Students are organized into groups; each group installs a few detectors and collects data from the detectors. Data from detectors are sent to PCs in the form of simple text files that describe the sampling of A/D converter levels on the multiple channels of the instrument (Figure 17.1). (We can think of these "raw data files"

as describing potential cosmic ray events in the form of digitized waveform descriptions.) Analysis then consists of processing these raw waveforms to eliminate noise, extracting a signal, and then searching for correlations in the data from multiple channels, multiple instruments at varying locations, and multiple runs.
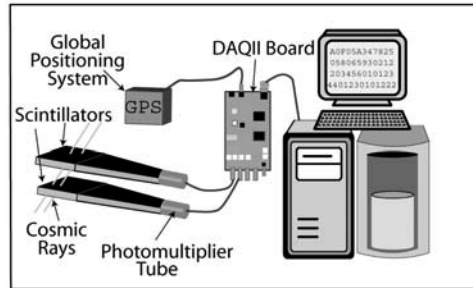
Figure 17.1: Cosmic ray detector.

As depicted in Figure 17.2, a suitable physical data set organization for this application is a hierarchical directory structure that provides for multiple experimental groups, each with data from one or more *detectors*. (Note that directories are distinguished from files by a trailing "/".) Observations consist of *raw data* from the instruments along with metadata about the time period of the recording and the physical location and orientation of the detector ("*geometry*"). One metadata file per detector ("*detector_info*") is also present in the structure. Derived data produced via various data analysis procedures are stored in the same structure. *Pulse* files are an example of an output data set added to the observation following the application of a reconstruction procedure to the raw events.

To illustrate how "messy" a physical representation can be, consider that in this application we could represent the start date/time of an observation using the creation time of the *rawdata* file and the end time of the observation by the modification time of that file.

In contrast to these ad hoc physical encodings, the *logical* structure of such a physical data set representation can be uniformly and explicitly described by XML schema, as illustrated in Figure 17.3.

17.5 Virtual Data Language

XDTM specifies how we define XML structures and associate physical representations with them. However, it does not address how we write programs that operate on XDTM-defined data. That is the focus of the

XDTM-based Virtual Data Language (VDL). This language, derived loosely from an earlier VDL [148] that dealt solely with untyped files, allows users to define procedures that accept, return, and operate on data sets with type, representation, and location defined by XDTM. We introduce the principal features of VDL via a tutorial example.

17.5.1 Representing Logical Structure: The VDL Type System

VDL uses a C-like syntax to represent XML schema complex types, as illustrated in Figure 17.4, which shows VDL type definitions corresponding to the XML schema of Figure 17.3. The *Detector* type contains information about the detector hardware — such as serial number, installation date, and firmware revision (*DetectorInfo*) — and a set of *Observations*. Each *Observation* contains the time range for which the raw data are gathered (*ostart*, *oend*), the raw data themselves, some geometry information, and a derived data type *Pulse*. The conversion from this notation to XML schema is straightforward: The VDL data model of named member fields ("structures" or "records") and arrays is mapped to XML schema constructs for sequences and element cardinality (occurrence).

17.5.2 Accessing Physical Structure: Mapping Functions

The process of *mapping*, as defined by XDTM, converts between a data set's physical representation (typically in persistent storage) and a logical XML view of those data. VDL programs operate on this logical view, and *mapping functions* implement the actions used to convert back and forth between the XML view and the physical representation.

Associated with each logical type is a mapping descriptor, which provides access to a set of mapping functions that the VDL implementation may invoke during program execution. A mapping descriptor must include the following four functions:

```
/quarknet/
/quarknet/group1/
/quarknet/group1/detector1/
/quarknet/group1/detector1/detector_info
/quarknet/group1/detector1/observation1/
/quarknet/group1/detector1/observation1/geometry
/quarknet/group1/detector1/observation1/rawdata
/quarknet/group1/detector1/observation1/pulse
/quarkent/group1/detector1/observation2/
...
/quarknet/group1/detector2/
...
```

Figure 17.2: Physical directory structure of detector data.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://quarknet.org/schema/cosmic.xsd"
           xmlns="http://quarknet.org/schema/cosmic.xsd"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:complexType name="Observation">
    <xs:sequence>
      <xs:element name="ostart" type="xs:date"/>
      <xs:element name="oend" type="xs:date"/>
      <xs:element name="rawdata" type="RawData"/>
      <xs:element name="geo" type="Geometry"/>
      <xs:element name="pulse" type="Pulse"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="Detector">
    <xs:sequence>
      <xs:element name="info" type="DetectorInfo"/>
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
        <xs:element name="ob" type="Observation"/>
      </xs:sequence>
    </xs:sequence>
  </xs:complexType>

</xs:schema>
```

Figure 17.3: XML schema of detector data.

```
type Detector {
    DetectorInfo info;
    Observation ob[ ];
}

type Observation {
    Date ostart, oend;
    RawData rawdata;
    Geometry geo;
    Pulse pulse; /* a derived file: pulses reconstructed from raw */
}

type DetectorInfo {
    Int serialNum;
    Date installDate;
    String swRev;
}
```

Figure 17.4: VDL type definition for detector data.

- create data set: creates a physical data set conforming to the desired physical representation;
- store member: stores a specific element of the logical structure into its physical storage;
- get member: gets a specific logical child element from the data set's physical storage;
- get member list: gets a list of child elements from the physical storage.

In addition, a mapping descriptor often includes additional mapping functions that provide access to various components of the physical data representation. For example:

- filename: provides access to the file or directory name of a data set member's physical representation.

To realize the mapping model in VDL, we formalize the concept of the XDTM logical XML view by defining a construct much like an XML store [435], which we call the *xview*. The xview is managed by the VDL runtime implementation, which we refer to abstractly as the *virtual data machine*, or VDM.

As a VDL program executes, the VDM performs VDL expression evaluation by invoking the appropriate mapping functions to move data back and forth between physical data representations and the xview. When a mapping function maps a data set's representation into the xview, it creates the XML representation of the physical data structure, in which each field (or member) of a data set type becomes either an atomic value or a handle to the corresponding physical data set component. VDL variables that are defined as local to a VDL procedure (i.e., "stack variables" in a procedure's activation record [363]) are represented in a similar manner.

The xview can be implemented in many ways (for instance, in an XML database) and has the desirable features that (a) it can be processed by standard XML tools such as XPath and XQuery and (b) it can operate as a cache, logically representing an entire physical data set but physically "faulting in" data sets as they are referenced in a "lazy evaluation" mode and swapping out data sets that are not currently being referenced on a least recently used basis.

17.5.3 Procedures

Data Sets are operated on by *procedures*, which take data sets described by XDTM as input, perform computations on those data sets, and produce data sets described by XDTM as output.

A VDL procedure can be either an *atomic procedure* or a named workflow template defining a DAG of multiple nodes (*compound procedure*). An atomic procedure definition specifies an interface to an executable program or service (more on this topic below), while a compound procedure composes calls to atomic procedures, other compound procedures, and/or control statements.

VDL *procedure definitions* specify formal parameters; procedure calls supply the actual argument values. For example, consider an atomic procedure *reconstruct* that performs reconstruction of pulse data from raw data. The following interface specifies that this procedure takes Raw and Geometry as input data set types and produces Pulse as an output type:

```
(Pulse pulse ) reconstruct ( Raw raw, Geometry geo) {
    /* procedure body */
}
```

We present the corresponding procedure body in Section 17.5.4.

17.5.4 Atomic Procedures

An atomic procedure defines an interface to an external executable program or Web service and specifies how logical data types are mapped to and from application program or service arguments and results.

Invoking Application Programs

An atomic procedure that defines an interface to an external program must specify:

- The interpreter that will be used to execute the program
- The name of the program to be executed
- The syntax of the command line that must be created to invoke the program
- The data context in which the program is to execute (i.e., the physical data sets that need to be available to the program when it executes)

An atomic procedure has a header that specifies its interface in terms of data set types, but its body operates on data set representations. Thus, expressions in the data set body must be able to use mapping functions (Section 17.5.2) to map between types and representations.

The header of an atomic procedure that defines an interface to an external program specifies the name of the program to be invoked (the atomic procedure call), the data to be passed to the procedure's execution site (the atomic procedure's input arguments), and the resulting data to be returned back from the procedure's execution site (the atomic procedure's return arguments).

The body of such an atomic procedure specifies how to set up its execution environment and how to assemble the call to the procedure. For example, the following procedure *reconstruct* defines a VDL interface to a cosmic ray data-processing application of the same name. The statements in the body assemble the command to invoke the program, with the "bash" statement indicating that invocation is to occur by using the bash shell command interpreter. The @ notation is used to invoke a mapping function. In this example, the mapping function "filename" is called to extract filenames from the data set representation so they can be passed to the shell.

```
(Pulse pulse ) reconstruct ( Raw raw, Geometry geo) {
    bash {
      reconstruct
        @filename( raw )
        @filename( geo )
        @filename( pulse )
    }
}
```

This atomic procedure may be invoked by a procedure call such as

```
Pulse p1 = reconstruct ( raw1, geo );
```

which takes a raw data set *raw1* and a geometry data set *geo* as inputs and generates as output a pulse data set *p1*. The data sets *raw1*, *geo*, and *p1* are defined as data set handles, which include the typing and mapping specifications for these data sets.

```
RawData   raw1 <file_mapper;
               location="/quarknet/group1/detector1/observation1/rawdata">
Geometry geo  <file_mapper;
               location="/quarknet/group1/detector1/observation1/geometry">
Pulse    p1   <file_mapper;
               location="/quarknet/group1/detector1/processed/pulse/p1">
```

The procedure call is compiled into the execution of the following command line:

```
    reconstruct    /quarknet/group1/detector1/observation1/rawdata \
                   /quarknet/group1/detector1/observation1/geometry \
                   /quarknet/group1/detector1/processed/pulse/p1
```

If this command is executed on a remote host that does not share a file system, then VDS must ensure that the physical representations of data sets passed as input arguments are transferred to the remote site, enabling the executing application to access the required physical files. For example, in the call just shown, the physical representations of the data sets *raw1* and *geo* must be transferred to the remote site.

Similarly, output data (e.g., *p1* in the example call) must be made accessible to other program components. To this end, the existence of the physical data on the remote site is recorded. In addition, the data are optionally copied to a specified site to create an additional replica (which often serves as an archival copy). Finally, the xview itself must be updated to be brought back in sync with the physical representation.

Invoking Web Services

We envision that atomic procedure definitions could also specify Web service interfaces. Such procedures would have the same procedure prototype header as an application program interface but provide a different body. The following example defines a Web service implementation of the same *reconstruct* procedure that was defined above as an executable application.

```
            (Pulse pulse) reconstruct (Raw raw, Geometry geo)
            {
                service {
                    wsdlURI = "http://quarknet.org/cosmic.wsdl";
                    portType = "detectorPT";
                    operation = "reconstruct";
                    soapRequestMsg  = { rawdata = raw;
                            geometry = geo};
                    soapResponseMsg = { pulsedata = pulse};
                }
            }
```

Not shown here is the specification of how arguments such as raw, geo, and pulse are to be passed to and from the Web service. For this, data transport options such as the following will be required:

1. File reference: A reference to a file is passed in the Web service message in the form of a URI.
2. File content: The content of a file is encoded into an XML element and passed in the message body.
3. SOAP attachment: The content of a file is passed as a SOAP attachment.

17.5.5 Compound Procedures

A compound procedure contains a set of calls to other procedures. Variables in the body of a compound procedure specify data dependencies and thus the directed arcs for the DAG corresponding to the compound procedure's workflow. For example:

```
(Shower s) showerstudy (Observation o1, Observation o2) {
    Pulse p1 = thresholdPulse (o1.pulse);
    Pulse p2 = thresholdPulse (o2.pulse);
    Pulse p = correlateEvents (p1, p2);
    s = selectEvents (p);
}
```

In the procedure *showerstudy*, which computes the correlation between two observations, the pulse events from each observation are first filtered by a thresholding procedure, then the results of the thresholding procedures are combined by a correlation procedure, and finally interesting shower events are selected from the combined events. In this compound procedure, data dependencies dictate that the two invocations of *thresholdPulse* can be executed in parallel, after which the calls to *correlateEvents* and *selectEvents* must execute in sequence.

Arbitrary workflow DAGs can be specified in this manner, with the nodes of the DAGs being procedure calls and the edges represented by variables, which are employed to pass the output of one procedure to the input of another.

17.5.6 Control-Flow Constructs

Control-flow constructs are special control entities in a workflow that control the direction of execution. VDL provides *if*, *switch*, *foreach*, and *while* constructs, with syntax and semantics similar to comparable constructs in high-level languages. We illustrate the use of the *foreach* construct in the following example:

```
genPulses ( Detector det ) {
    foreach Observation o in det.ob {
        o.pulse = reconstruct (o.raw, o.geo);
    }
}
```

This example applies the atomic procedure *reconstruct* to each of the observations associated with a specific detector *det* and generates the pulse data for each observation from the raw data. All of the calls to *reconstruct* can be scheduled to run in parallel.

17.6 An Application Example: Functional MRI

VDL provides an effective way to specify the preprocessing and analysis of the terabytes of data contained in scientific archives such as the fMRI Data Center (fMRIDC: `www.fmridc.org`), based at Dartmouth College. This publicly available repository includes complete data sets from published studies of human cognition using functional magnetic resonance imaging (fMRI). Data Sets include 4D functional image-volume time-course data, high-resolution images of brain anatomy, study metadata, and other supporting data collected as part of the study. The fMRIDC curates and packages these data sets for open dissemination to researchers around the world, who may use the data to conduct novel analyses, test alternative hypotheses, explore new means of data visualization, or for education and training.

17.6.1 Overview of fMRI Data Sets

fMRI data sets are derived by scanning the brains of subjects as they perform cognitive or manual tasks. The raw data for a typical study might consist of three subject groups with 20 subjects per group, five experimental runs per subject, and 300 volume images per run, yielding 90,000 volumes and over 60 GB of data. A fully processed and analyzed study data set can contain over 1.2 million files. In a typical year at the Dartmouth Brain Imaging Center, about 60 researchers preprocess and analyze about 20 concurrent studies.

Experimental subjects are scanned once to obtain a high-resolution image of their brain anatomy ("anatomical volume") and then scanned with lower resolution at rapid intervals to observe the effects of blood flow from the "BOLD" (blood oxygenated level dependent) signal while performing some task ("functional runs"). These images are preprocessed and subjected to intensive analysis that begins with image processing and concludes with a statistical analysis of correlations between stimuli and neural activity.

Figure 17.5 illustrates some of the conventions that are frequently used in the physical representation of such fMRI data sets. The logical representation on the left shows the hierarchy of objects in a hypothetical study, while the physical representation on the right indicates the typical manner in which the objects in the logical view are physically represented in a hierarchical file system directory, making heavy use of the encoding of object identifiers into the names of files and directories.

The VDL examples in the next subsections are based on a workflow, AIRSN, that performs *spatial normalization* for preprocessing raw fMRI data prior to analysis. AIRSN normalizes sets of time series of 3D volumes to a standardized coordinate system and applies motion correction and Gaussian smoothing.

```
DBIC Archive                        DBIC Archive
  Study #'2004 0521 hgd'              Study_2004.0521.hgd
    Group #1                            Group 1
      Subject #'2004 e024'                Subject_2004.e024
        Anatomy                                 volume_anat.img
          high-res volume                       volume_anat.hdr
        Functional Runs                         bold1_001.img
          run #1                                bold1_001.hdr
            volume #001                         ...
            ...                                 bold1_275.img
            volume #275                         bold1_275.hdr
          ...                                     ...
          run #5                                bold5_001.img
            volume #001                           ...
            ...                                 snrbold*_*
            volume #242                     ...air*
                                          ...
    Group #5                            Group 5
      ...                                  ...
  Study #...                          Study ...
```

Figure 17.5: fMRI structure — logical (left) and physical (right).

17.6.2 fMRI Data Set Type Definitions

Figure 17.6 shows the VDL types that represent the data objects of Figure 17.5. A *Volume* contains a 3D image of a volumetric slice of a brain image, represented by an *Image* (voxels) and a *Header* (scanner metadata). As we do not manipulate the contents of those objects directly within this VDL program, we do not further decompose their structure. A time series of volumes taken from a functional scan of one subject, doing one task, forms a *Run*. In typical experiments, each *Subject* has an anatomical data set, *Anat*, and multiple input and normalized runs.

Specific output formats involved in processing raw input volumes and runs may include outputs from various image-processing tools, such as the automated image registration (AIR) suite [475]. The type *Air* corresponds to one of the data set types created by these tools (and it, too, needs no finer decomposition).

```
type Volume { Image img; Header hdr; }
type Run { Volume v[ ]; }
type Anat Volume;
type Subject { Anat anat; Run run [ ]; Run snrun [ ]; }
type Group { Subject s[ ]; }
type Study { Group g[ ]; }
type AirVector { Air a[ ]; }
type NormAnat { Anat aVol; Warp aWarp; Volume nHires; }
```

Figure 17.6: VDL type definition for fMRI data.

17.6.3 VDL Procedures for AIRSN

Figure 17.7 shows a subset of the VDL procedures for AIRSN. The procedure *functional()* expresses the steps in Figure 17.8; *airsn_subject()* calls this procedure once per each component and *anatomical()* (not shown) to process a *Subject*. *airsn_subject()* creates in the *Subject* data set a new spatially normalized *Run* for each raw *Run*. Such procedures define how the workflow is expanded to process the members of a data set, as in Figure 17.9.

```
(Run snr) functional( Run r, NormAnat a, Air shrink ) {
    Run yroRun = reorientRun( r , "y" );
    Run roRun = reorientRun( yroRun , "x" );
    Volume std = roRun[0];
    Run rndr = random_select( roRun, .1 ); //10% sample
    AirVector rndAirVec = align_linearRun( rndr, std, 12, 1000, 1000, [81,3,3] );
    Run reslicedRndr = resliceRun( rndr, rndAirVec, "o", "k" );
    Volume meanRand = softmean( reslicedRndr, "y", null );
    Air mnQAAir = alignlinear( a.nHires, meanRand, 6, 1000, 4, [81,3,3] );
    Volume mnQA = reslice( meanRand, mnQAAir, "o", "k" );
    Warp boldNormWarp = combinewarp( shrink, a.aWarp, mnQAAir );
    Run nr = reslice_warp_run( boldNormWarp, roRun );
    Volume meanAll = strictmean ( nr, "y", null )
    Volume boldMask = binarize( meanAll, "y" );
    snr = gsmoothRun( nr, boldMask, 6, 6, 6 );
}

airsn_subject( Subject s, Volume atlas, Air ashrink, Air fshrink ) {
    NormAnat a = anatomical( s.anat, atlas, ashrink );
    Run r, snr;
        int i;
        foreach (r,i) in s.run {
                snr = functional( r, a, fshrink );
                s.snrun[ i ] = snr;
        }
}
```

Figure 17.7: VDL fMRI procedure examples.

17.6.4 The AIRSN Workflow

Figures 17.8 and 17.9 show two views of the most data-intensive segment of the AIRSN workflow, which processes the data from the functional runs of a study. Figure 17.8 is a high-level representation in which each oval represents an operation performed on an entire *Run*. Figure 17.9 expands the workflow to the *Volume* level for a data set of ten functional volumes. (Note that the *random_select* call is omitted in Figure 17.9). In realistic fMRI experiments, *Runs* might include hundreds or thousands of *Volumes*.
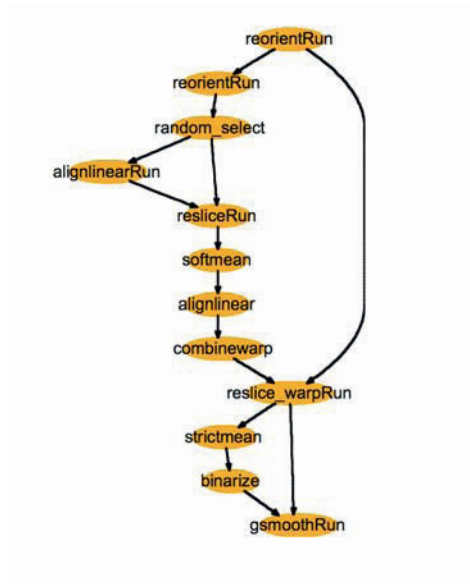
Figure 17.8: AIRSN workflow high-level representation.
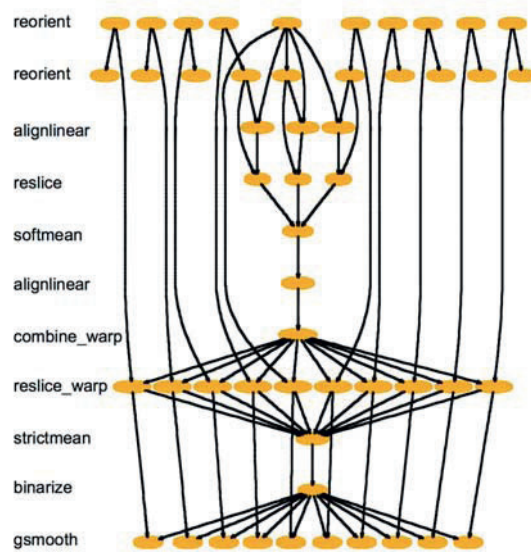
Figure 17.9: AIRSN workflow expanded to show all atomic file operations for a ten volume run.

17.7 VDL Implementation

We have developed a prototype system that can process VDL type definitions and mappings, convert a typed workflow definition into an executable DAG, expand DAG nodes dynamically to process subcomponents of a compound data set, and submit and execute the resulting DAG in a Grid environment. The separation of data set type and physical representation that we achieve with VDL can facilitate various runtime optimizations and graph-rewriting operations [112]. The prototype implements the runtime operations needed to support typed VDL data set processing and execution, which is the principal technical challenge of implementing VDL. We have also verified that we can invoke equivalent services and applications from the same VDL.

The prototype extends an earlier VDS implementation with features to support data typing and mapping. We use the VDS graph traversal mechanism to generate an abstract DAG in which transformations are not yet tied to specific applications or services and data objects are not yet bound to specific locations and physical representations. The extended VDS also enhances the DAG representation by introducing "foreach" nodes (in addition to the existing "atomic" nodes) to represent *foreach* statements in a VDL procedure. These nodes are expanded at runtime (see Section 17.7.2), thus enabling data sets to have a dynamically determined size.

The resulting concrete DAG is executed by the DAGMan ("DAG manager") tool. DAGMan provides many necessary facilities for workflow execution, such as logging, job status monitoring, workflow persistence, and recursive fault recovery. DAGMan submits jobs to Grid sites via the Globus GRAM protocol.

17.7.1 Data Mapping

Our prototype employs a table-driven approach to implement XDTM mapping for data sets stored on file systems. Each table entry specifies

```
name:     the data object name
pattern:  the pattern used to match filenames
mode:FILE (find matches in directory)
     RLS (find matches via replica location service)
     ENUM (data set content is enumerated)
content:  used in ENUM mode to list content
```

When mapping an input data set, this table is consulted, the pattern is used to match a directory or replica location service according to the mode, and the members of the data set are enumerated in an in-memory structure that models the behavior of the xview. This structure is then used to expand *foreach* statements and to set command-line arguments.

For example, in Figure 17.5, a *Volume* is physically represented as an image/header file pair, and a *Run* as a set of such pairs. Furthermore, multiple *Runs* may be stored in the same directory, with different *Runs* distinguished by a prefix and different *Volumes* by a suffix. To map this representation to

the logical *Run* structure, the pattern "bold*N*∗" is used to identify all pairs in Run *N* at a specified location. Thus, the mapper, when applied to the following eight files, identifies two runs, one with three *Volumes* (*Run 1*) and the other with one (*Run 2*).

```
bold1_001.img    bold1_001.hdr
bold1_002.img    bold1_002.hdr
bold1_003.img    bold1_003.hdr
bold2_007.img    bold2_007.hdr
```

17.7.2 Dynamic Node Expansion

A node containing a *foreach* statement must be expanded dynamically into a set of nodes: one for each member of the target data set specified in the *foreach*. This expansion is performed at runtime: When a *foreach* node is scheduled for execution, the appropriate mapper function is called on the specified data set to determine its members, and for each member of the data set identified (e.g., for each *Volume* in a *Run*), a new job is created in a "sub-DAG."

The new sub-DAG is submitted for execution, and the main job waits for the sub-DAG to finish before proceeding. A postscript for the main job takes care of the transfer and registration of all output files and the collection of those files into the output data set. This workflow expansion process may recurse further if the subcomponents themselves also include *foreach* statements. DAGMan provides workflow persistence in the event of system failures during recursion.

The process of dynamic node expansion can be performed in a cursor-like manner to efficiently navigate large data sets. Large data sets behave *as if* the entire data set is expanded in the xview. A naïve implementation would do exactly that, but a more sophisticated implementation can yield better performance by taking advantage of operations that "close" members after they are mapped and that scroll through large sequences of members in a cursor-like fashion to enable arbitrarily large data sets to be mapped.

17.7.3 Optimizations and Graph Transformation

Since data set mapping and node expansion are carried out at runtime, we can use graph transformations to apply optimization strategies. For example, in the AIRSN workflow, some processes, such as the *reorient* of a single *Volume*, only take a few seconds to run. It is inefficient to schedule a distinct process for each *Volume* in a *Run*. Rather, we can combine multiple such processes to run as a single job, thus reducing scheduling and queuing overhead.

As a second example, the *softmean* procedure computes the mean of all *Volumes* in a *Run*. For a data set with a large number of *Volumes*, this stage is a bottleneck, as no parallelism is engaged. There is also a practical issue: The executable takes all *Volume* filenames as command-line arguments, which can exceed limits defined by the Condor and UNIX shell tools used within our

VDS implementation. Thus, we transform this node into a tree in which leaf nodes compute over subsets of the data set. The process repeats until we get a single output. The shape of this tree can be tuned according to the available computing nodes and data set sizes to achieve optimal parallelism and avoid command-line length limitations.

17.8 Conclusion

We have designed a typed workflow notation and system that allows workflows to be expressed in terms of declarative procedures that operate on XML data types and are then executed on diverse physical representations and distributed computers. We have shown, via studies that compare program sizes with and without our notation [496], that this notation and system can be used to express large amounts of distributed computation easily.

The productivity leverage of this approach is apparent: A small group of developers can define VDL interfaces to the application programs and utilities used in a scientific domain and then create a library of data set types, mappers, and data set iteration functions. Such a "virtual data library" encapsulates low-level details concerning how data are grouped, transported, cataloged, passed to applications, and collected as results. Other scientists can then use such libraries to construct workflows without needing to understand the details of physical representation and furthermore are protected by the XDTM type system from forming workflows that are not type compliant. The data management conventions of a research group can be encoded and uniformly maintained with XDTM mapping functions, thus making it easier to curate data set collections that may include many tens of millions of files.

We next plan to automate the compilation steps that were performed manually in our prototype and to create a complete workflow development and execution environment for our XDTM-based VDL. We will also investigate support for services, automation of type coercions between differing physical representations, and recording of provenance for large data collections.

Acknowledgments

Frameworks and Tools: Workflow Generation,
Refinement, and Execution

Workflow-Level Parametric Study Support by MOTEUR and the P-GRADE Portal

Tristan Glatard, Gergely Sipos, Johan Montagnat, Zoltan Farkas, and Peter Kacsuk

18.1 Introduction

Many large-scale scientific applications require the processing of complete data sets made of individual data segments that can be manipulated independently following a single analysis procedure. Workflow managers have been designed for describing and controlling such complex application control flows. However, when considering very data-intensive applications, there is a large potential parallelism that should be properly exploited to ensure efficient processing. Distributed systems such as Grid infrastructures are promising for handling the load resulting from parallel data analysis and manipulation. Workflow managers can help in exploiting the infrastructure parallelism, given that they are able to handle the data flow resulting from the application's execution.

To handle users' processing requests, two main strategies have been proposed and implemented in Grid middleware: the *task-based* approach, where a computing task is formally described before being submitted; and the *service-based* approach, where a computation handled by an external service is invoked through a standard interface. Both approaches have led to the design of different workflow managers. They significantly differ

- in the way data flows are described and manipulated; and
- regarding the optimizations that can be achieved for executing the workflows.

In particular, in the context of scientific applications, it is often necessary to run experiments following a single workflow but considering different, and sometimes dynamic, input data sets. We will name as *parametric applications* such data-intensive scientific procedures to underline the variable nature of their data flows. Workflow managers are expected to offer both

- a high level of flexibility in order to enable *parametric studies* based on these applications; and
- a Grid interface and workflow optimization strategies in order to ensure efficient processing.

In Section 18.2, we introduce the task-based and the service-based approaches in more detail. We then study their differences in terms of managing the resulting data flows (Section 18.3) and computation flows (Section 18.4). In Section 18.7, we introduce P-GRADE portal, a generic interface to both approaches. P-GRADE portal is able to use both the task-based DAGMan and the service-based MOTEUR [161, 304] (hoMe-made OpTimisEd scUfl enactoR) workflow managers. It conciliates to both approaches as much as possible (Section 18.5), and it offers a single interface to describe a data-intensive workflow. The execution technique to be used can then be selected by the user.

18.2 Task-Based and Service-Based Workflows

In the *task-based* strategy, also referred to as *global computing*, users define computing tasks to be executed. Any executable code may be requested by specifying the executable code file, input data files, and command-line parameters to invoke the execution. The task-based strategy, implemented in Globus [144], LCG2 [248], or gLite [165] middleware for instance, has already been used for decades in batch computing. It straightforwardly enables legacy code execution without requiring any modification, provided that the user knows the command line of the code to be launched. An emblematic workflow manager using the task-based framework is the directed acyclic graph manager (DAGMan [97]) from Condor (see Chapter 22) and other frameworks (e.g., VDS), are built on top of this (see Chapters 17 and 23 for instance).

The *service-based* strategy, also referred to as *meta computing*, consists of wrapping application codes into standard interfaces. Such services are seen as black boxes from the workflow manager, for which only the invocation interface is known. Various interfaces, such as Web services [457] (also see Chapter 12) or GridRPC [309], have been standardized. The services paradigm has been widely adopted by middleware developers for the high level of flexibility that it offers (e.g. in the Open Grid Service Architecture [146] and the WS-RF extension to Web services). However, this approach is less common for application code, as it requires all codes to be instrumented with the common service interface. Yet, the service-based approach has been adopted in well-known workflow managers such as the Kepler system [272], Taverna (see Chapter 19), Triana (see Chapter 20), and MOTEUR.

The main difference between the task-based and the service-based approaches is the way data sets to be processed are being handled. In the task-based approach, input data segments are specified with each task. This representation mixes data and processing descriptions. The dependency between two tasks is explicitly stated as a data dependency in these two task descriptions. This representation is static and convenient for optimizing the corresponding computations: The full oriented graph of tasks is known when the computations are scheduled, thus enabling many optimization

opportunities for the workflow scheduler [54]. Conversely, the service-based approach decouples data and processing. Input data sets are dynamically specified at execution time as input parameters to the workflow manager. Each service is defined independently from the data sets to be processed, and it is only at the service invocation time that input data segments are sent to the service. This eases the reexecution of application workflows on different input data. In this framework, the dependencies between consequent services are logically defined at the level of the workflow manager. Each service is designed independently of the others.

18.3 Describing Parametric Application Workflows

18.3.1 Dynamic Data Sets

The nonstatic nature of data descriptions in the service-based approach enables dynamic extensions of the data sets to be processed: A workflow can be defined and executed even though the complete input data sets are not known in advance, perhaps because the data segments are being dynamically fed in as they are produced. Indeed, it is common in scientific applications that data acquisition is a heavyweight process and that data are progressively produced. Some workflows may even act on the data production source itself, stopping data production when sufficient inputs are available to produce meaningful results.

Due to the dynamic nature of data and data interdependencies, it is not always possible to define loops and therefore task-based workflows are typically represented using directed and acyclic graphs (DAGs). Only in the case where the number of iterations is statically known may a loop be expressed by unfolding it in the DAG. However, if the loop condition is dynamically determined (e.g. in optimization loops, which are very frequent in scientific applications), the task-based approach cannot be used. In a workflow of services, loops may exist since the circular dependence on the data segments is not explicitly stated in the graph of services. This enables the implementation of more complex control structures.

Most importantly, the dynamic extensibility of input data sets for each service in a workflow can also be used for defining different data composition strategies, as introduced in Section 18.3.2. The data composition patterns and their combinations offer a very powerful tool for describing the complex data-processing scenarios needed in scientific applications. For the users, this means the ability to describe and schedule very complex processing in an elegant and compact framework.

18.3.2 Data Composition Patterns

A very important feature associated with the service-based approach for describing scientific applications is the ability to define different data com-

position strategies over the input data set of a service. When a service owns two or more input ports, a data composition strategy describes how the data segments received on the inputs are combined prior to service invocation. There are two main composition strategies illustrated in Figure 18.1.

Let us consider two input data sets, $\mathbf{A} = \{A_0, A_1, \ldots, A_n\}$ and $\mathbf{B} = \{B_0, B_1, \ldots, B_m\}$, as an example. The most common data composition pattern is a *one-to-one* association of the input data segments (A_0 is being processed with B_0, A_1 with $B_1, \ldots$) as illustrated in left of Figure 18.1. It results in the invocation of the service $\min(n, m)$ times (usually, $m = n$ in this context) and the production of as many results. Another common strategy is an *all-to-all* composition, illustrated on the right in Figure 18.1, where each data segment in the first set is processed with all data segments in the second set. It results in $m \times n$ service invocations. We will denote by $\mathbf{A} \oplus \mathbf{B}$ and $\mathbf{A} \otimes \mathbf{B}$ the one-to-one and the all-to-all compositions of data sets $\mathbf{A}$ and $\mathbf{B}$.

Many other strategies could be implemented, but these two are the most commonly encountered and are sufficient for implementing most applications. The consideration of binary composition strategies only is not a limitation, as several strategies may be used pairwise for describing the data composition pattern of a service with more than two inputs.
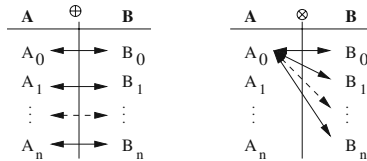
Figure 18.1: *One-to-one* (left) and *all-to-all* (right) composition strategies.

18.3.3 Data Synchronization Barriers

Some special workflow services require the complete data set (not just one data segment) to perform their computation. This is the case for many statistical operations computed on the data sets, such as the calculation of a mean or a standard deviation over the produced results, for instance. Such services are introducing *data synchronization* in the workflow execution, as they represent real barriers, waiting for all input data to be processed before being executed. They can be easily integrated into workflows of services. The workflow manager will take care of invoking the service only once, as soon as all input data sets are available.

18.3.4 Generating Parametric Workflows

The expressiveness of the application description language has consequences for the kind of applications that can be described. Using composition strategies

to design complex data interaction patterns is a very powerful tool for data-intensive application developers. In the task-based framework, two input data segments, even when processed by the same algorithm, result in the definition of two independent tasks. This becomes very tedious and quickly even humanly intractable when considering the very large data sets to be processed (the all-to-all compositions may produce a considerable number of tasks). Additional higher-level tools are needed to automatically produce the huge resulting DAGs, such as the P-GRADE portal (see Section 18.7).

Workflows of services easily handle the description of input data sets independently from the workflow topology itself. Adding extra inputs or considering parametric inputs does not result in any additional complexity. For instance, the Scufl description language from the Taverna workbench (see Chapter 19) can define one-to-one and all-to-all compositions (known as *dot product* and *cross product* iteration strategies). The service-based approach offers the maximum flexibility when dealing with dynamically extensible data sets.

18.4 Efficient Execution of Data-Intensive Workflows

When considering Grid infrastructures with a large potential for parallelism and optimization in data-intensive applications, efficiency needs to be taken into account to avoid performance drops. Although very convenient for representing workflows independently from data sets to be processed, the service-based approach introduces an extra layer between the workflow manager and the execution infrastructure that hides one from the other [162]. The workflow manager does not directly control the execution of computing tasks to a target infrastructure but delegates this role to the services, which are seen as black boxes. The infrastructure used and the way processings are handled are fully dependent on the service implementation.

Many solutions have been proposed in the task-based paradigm to optimize the scheduling of an application in distributed environments [81]. Concerning workflow-based applications, previous works [54] propose specific heuristics to optimize the resource allocation of a complete workflow. Even if it provides remarkable results, this kind of solution is not directly applicable to the service-based approach. Indeed, in this latest approach, the workflow manager is not responsible for the task submission and thus cannot optimize the resource allocation.

Focusing on the service-based approach, nice developments such as DIET middleware [78] and comparable approaches [31, 405] introduce specific strategies such as hierarchical scheduling. In [77], for instance, the authors describe a way to handle file persistence in distributed environments, which leads to strong performance improvements. However, these works focus on middleware design and do not include yat any workflow management. Moreover, those solutions require that specific middleware components be

deployed on the target infrastructure. Hence, there is a strong need for precisely identifying generic optimization solutions that apply to service-based workflows.

In the following sections, we explore different levels of parallelism that can be exploited for optimizing workflow execution in a service-based approach, thus offering the flexibility of services and the efficiency of tasks. We describe them and study their theoretical impact on performance with respect to the characteristics of the application considered.

18.4.1 Asynchronous Calls

To enable parallelism during the workflow execution, multiple application tasks or services have to be called concurrently. In the task-based approach, this means that the workflow manager should be able to concurrently submit jobs, as is commonly the case (e.g. in DAGMan). In workflows of services, this means that calls made from the workflow manager to the application services need to be non-blocking. GridRPC services may be called asynchronously, as defined in the standard [309]. Web services also theoretically enable asynchronous calls. However, the vast majority of existing Web service implementations do not cover the whole standard, and none of the major implementations [218, 438] provide any asynchronous service calls for now. As a consequence, asynchronous calls to Web services need to be implemented at the workflow manager level by spawning independent system threads for each service being executed.

18.4.2 Workflow Parallelism

Given that asynchronous calls are possible, the first level of parallelism that can be exploited is the intrinsic workflow parallelism depending on the graph topology. For instance, if we consider the meteorological application workflow that is presented in Figure 18.2, services `cummu`, `visib`, and `satel` may be executed in parallel. This optimization is usually implemented in all workflow managers.

18.4.3 Data Parallelism

When considering data-intensive applications, several input data sets need to be processed independently using a given workflow. Benefiting from the large number of resources available in a Grid, the same workflow service can be instantiated multiple times on different hardware resources to concurrently process different data segments. Enabling *data parallelism* implies, on the one hand, that the services are able to process many parallel connections and, on the other hand, that the workflow engine is able to submit several simultaneous queries to a service, leading to the dynamic creation of several threads.

Moreover, a data parallel workflow engine should implement a dedicated data management system. Indeed, in the case of a data parallel execution, a data segment is able to overtake another one during the processing, and this could lead to a causality problem. To properly tackle this problem, data provenance has to be monitored during the data parallel execution.

Consider the simple subworkflow made of three services and extracted from a meteorological application (Figure 18.2). Suppose that we want to execute this workflow on three independent input data sets D_0, D_1, and D_2. The data parallel execution diagram of this workflow is represented in Figure 18.3. In this kind of diagram, the abscissa axis represents time. When a data set D_i appears on a row corresponding to a service S_j, it means that D_i is being processed by S_j at the current time. To facilitate legibility, we represented with the D_i notation the data segment resulting from the processing of the initial input data set D_i all along the workflow. For example, it is implicit that on the S_2 service row, D_0 actually denotes the data segment resulting from the processing of the input data segment D_0 by S_1. Moreover, on those diagrams we made the assumption that the processing time of every data set by every service is constant, thus leading to cells of equal width. Data parallelism occurs when different data sets appear on a single square of the diagram, whereas intrinsic workflow parallelism occurs when the same data set appears many times on different cells of the same column. Crosses represent idle cycles.

As demonstrated in the next sections, fully taking into account this level of parallelism is critical in service-based workflows, whereas it does not make any sense in task-based ones. Indeed, in this case it is covered by the workflow parallelism because each task is explicitly described in the workflow description.
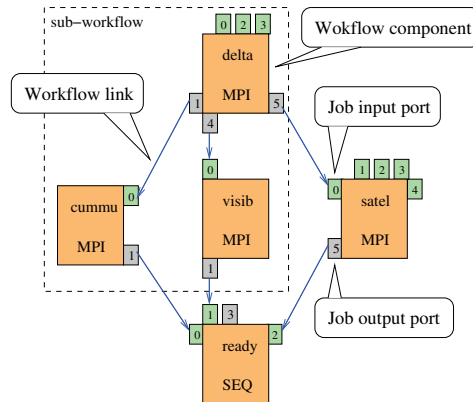
Figure 18.2: MEANDER nowcast meteorology application workflow.

18.4.4 Service Parallelism

Input data sets are likely to be independent from each other. For example, this is the case when a single workflow is iterated in parallel on many input data sets. *Service parallelism* denotes that the processing of two different data sets by two different services is totally independent. This pipelining model, very successfully exploited inside CPUs, can be adapted to sequential parts of service-based workflows. Consider again the simple subworkflow represented in Figure 18.2, to be executed on the three independent input data sets D_0, D_1, and D_2. Figure 18.3 (right) presents a service parallel execution diagram of this workflow. Service parallelism occurs when different data sets appear on different cells of the same column. We did not consider data parallelism in this example.

Here again, we show in the next section that service parallelism is of major importance to optimizing the execution of service-based workflows. In task-based workflows, this level of parallelism does not make any sense because it is included in the workflow parallelism. Data synchronization barriers, presented in Section 18.3.3, are of course a limitation to service parallelism. In this case, this level of parallelism cannot be exploited because the input data sets are dependent on each other.

S_1: delta	D_0 D_1 D_2	X
S_2: cummu	X	D_0 D_1 D_2
S_3: visib	X	D_0 D_1 D_2

S_1: delta	D_0	D_1	D_2	X
S_2: cummu	X	D_0	D_1	D_2
S_3: visib	X	D_0	D_1	D_2
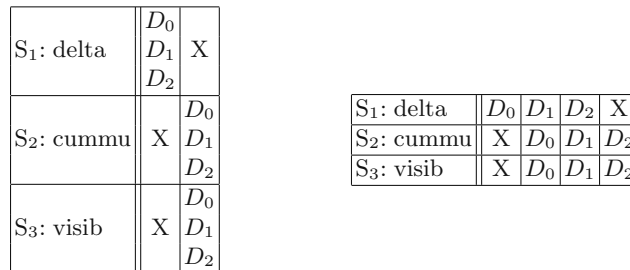
Figure 18.3: Data parallel (left) and service parallel (right) execution diagrams of the sub-workflow extracted from Figure 18.2.

18.4.5 Theoretical Performance Analysis

The data and service parallelisms described above are specific to the service-based workflow approach. To precisely quantify how they influence the application performance we model the workflow execution time for different configurations. We first present general results and then study particular cases, making assumptions on the type of application run.

Definitions and Notations
 In the workflow, a *path* denotes a set of services linking an input to an output. The *critical path* of the workflow denotes the longest path in terms of execution time. n_W denotes the number of services on the critical path

of the workflow, and n_D denotes the number of data sets to be executed by the workflow. i denotes the index of the ith service of the critical path of the workflow ($i \in [0, n_W - 1]$). Similarly, j denotes the index of the jth data set to be executed by the workflow ($j \in [0, n_D - 1]$). $T_{i,j}$ denotes the duration in seconds of the treatment of the data set j by the service i. If the service submits jobs to a Grid infrastructure, this duration includes the overhead introduced by the submission, scheduling, and queuing times. $\sigma_{i,j}$ denotes the absolute time in seconds of the end of the treatment of the data set j by the service i. The execution of the workflow is assumed to begin at $t = 0$. Thus $\sigma_{0,0} = T_{0,0} > 0$. $\Sigma = \max_{j < n_D}(\sigma_{n_W - 1, j})$ denotes the total execution time of the workflow.

Hypotheses
The critical path is assumed not to depend on the data set. This hypothesis seems reasonable for most applications but may not hold in some cases, as for example when workflows include algorithms that contain optimization loops whose convergence time is likely to vary in a complex way with respect to the nature of the input data set.

Data parallelism is assumed not to be limited by infrastructure constraints. We justify this hypothesis by considering that our target infrastructure is a Grid whose computing power is sufficient for our application.

In this section, workflows are assumed not to contain any synchronization service. Workflows containing synchronization barriers may be analyzed as two subworkflows corresponding to the parts of the initial workflow preceding and succeeding the synchronization barrier.

Execution Time Modeling
Under those hypotheses, we can determine the expression of the total execution time of the workflow for different execution policies:

Sequential case (no parallelism) : $\quad \Sigma = \sum_{i < n_W} \sum_{j < n_D} T_{i,j},$

Case DP, data parallelism only : $\quad \Sigma_{DP} = \sum_{i < n_W} \max_{j < n_D} \{T_{i,j}\},$

Case SP, service parallelism only : $\quad \Sigma_{SP} = T_{n_W - 1, n_D - 1} + m_{n_W - 1, n_D - 1},$

$\quad$ with $\begin{cases} \forall i \neq 0, \forall j \neq 0, m_{i,j} = \max(T_{i-1,j} + m_{i-1,j}, T_{i,j-1} + m_{i,j-1}) \\ m_{0,j} = \sum_{k<j} T_{0,k} \text{ and } m_{i,0} = \sum_{k<i} T_{k,0}, \end{cases}$

Case DSP, data + service parallelism : $\Sigma_{DSP} = \max_{j < n_D} \left\{ \sum_{i < n_W} T_{i,j} \right\}.$

All the expressions of the execution time above can easily be shown recursively. Here is an example of such a proof for Σ_{SP}. We first can write that, for a service-parallel but not data-parallel execution:

$$\forall i \neq 0, \forall j \neq 0, \sigma_{i,j} = T_{i,j} + \max(\sigma_{i-1,j}, \sigma_{i,j-1}). \qquad (18.1)$$

Indeed, without data parallelism, data sets are processed one by one and service i has to wait for data segment $j-1$ to be processed by service i before starting to process the data segment j. This expression is illustrated by the two configurations displayed in Figure 18.4. We, moreover, note that (i) service 0 is never idle until the last data set has been processed and (ii) D_0 is sequentially processed by all services. Thus

$$\sigma_{0,j} = \sum_{k \leq j} T_{0,k} \quad \text{and} \quad \sigma_{i,0} = \sum_{k \leq i} T_{k,0}. \tag{18.2}$$
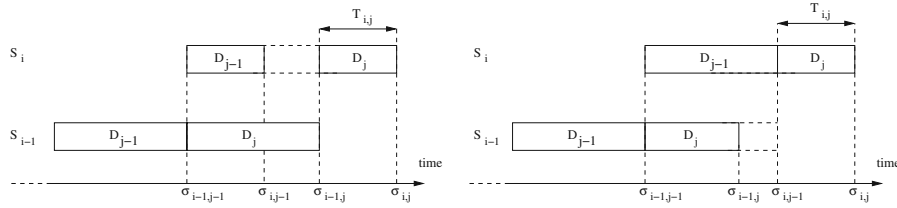
Figure 18.4: Two different configurations for an execution with service parallelism but no data parallelism.

We can then use the following lemma, whose proof is deferred to the end of the section: $P(i,j) : \sigma_{i,j} = T_{i,j} + m_{i,j}$ with $\forall i \neq 0$ and $\forall j \neq 0$, $m_{i,j} = \max(T_{i-1,j} + m_{i-1,j}, T_{i,j-1} + m_{i,j-1})$, $m_{0,j} = \sum_{k<j} T_{0,k}$, and $m_{i,0} = \sum_{k<i} T_{k,0}$. Moreover, we can deduce from Equation 18.1 that for every nonnull integer j, $\sigma_{i,j} > \sigma_{i,j-1}$, which implies that $\Sigma_{SP} = \sigma_{n_W-1,n_D-1}$ (by definition of Σ).

Thus, according to the lemma, $\Sigma_{SP} = T_{n_W-1,n_D-1} + m_{n_W-1,n_D-1}$ with $\forall i \neq 0, \forall j \neq 0, m_{i,j} = \max(T_{i-1,j}+m_{i-1,j}, T_{i,j-1}+m_{i,j-1})$, $m_{0,j} = \sum_{k<j} T_{0,k}$, and $m_{i,0} = \sum_{k<i} T_{k,0}$.

The lemma can be shown via a *double recurrence*, first on i and then on j. Recursively, with respect to i:

- $i = 0$: According to Equation 18.2:

$$\forall j < n_D, \quad \sigma_{0,j} = \sum_{k \leq j} T_{0,k} = T_{0,j} + m_{0,j} \text{ with } m_{0,j} = \sum_{k<j} T_{0,k}.$$

  Thus, $\forall j < n_D$, P(0, j) is true.
- Suppose H_i: $\forall j < n_D$, P(i,j) true. We are going to show recursively with respect to j that H_{i+1} is true:
    - $j = 0$: According to Equation 18.2:

$$\sigma_{i+1,0} = \sum_{k \leq i+1} T_{k,0} = T_{i+1,0} + m_{i+1,0} \text{ with } m_{i+1,0} = \sum_{k<i+1} T_{k,0}.$$

      H_{i+1} is thus true for $j = 0$.

– Suppose K_j: H_{i+1} is true for j. We are going to show that K_{j+1} is true.
According to Equation 18.1, $\sigma_{i+1,j+1} = T_{i+1,j+1} + \max(\sigma_{i,j+1}, \sigma_{i+1,j})$.
Thus, according to K_j, $\sigma_{i+1,j+1} = T_{i+1,j+1} + \max(\sigma_{i,j+1}, T_{i+1,j} + m_{i+1,j})$
and according to H_i,

$$\sigma_{i+1,j+1} = T_{i+1,j+1} + \max(T_{i,j+1} + m_{i,j+1}, T_{i+1,j} + m_{i+1,j})$$
$$= T_{i+1,j+1} + m_{i+1,j+1}$$

with $m_{i+1,j+1} = \max(T_{i,j+1} + m_{i,j+1}, T_{i+1,j} + m_{i+1,j})$.
K_{j+1} is thus true. H_{i+1} is thus true. The lemma is thus true.

Asymptotic Speed-ups
To better understand the properties of each kind of parallelism, it is interesting to study the asymptotic speedups resulting from service and data parallelism in particular application cases.

Massively data-parallel workflows. Let us consider a massively (*embarrassingly*) data-parallel application (a single service S_0 and a very large number of input data). In this case, $n_W = 1$ and the execution time is

$$\Sigma_{DP} = \Sigma_{DSP} = \max_{j < n_D}(T_{0,j}) \ll \Sigma = \Sigma_{SP} = \sum_{j < n_D} T_{0,j}.$$

In this case, data parallelism leads to a significant speedup. Service parallelism is useless, but it does not lead to any overhead.

Non-data-intensive workflows. In such workflows, $n_D = 1$ and the execution time is $\Sigma_{DSP} = \Sigma_{DP} = \Sigma_{SP} = \Sigma = \sum_{i < n_W} T_{i,0}$. In this case, neither data nor service parallelism lead to any speedup. Nevertheless, neither of them introduce any overhead.

Data-intensive complex workflows. In this case, we will suppose that $n_W > 1$ and $n_D > 1$. In order to analyze the speedups introduced by service and data parallelism, we make the simplifying assumption of constant execution times: $T_{i,j} = T$. The workflow execution time then resumes to

$$\Sigma = n_D \times n_W \times T, \quad \Sigma_{DP} = \Sigma_{DSP} = n_W \times T, \quad \Sigma_{SP} = (n_D + n_W - 1) \times T.$$

The speedups associated to the different configurations are thus

$$S_{DP} = \frac{\Sigma}{\Sigma_{DP}} = n_D, S_{DSP} = \frac{\Sigma_{SP}}{\Sigma_{DSP}} = \frac{n_D + n_W - 1}{n_W}, S_{SP} = \frac{\Sigma}{\Sigma_{SP}} = \frac{n_D \times n_W}{n_D + n_W - 1}.$$

Service parallelism does not lead to any speedup if it is coupled with data parallelism: $S_{SDP} = \frac{\Sigma_{DP}}{\Sigma_{DSP}} = 1$. Thus, under those assumptions, service parallelism may not be of any use on fully distributed systems. However, in practice, even in the case of homogeneous input data sets, T is hardly constant in production systems because of the high variability of the overhead due to submission, scheduling, and queuing times on such large-scale and multiuser platforms. The constant execution time hypothesis does not hold. Figure 18.5

illustrates in a simple example why service parallelism can provide a speedup even if data parallelism is enabled, if the assumption of constant execution times does not hold. The left diagram does not take into account service parallelism, whereas the right one does. The processing time of the data set D_0 is twice as long as the other ones on service S_0, and the execution time of the data set D_1 is three times as long as the other ones on service S_1. This can, for example, occur if D_0 was submitted twice because an error occurred and if D_1 remained blocked on a waiting queue. In this case, service parallelism improves performance beyond data parallelism, as it enables some computations to overlap.
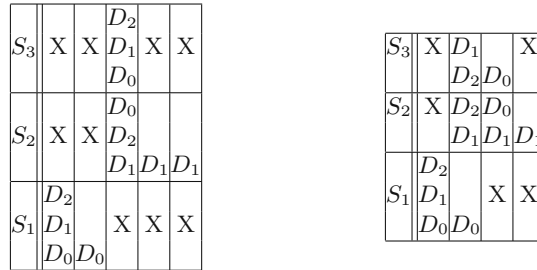
Figure 18.5: Workflow execution time without (left) and with (right) service parallelism when the execution time is not constant.

18.4.6 Application-Level Parallelism

In addition, an application code may be instrumented to benefit from a parallel execution through a standard library (e.g. MPI). The exploitation of this fine-grain level of parallelism is very dependent on the application code and cannot be controlled at the workflow management level. However, the procedure for submitting parallel tasks is often specific in Grid middleware and the workflow manager needs to recognize the specific nature of such jobs to handle them properly. Usually, application-level parallelism can only be exploited intrasite for performance reasons (intersite communication being too slow), while the other levels of parallelism are coarse-grained and can be exploited intersite.

18.5 Exploiting Both Task- and Service-Based Approaches in Parametric Data-Intensive Applications

To execute parametric and data-intensive applications, two approaches are thus possible:

1. In the task-based approach, a high-level tool for transforming the parametric description of the application into a concrete execution DAG is needed prior to the execution of the workflow manager.

2. In the service-based approach, the separate description of the workflow topology and the input data sets is sufficient. However, the efficient execution relies on an optimized workflow manager capable of exploiting parallelism through parallel service calls.

In the task-based framework, it is not possible to express dynamically expandable data sets and loops. However, parallelism is explicitly stated in the application DAG and easy to exploit. The service-based approach offers more flexibility but requires an optimized application enactor, such as MOTEUR, to efficiently process the workflow, enabling all levels of parallelism described above. In the following sections, we introduce the P-GRADE portal and MOTEUR. P-GRADE conciliates both approaches by providing a unique GUI for describing the application workflow in a high-level framework. P-GRADE is interfaced with both DAGMan, for dealing with task-based workflows, and MOTEUR, for handling workflows of services.

18.6 MOTEUR Service-Based Workflow Enactor

MOTEUR [304] was designed with the idea that the service-based approach is making services and data composition easier from the application developer point of view. It is therefore more convenient, provided that it does not lead to performance losses. The MOTEUR (hoMe-made OpTimisEd scUfl enactoR) workflow manager was implemented to support workflow, data, and service parallelism, described in Section 18.4. Our prototype was implemented in Java in order to be platform independent. It is freely available under CeCILL public license (a GPL-compatible open source license).

The workflow description language adopted is the Simple Conceptual Unified Flow Language (Scufl) used by the Taverna engine (see Chapter 19). Apart from describing the data links between the services, the Scufl language allows one to define *coordination constraints* that are control links enforcing an order of execution between two services even if there is no data dependency between them. We used those coordination constraints to identify services that require data synchronization. The Scufl language also specifies the number of threads of a service (fixed number of parallel data). In the case of MOTEUR, this number is ignored, as it is dynamically determined during the execution, considering the number of input data segments available for processing. We developed an XML-based language to describe input data sets. This language aims at providing a file format to save and store the input data set in order to be able to re-execute workflows on the same data set. It simply describes each item of the different inputs of the workflow.

Handling the composition strategies presented in Section 18.3 in a service and data parallel workflow is not straightforward because the data sets produced have to be uniquely identified. Indeed, they are likely to be computed in a different order in every service, which could lead to causality problems

and incorrect mapping of the input parameters in one-to-one composition patterns. Moreover, due to service parallelism, several data sets are processed concurrently and one cannot number all the produced data segments once computations are completed. We have implemented a data provenance strategy to sort out the causality problems that may occur. Attached to each processed data is a history tree keeping track of all the intermediate results computed to process it. This tree unambiguously identifies the data segment.

Finally, MOTEUR implements an interface to both Web services and GridRPC instrumented application code. To ease application code wrapping in services and job submissions on a Grid infrastructure, we provide a generic submission Web service. It encapsulates the user code and handles the interface with the Grid infrastructure. It has been interfaced with both the EGEE [128] production Grid infrastructure and the Grid5000 [173] experimental Grid infrastructure.

18.7 P-GRADE Portal

The goal of the P-GRADE portal is to provide a high-level user interface that hides the low-level details of the underlying Grid systems. Users can construct complex Grid applications as workflows without learning the specific Grid interface. Moreover, the P-GRADE portal plays the role of a bridge between different Grids, solving the interoperability problem at the portal level [230]. The components of a workflow can be executed on any Grid that is connected to the portal and for which the user owns an access certificate. P-GRADE portal 2.3 [352] serves as the production portal service for several different Grid systems: VOCE (Virtual Organization Central Europe of EGEE), HunGrid (Hungarian VO of EGEE), EGRID (Economics VO of EGEE), SEE-GRID (South Eastern European Grid), and UK NGS (National Grid Service). If a portal is configured to access all these Grids, then users can access any resource of these Grids from the same workflow.

The portal provides a graphical interface through which users can easily construct workflows based on the DAG concept. Nodes of the graph can be jobs or GEMLCA legacy code services [117]. Arcs among the nodes represent file transfers between the nodes. The workflow enactor of portal version 2.3 is based on DAGMan, which supports only the task-based strategy. Therefore, parametric applications cannot be defined. This portal version supports two levels of parallelism: application parallelism (Section 18.4.6), which is employed when a node of the workflow is an MPI job that is assigned to a multiprocessor Grid site; and workflow parallelism (Section 18.4.2). However, portal version 2.3 is not able to support data and service parallelisms described in Sections 18.4.3 and 18.4.4, respectively.

In order to support the service-based strategy, parametric study applications, and all kinds of parallelism, we extended the portal with two new features:

1. We have extended the workflow creation interface of the portal in order to enable the definition of parametric study applications.
2. We integrated the MOTEUR workflow enactor within the portal in order to support the service-based strategy and to exploit data parallelism and service parallelism.

This new portal version will support the development of DAGs consisting of normal and parametric jobs as well as Web services. It will also support the execution of components of such workflows in Globus-2, Globus-4, LCG-2, gLite, and Web services Grids. While the normal and parametric job components will be executed in Globus-based Grids using DAGMan, Web service invocations will be forwarded to the MOTEUR workflow enactor as illustrated in Figure 18.6.

The current section focuses on the parametric study extension of the portal and shows the workflow user interface that can support both the MOTEUR enactor described in Section 18.6 and the Condor DAGMan-based enactor.
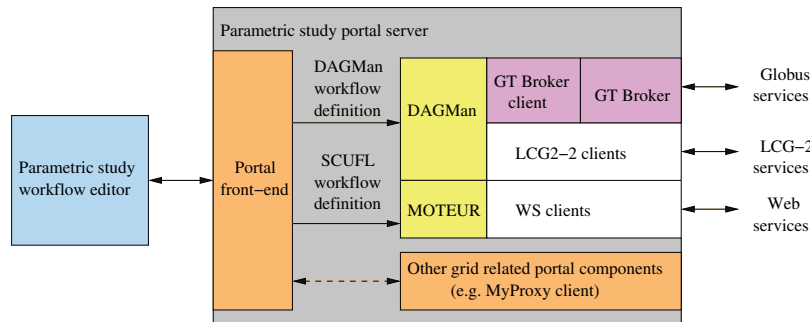
Figure 18.6: Structure of the parametric study version of the P-GRADE portal.

18.7.1 Interface to Workflow Managers

In order to enable parametric studies, the P-GRADE portal includes the new concept of parametric value. It is based on multiple layers, from high-level graphical definition of the workflows to low-level workflow enactment, as illustrated in Figure 18.7. This architecture enables both the representation of parametric application workflows and the transformation of the abstract workflow into a graph of services or a DAG of tasks as required by the underlying workflow enactors.

At the top of the P-GRADE workflow definition process, *parameter spaces* are defined. Parameter spaces enable the description of parametric values. These parametric values are transformed into data segments corresponding to the *data streams* (application input data sets) that will be handled by the workflow manager. At this layer, there are two possibilities, depending on the
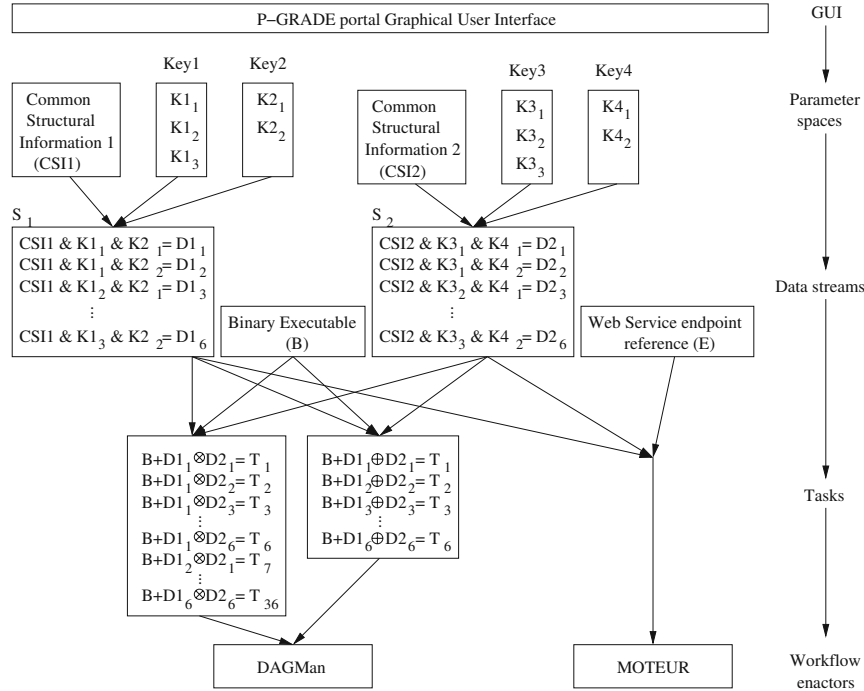
Figure 18.7: The P-GRADE portal multilayer architecture.

user setting: either the input data sets and the services description are sent to MOTEUR for execution in the service-based framework, or data segments are composed with binary executables according to the data composition patterns to build tasks. The DAG of tasks can then be submitted to DAGMan for workflow enactment in the task-based framework.

The P-GRADE portal defines all elements required for defining such parametric application workflows. It proposes a rich and intuitive GUI for describing the workflow graph, defining parameter spaces, and composing data streams. During workflow execution, the P-GRADE portal handles the interface to the workflow manager, monitors the execution, and provides graphical feedback to the user. Examples of workflows described through the P-GRADE GUI are given in Figures 18.2 and 18.8.

18.7.2 DAGMan Workflow Elements

Figure 18.2 illustrates the workflow elements available in P-GRADE portal version 2.3 to define DAGMan workflows on a real application. They include the following elements:

- *Component.* All components are *normal jobs.* A normal job is a program that has one binary executable file and must be started in batch mode. The normal job can be either a sequential or an MPI job. The binary executable of the program is taken from the portal user's machine.
- *Port.* Input and output ports can optionally be connected to jobs. *Normal input ports* represent one file to be used by the connected component as input. A *Normal output port* represents one file to be generated by the connected job during execution.
- *Link.* All links in a task-based workflow are *normal file channels.* They define a data channel between a normal output port and a normal input port that represents a transformation of an output file into an input file of a subsequent task.

Based on these elements, a user can create complex workflow applications that can exploit intrasite (MPI) and intersite (workflow) parallelism.

18.7.3 Parametric Workflow Elements

The parametric workflow elements are useful for representing parametric data-intensive applications. In the P-GRADE portal, the same elements are used for specifying parametric task-based or service-based workflows even though they can be executed in different ways. Figure 18.8 displays the new parametric elements.
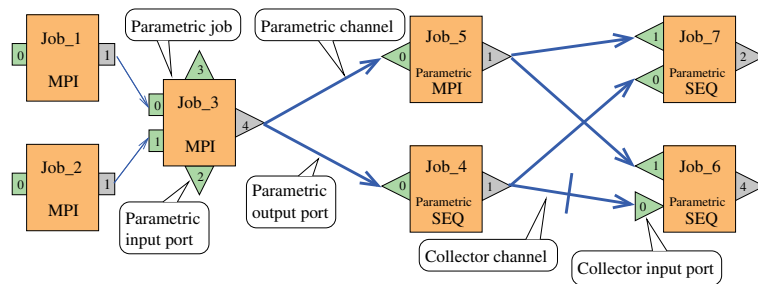
Figure 18.8: Normal and parametric workflow elements in the P-GRADE portal GUI.

Although represented identically in the GUI, the parametric elements differ in their nature. In particular, parametric job inputs are files, represented through *ports*, while Web service inputs are strings (possibly identifying a file), represented through *fields*. The new workflow elements are:

- *Component. Parametric jobs* represent a program that has one binary executable file and must be started in batch mode on independent input

file sets. *Parametric Web services* represent one operation of one Web service that must be invoked multiple times with independent input string sets. Depending on the service implementation, it can submit jobs to a Grid infrastructure when serving the request. Graphically, parametric Web services are identified by the "WS" label, while parametric jobs are labeled "SEQ" or "MPI."

- *Port.* For parametric jobs, *parametric input ports* represent the simultaneously processable instances of a single *file* (files with the same structure but different contents), and *parametric output ports* represent the instances of a single output *file* generated by the instances of a parametric job component (files with the same structure but different contents). Similarly for parametric Web services, *parametric input fields* represent the simultaneously processable instances of an input string, and *parametric output fields* represent the instances of an output string generated by a Web service component.
- *Link. Parametric file* (resp. *parametric string*) *channels* define a data channel between a parametric output and a parametric input port (resp. field). These channels "fire" each time an output data segment becomes available.

In addition, *collector* ports and channels are introduced to represent data synchronization barriers (Section 18.3.3). *Collector input ports* (resp. *fields*) represent N files (resp. strings) with different structures and different contents, which are expected by the connected component as input. They can be connected to both parametric and nonparametric job components through *collector file* (resp. *string*) *channels*. These channels fire only when every output file is available.

Some constraints on the components apply in order to form a semantically correct parametric study workflow application. It makes sense for normal input ports to be connected to a parametric job (every instance of the job is using the same file), while it is not the case for normal output ports. Parametric input ports (resp. fields) can only be connected to parametric job (resp. Web service) components. Parametric jobs (resp. Web services) always have at least one input port (resp. field).

18.7.4 Parameter Spaces and Data Flows

The P-GRADE portal provides a flexible framework for defining variable values of parameters sent to parametric jobs and Web services. The property window of an input parametric port (on the left in Figure 18.9) enables the definition of *keys* (variable values) and *common structural information* (CSI) of the parameters (the common structure of all inputs). The user defines the CSIs for each parameter. A parameter may be n-dimensional, as it may depend on n different input keys $K_1, \ldots, K_n$. The parameter key definition window (on the right in Figure 18.9) enables the definition of a key value generation

rule (types of values, values read from files or generated according to different rules, etc.).
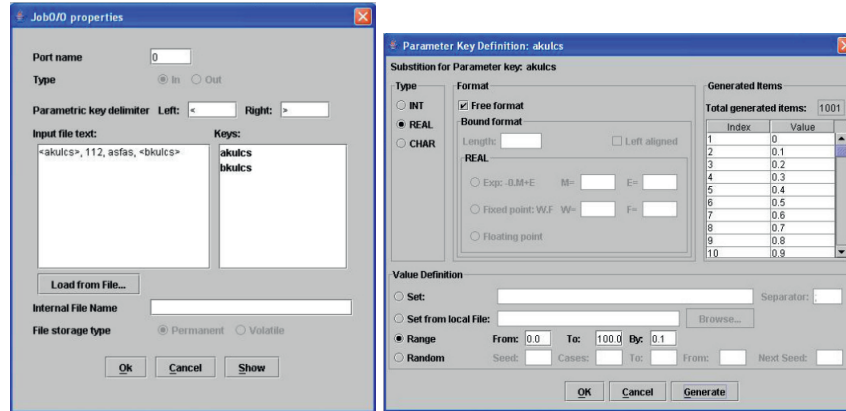
Figure 18.9: Parameter space definition user interface.

The transformation between a parameter space definition and data streams (see Figure 18.7) is an automatic generation process where every occurrence of a key in the CSI is replaced with a parameter value, according to the algorithm presented in Figure 18.10. This algorithm produces an indexed (ordered) array of data segments D. It assumes a precedence order among the keys (primary, secondary...). This precedence order influences the indexing order of data segments. In the P-GRADE portal, the precedence order of keys is the key declaration order. For example, the CSI given in Figure 18.9 (<akulcs>, 112, asfas, <bkulcs>) contains two keys (akulcs and bkulcs). The algorithm will produce the data segments (0, 112, asfas, 0), (0, 112, asfas, 0.1)...

```
for i = 0 to (K1.length - 1)
  primaryKey = K1[i]
  for j = 0 to (K2.length - 1)
    secondaryKey = K2[j]
    D[i * K1.length + j] = replace(CSI, primaryKey, secondaryKey)
  end
end
```

Figure 18.10: Parameter generation algorithm.

18.7.5 Workflow Execution

Workflow applications are taken as input sets of data segments ($S_i = Di_j$). In the case of the MOTEUR enactor, the definition of the input data sets is sufficient to process the workflow. In the case of DAGMan, data streams still need to be composed according to the data composition operators (Section 18.3.2) to produce a list of tasks. The P-GRADE portal interface allows the definition of the one-to-one and the all-to-all data composition strategies on the parametric input data ports (or fields) pairwise. From this input, the data elements, and the job binary, the system generates several computational tasks for each parametric job component (see the tasks layer of Figure 18.7).

Each data segment generated has a unique index value within its set (these values are denoted by the lower indexes in Figure 18.7). The indexes are used by the workflow enactors during workflow execution to determine the order of elements for a one-to-one or all-to-all data composition. Since the computational tasks or the service invocation requests represented by a parametric component are independent from each other, their submission order is irrelevant. Even in the case of a known submission order, the completion time of a task or service is unpredictable. It is the responsibility of the workflow enactment system to keep track of the order of the execution results according to the workflow description.

18.8 Conclusions

Task-based and service-based approaches are two very common frameworks for handling scientific workflows. The service-based approach is very flexible, enabling the expression of complex data composition patterns and dealing with parametric data sets. The task-based approach is more static, but it eases the optimization of the workflow execution since the complete DAG of tasks is known prior to the application execution.

The MOTEUR service-based workflow manager was specifically designed to exploit all levels of parallelism that can be automatically handled by the workflow manager. Using a high-level tool such as the P-GRADE portal, it is possible to describe parametric workflows that will be instantiated either as workflows of services or DAGs of tasks. The P-GRADE portal conciliates the two approaches to some extent, as it automatically produces large DAGs corresponding to data-intensive parametric applications. Yet, the static nature of DAGs does not permit dynamic input data set management, contrary to workflows of services. The P-GRADE portal provides a unique interface for exploiting both approaches. It is relying on MOTEUR and the DAGMan workflow managers to deal with the low-level execution.

18.9 Acknowledgments

Taverna/$^{\mathrm{my}}$Grid: Aligning a Workflow System with the Life Sciences Community

Tom Oinn, Peter Li, Douglas B. Kell, Carole Goble, Antoon Goderis, Mark Greenwood, Duncan Hull, Robert Stevens, Daniele Turi, and Jun Zhao

19.1 Introduction

Bioinformatics is a discipline that uses computational and mathematical techniques to store, manage, and analyze biological data in order to answer biological questions. Bioinformatics has over 850 databases [154] and numerous tools that work over those databases and local data to produce even more data themselves. In order to perform an analysis, a bioinformatician uses one or more of these resources to gather, filter, and transform data to answer a question. Thus, bioinformatics is an *in silico* science.

The traditional bioinformatics technique of cutting and pasting between Web pages can be effective, but it is neither scalable nor does it support scientific best practice, such as record keeping. In addition, as such methods are scaled up, slips and omissions are more likely to occur. A final human factor is the tedium of such repetitive tasks [397].

Doing these tasks programmatically is an obvious solution, especially for the repetitive nature of the tasks. Some bioinformaticians have the programming skills to wrap these distributed resources. Such solutions are, however, not easy to disseminate, adapt, and verify. Moreover, one of the consequences of the autonomy of bioinformatics service providers is massive heterogeneity within those resources. The advent of Web services has brought about a major change in the availability of bioinformatics resources from Web pages and command-line programs to Web services [395], though much of the structural, value-based, and syntactic heterogeneity remains. The consequent lack of a common type system means that services are difficult to join together programmatically, and any technical solution to *in silico* experiments in biology has to address this issue.

Many scientific computing projects within the academic community have turned to workflows as a means of orchestrating complex tasks (*in silico* experiments) over a distributed set of resources. Examples include DiscoveryNet [373] for molecular biology and environmental data analysis,

SEEK for ecology [19, 20], GriPhyn for particle physics [110], and SCEC/IT for earthquake analysis and prediction [236].

Workflows offer a high-level alternative for encoding bioinformatics *in silico* experiments. The high-level nature of the encoding means a broader community can create templates for *in silico* experiments. They are also easier to adapt or repurpose by substitution or extension. Finally, workflows are less of a black box than a script or traditional program; the experimental protocol captured in the workflow is displayed in such a way that a user can see the components, their order, and inputs and outputs. Such a workflow can be seen in Figure 19.1.

$^{\mathrm{my}}$Grid is a project to build middleware to support workflow-based *in silico* experiments in biology. Funded by the United Kingdom's e-Science Programme since 2001, it has developed a set of open-source components that can be used independently and together. These include a service directory [267], ontology-driven search tools over semantic descriptions of external resources and data [267], data repositories and semantically driven metadata stores for recording the provenance of a workflow and the experimental life cycle [494], and other components, such as distributed query processing [16] and event notification.[1]

$^{\mathrm{my}}$Grid's workflow execution and development environment, Taverna, links together and executes external remote or local, private or public, third-party or home-grown, heterogeneous open services (applications, databases, etc.). The Freefluo workflow enactment engine[2] enacts the workflows. The Taverna workbench is a GUI-based application for bioinformaticians to assemble, adapt, and run workflows and manage the generated data and metadata. $^{\mathrm{my}}$Grid components are Taverna plug-ins (for results collection and browsing, provenance capture, service publication, and discovery) and services (such as specialist text mining). Thus the workbench is the user-facing application for the $^{\mathrm{my}}$Grid middleware services. At the time of writing, Taverna 1.3 has been downloaded over 14,000 times[3] and has an estimated user base of around 1500 installations. Taverna has been used in many different areas of research throughout Europe and the United States for functional genomics, systems biology, protein structure analysis, image processing, chemoinformatics, and simulation coordination. Since 2006, $^{\mathrm{my}}$Grid has been incorporated into the United Kingdom's Open Middleware Infrastructure Institute to be "hardened" and developed to continue to support life scientists.

19.1.1 A Bioinformatics Case Study

An exemplar Taverna workflow currently being used for systems biology is shown in Figure 19.1. This workflow uses data stored in distributed databases

---

[1] http://www.mygrid.org.uk.

[2] http://freefluo.sourceforge.net.

[3] See http://taverna.sourceforge.net/index.php?doc=stats.php.

to automate the reconstruction of biological pathways that represent the relationships between biological entities such as genes, proteins, and metabolites.

The interaction pathways generated by the workflow are in the form of a data model, which is specified by the XML-based Systems Biology Markup Language (SBML) [201]. A core SBML workflow is responsible for generating an SBML model. This is then populated, through the SBML API, by the supplementary workflows that gather data for the model (see Figure 19.1). The SBML model can then be used to perform biological simulations.

These workflows typify the needs of bioinformatics analyses. It is a typically datacentric workflow, gathering many kinds of data from a variety of locations and from services of a variety of technology types. As will be seen throughout the chapter, many types of resources are used, and all of these can be incorporated into Taverna. The workflows have to be run repeatedly, and such an analysis would be long and tedious to perform manually.
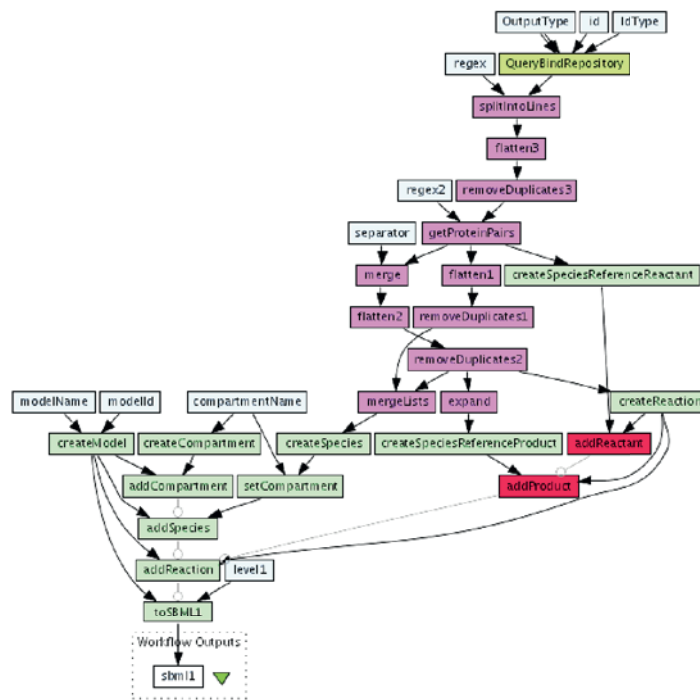
Figure 19.1: An SBML model construction workflow. This workflow retrieves protein interactions from the BIND database, which are then used to populate an SBML model using the core SBML workflow. Four types of processors are used in this example: WSDL, consumer API, local Java, and nested workflow processors. These processors are joined together by data links (arrows) and coordination links.

The rest of this chapter is organized as follows. Section 19.2 further elaborates on the background to Taverna and then Section 19.3 outlines requirements in detail. Section 19.4 introduces the major Taverna components, and architecture. Section 19.5 concentrates on the workflow design and Section 19.6 on executing and monitoring workflows. Section 19.7 completes the workflow life cycle with metadata and provenance associated with managing and sharing results and the workflows themselves. Section 19.8 discusses related work and Section 19.9 reflects on our experiences and showcases future developments in Taverna 2.0.

19.2 The Bioinformatics Background

Life scientists are accustomed to making use of a wide variety of Web-based resources. However, building applications that integrate resources with interfaces designed for humans is difficult and error-prone [395]. The emergence of Web services [58], along with the availability of suitable tool support, has seen a significant number of bioinformatics Web resources become publicly available and described with a Web Services Description Language (WSDL) interface.

There are currently over 3000 services accessible to a ^{my}Grid user. Although the majority involve complex interaction patterns or specific messaging formats, or use different protocols and paradigms, they actually follow a small number of stereotyped patterns. The users' lack of middleware knowledge means they should not be expected to deal with the differences between these patterns. In addition, given the number and distribution of services, users cannot be expected to have existing knowledge of what services are available, where they are, or what they do.

The data produced by these services are mostly semistructured and heterogeneous. There are a large number of data formats, including those for gene sequences and protein sequences, as well as bespoke formats produced by many analysis tools. These are rarely encoded in XML, and there is usually no formal specification that describes these formats. Interpreting or reconciling these data as they are passed between different databases and analysis tools is therefore difficult.

This situation is in contrast with data in other scientific workflow projects that have much more centralized control of data formats. For example, the SEEK project provides tools for ecologists to describe their data using XML schema and ontologies and so support middleware-driven data integration [59].

DiscoveryNet [373] requires each application service to be wrapped, allowing data to adhere to a common format. Other projects are more uniform than ^{my}Grid in the way applications on distributed resources are accessed. For example, abstract Pegasus workflows used in the SCEC/IT project are first compiled into concrete workflows. Each step of a concrete workflow corresponds to a job to be scheduled on a Condor cluster [111].

Taverna differs from these projects by placing an emphasis on coping with an environment of autonomous service providers and a corresponding "open world" model for the underlying Grid and service-oriented architecture. Taverna's target audience of life scientists wants easy access and composition of as wide a range of services as feasible, and this reinforces the need for an open access policy for services, despite the obvious difficulties.

19.3 Aligning with Life Science

From the background and introduction, we can define the key requirements for the Taverna workflow system that drive us to align with life science:

- *Ease of use.* The target end users for Taverna are not necessarily expert programmers.
- *Dataflow centric.* Bioinformaticians are familiar with the notion of dataflow centric analysis. We want to enhance how biologists perform their models of analysis, not change their model of analysis.
- *Open world assumption.* We want to be able to use any service as presented rather than require service providers to implement services in a prescribed manner and thus create a barrier to adoption.
- *Easy and rapid user-driven ad hoc workflow design.* Quickly and easily finding services and adapting previous workflows is key to effective workflow prototyping.
- *Fault tolerant.* Any software operating in a networked, distributed environment is required to cope gracefully with failure.
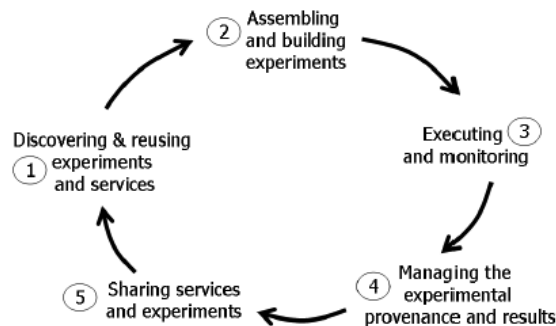
Figure 19.2: The e-Science life cycle.

- *Support for the e-Science life cycle.* Workflows are not a complete solution for supporting *in silico* experiments. They exist in a wider context of scientific data management, as illustrated in Figure 19.2. It is essential that data produced by a workflow carry some record of how and why they were produced, i.e., the provenance of the data.

19.4 Architecture of Taverna

The requirements described have led to several major design lessons. Figure 19.3 illustrates how Taverna takes a layered approach to its overall architecture. This is driven by the need to present a useful, high-level presentation in which biologists can coordinate a variety of resources. Our user base neither knows nor cares about such things as port types, etc. We have a requirement both to present a straightforward perspective to our users and yet cope with the heterogeneous interfaces of our services. A major consequence of this for the workflow system architecture has been to provide a multitiered approach to resource discovery and execution that separates application and user concerns from operational and middleware concerns.

Scufl, a workflow language for linking applications [326], is at the abstraction level of the user; an extensible processor plug-in architecture for the Freefluo enactor manages the low-level "plumbing" invocation complexity of different families of services. In between lies an execution layer interpreting the Taverna Data Object Model that handles user-implied control flows such as implicit iteration over lists and a user's fault-tolerance policies.

Figure 19.3 shows how the $^{\mathrm{my}}$Grid components are divided between the three layers of $^{\mathrm{my}}$Grid's design.

- The Application Data Flow layer is aimed at the user and is characterized by a User-Level workflow object model. The purpose is to present the workflows from a problem-oriented view, hiding the complexity of the interoperation of the services. When combining services into workflows, users think in terms of (see Figure 19.4) the data consumed and produced by logical services and connecting them together. They are not interested in the implementation styles of the services.
- The Execution Flow layer relieves the user of most of the details of the execution flow of the workflow and expands on control-flow assumptions that tend to be made by users. This layer is characterized by the Enactor Internal Object Model and by the $^{\mathrm{my}}$Grid Contextual Information Model. The layer manages list and tree data structures, implicitly iterates over collections of inputs, and implements fault recovery strategies on behalf of the user. This saves the user explicitly handling these at the application layer and avoids mixing the mechanics of the workflow with its conceptual purpose. A drawback is that an expert bioinformatician needs
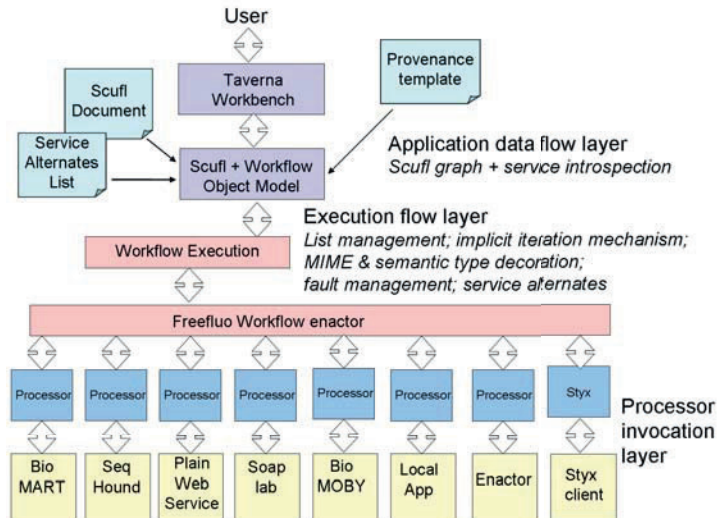
Figure 19.3: An overview of Taverna in layers.

to understand the behavioral semantics of this layer to avoid duplicating the implicit behavior.

- The Processor Invocation layer is aimed at interacting with and invoking concrete services. Bioinformatics services developed by autonomous groups can be implemented in a variety of different styles even when they are similar logical services from a scientist's perspective. This layer is characterized by the Enactor Internal Object Model and is catered to by an extensible processor plug-in architecture for the Freefluo enactment engine.

^{my}Grid is designed to have a framework that can be extended at three levels:

- The first level provides a plug-in framework to add new GUI panels to facilitate user interaction for deriving and managing the behavioral extensions incorporated into Taverna. This extensibility is made available at the workbench layer.
- The second level allows for new processor types to be plugged in to enable the enactment engine to recognize and invoke new types of services (which can be both local and external services). This permits a wider variety of workflows to be constructed and executed. This level of extensibility is provided at the workflow execution layer.
- The third level is provided for loosely integrating external components via an event–observer interface. The workflow enactor generates events during critical state changes as it executes the workflow, exposing snapshots of important parts of its internal state via event objects (i.e., messages). Those event objects are then intercepted and processed by observer plug-

ins that can interact with external services. This level of extensibility is made available at the workflow execution layer.
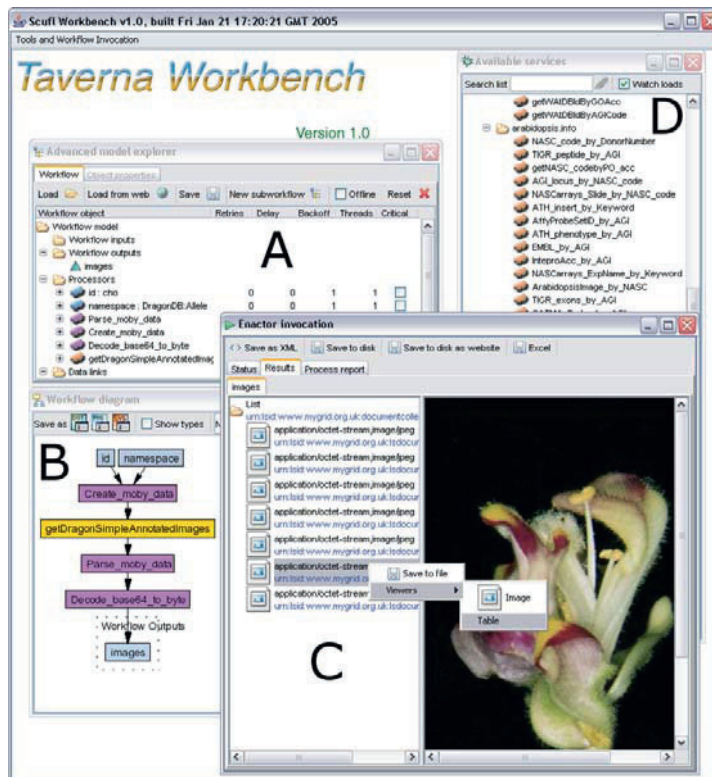
Figure 19.4: The Taverna workbench showing a tree structure explorer (a) and a graphical diagram view (b) of a Scufl workflow. The results of this workflow are shown in the enactor invocation window in the foreground (c). A service palette showing the range of operations that can be used in the composition of a workflow is also shown (d).

The Scufl language [326] is essentially a dataflow centric language, defining a graph of data interactions between different services (or, more strictly, processors). Scufl is designed to reflect the user's abstraction of the *in silico* experiment rather than the low-level details of the enactment of that experiment.

Internally to Taverna, Scufl is represented using a Workflow Object Model along with additional information gained from introspecting over the services. A typical workflow developed in the systems biology use case is shown in Figure 19.1.

The components of a Scufl workflow are:

- A set of inputs that are entry points for the data for the workflow.
- A set of outputs that are exit points for the data for the workflow.
- A set of processors, each of which represents a logical service — an individual step within a workflow. A processor includes a set of input ports and a set of output ports. From the user's perspective, the behavior of a processor is to receive data on its input ports (processing the data internally) and to produce data on its output ports.
- A set of data links that link data sources to data destinations. The data sources can be inputs or processor output ports, and data destinations can be outputs or processor input ports.
- A set of coordination links that enable running order dependencies to be expressed where direct data flow is not required by providing additional constraints on the behavior of the linked processors. For example, in Figure 19.1, the coordination links are defined so that one processor will not process its data until another processor completes, even though there is no direct data connection between them.

Part of the complexity of workflow design is when the user needs to deal with collections, control structures such as iterations, and error handling. Scufl is simplified to the extent that these are implicit. This layer fills in these implicit assumptions by interpreting an Internal Object Model that encodes the data that passes through a workflow. This data model is lightweight; it contains some basic data structures, such as lists and trees, and enables the decoration of data with MIME types and semantic descriptions to enable later discovery or viewing of the data.

The addition of data structures such as lists to the data object model brings about an added complexity. There are a number of ways in which the list could be handled by the service. Taverna uses an implicit, but configurable, iteration mechanism, as shown in Figure 19.5. Where a processor takes a single list as inputs, the enactment engine will invoke the processor multiple times and collate the results into a new list. Where a processor takes two (or more) list inputs, the service will be invoked with either the cross or dot product of the two lists.

Taverna supports fault tolerance through a configurable mechanism; processors will retry a failed service invocation a number of times, often with increasing delays between retry attempts before finally reporting failure. Users can specify alternative services for any Scufl processor in the order in which they should be substituted. Alternative services are typically either an identical service supplied by an alternative service provider or, rarely, a completely different service that the user deems to be substitutable without damaging the workflow's intention.

While the Scufl language defines the data flow, it does not fully describe the service interactions to enable this data flow.
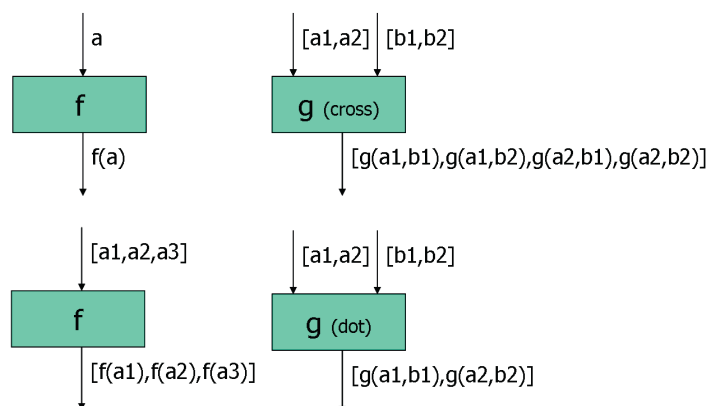
Figure 19.5: Configurable iteration. For example, a processor implements a function f — it takes one input a and produces result $f(a)$. If this processor is given a list of inputs $[a1,a2,a3]$, the implicit iteration will produce a list of results, one for each input. This is equivalent to "map f $[a1,a2,a3]$." Where a processor has more than one input, the default is to apply the function to the cross product of all the input lists, however, sometimes the dot product is required. The configurable iterators allow users to specify how the lists of input values should be combined using these cross and dot operators.

It would be impossible to describe the interaction with all of the different service interfaces within a language like Scufl. Instead, Scufl is designed to be extensible through the use of processor types. We define a set of *processor plug-ins* that manage service interaction by presenting a common abstraction over these different styles. Current processors include:

- A WSDL Scufl processor implemented by a single Web service operation described in a WSDL file.
- A local Java function processor, where services are provided directly through a Java implementation with parameters as input ports and results as output ports (Figure 19.1).
- A Soaplab processor, implemented through a CORBA-like stateful protocol of the Web service operations in a Soaplab service.
- A nested workflow processor, implemented by a Scufl workflow (Figure 19.1).
- A BioMOBY processor (Figure 19.6). Several smaller groups have adopted the BioMOBY project's conventions for publishing Web services. BioMOBY provides a registry and messaging format for bioinformatics services [469].
- A SeqHound processor that manages a representational state transfer (REST) style interface, where all information required for the service

invocation is encoded in a single HTTP GET or POST request (Figure 19.6).

- A BioMart processor that directly accesses predefined queries over a relational database using a JDBC connection (Figure 19.6).
- A Styx processor that executes a workflow subgraph containing streamed services using peer-to-peer data transfer based on the Styx Grid service protocol [357].

The Freefluo engine is responsible for the enactment of the workflow. The core of the engine is workflow language independent, with specific extensions that specialize Freefluo to enable it to enact Scufl.

19.5 Discovering Resources and Designing Workflows

Workflow construction is driven by the domain expert, that is, the scientist. This corresponds to designing a suitable laboratory protocol for their investigation. The life cycle of an *in silico* experiment (see Figure 19.2) has the following stages:

- *Hypothesis formation.* First, the scientist determines the overall intention of the experiment. This informs a top-level design, and would be the overall "shape" of the workflow, including its inputs and desired outputs.
- *Workflow design.* Second, this design is translated into a concrete plan. In the laboratory, this translation would consist of choosing appropriate experimental protocols and conditions. In an e-Science workflow, this maps to the choice and configuration of data and analysis services.
- *Collecting.* The workflow needs to be run, the services invoked, data coordinated, etc (See Section 19.6). In the laboratory, this is handled by protocols for entering results in laboratory books. As the workflow is executed, the results have to be collected and coordinated to record their derivation path. To comply with scientific practice, records need to be kept on where these data came from, when they were acquired, who designed and who ran the workflow, and so forth. This is the provenance of the workflow and is described more fully in Section 19.7.
- *Analyzing and sharing.* As in a laboratory experiment, results are analyzed and then shared.

19.5.1 Service Discovery

In this section, we describe the *service discovery* and *service choice* aspects of running *in silico* experiments in Taverna.

Taverna uses a variety of different mechanisms for discovery of services and populates the service list using an incremental approach. Flexible approaches to discovering available resources are an essential part of supporting the experimental life cycle:

- *Public registries such as UDDI* [430]. We are in favor of registries, but their limited usefulness is due to the lack of widespread deployment. They are generally perceived by the community to be a heavyweight solution [430].
- *GRIMOIRES.* An enriched prototype UDDI registry service developed by ^{my}Grid, with the ability to store semantic metadata about services.
- *URL submission.* Users can add new services by directly pointing to a URL containing WSDL files. The workbench will introspect over the description and add the described services to a palette of services.
- *Workflow introspection.* Users can exploit existing experience by loading existing workflows, observing how services have been used in context, and adding those services to the available services palette.
- *Processor-specific mechanisms.* Many of the service types Taverna supports through its processor plug-ins provide their own methods for service discovery.
- *Scavenging.* Local disks are scavenged for WSDL files that are introspected over, or users create a Web page containing links to service descriptions and, when pointed at this page, Taverna explores all available service descriptions, extracts services, and makes them available. While crude, this works well and gives users considerable flexibility in loading the palette of available services that fits their current requirements.

Taverna's access to 3000 services means that service selection is increasingly important. Figure 19.6 is grouped according to the service locations, which means that services of the same type are grouped together and color coded. In addition, there is a simple search by name facility.

A common task is to locate a new service based on some conceptual description of the service semantics. To enable service selection by bioinformaticians, we must represent their view of the services and domain [480]. We have investigated a number of different mechanisms to drive the search process, including an RDF-based metadata-enriched UDDI registry [269], and a domain ontology [481] described in the W3C Web Ontology Language OWL.

Feta is our third and most recent version of a component for semantically searching for candidate services that takes a user-oriented approach to service discovery [268], a path also being trodden by the BioMOBY project. In practice, this means we describe an abstraction over the services—provided by the Taverna processors—rather than the services themselves. We have relatively shallow descriptions of the services. Although richer descriptions might enable more refined searching and sophisticated reasoning, they are expensive and time consuming to provide. In practice, search results do not have to be precise, as the final choice is made by the workflow designer (a biologist), not automatically by a machine. Finally, the use of shallow descriptions enables us to use simpler technologies to answer queries.
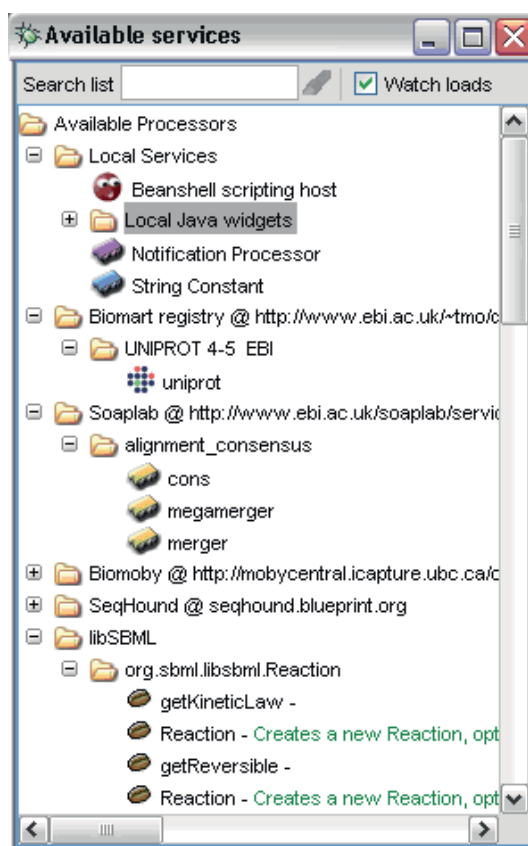
Figure 19.6: An example palette of local (BeanShell scripts, Java widgets) and remote (Biomart, Soaplab, BioMOBY, Seqhound) services that can be used for the construction of workflows in Taverna. libSBML methods made available as local services via the API consumer and that were used for the construction of the exemplar systems biology workflow are also shown.

19.5.2 Service Composition

Most workflow design packages have adopted a view analogous to electric circuit layout, with services represented as "chips" with pins for input and output [20,409]. However, from a user interface point of view, this arrangement can become less understandable as complexity increases. If the layout of service components onscreen is left under the user's control, then the user can tailor the workflow appearance, but this can result in a large amount of time being spent effectively doing graph layout rather than e-Science. In Taverna, the graphical view of a workflow is read-only; it is generated from the underlying workflow model. One advantage of this is that it is easy to

generate different graphical views of the workflow, showing more or less detail as required.

When composing workflows in an open world, we have no control over the data types used by the component services. A service identified by a scientist as being suitable may not use the same type as the preceding service in the workflow, even if the data match at a conceptual level. Consequently, many of the bioinformatics workflows created in Taverna contain numerous "shim" services [202] that reconcile the inevitable type mismatches between autonomous third-party services. We are currently building libraries of shims for dereferencing identifiers, syntax and semantic translation, mapping, parsing, differencing, and so on.

19.6 Executing and Monitoring Workflows

Execution of a workflow is largely an unseen activity, except for monitoring the process and reviewing records of an experimental run (see Section 19.7). A critical requirement of $^{\mathrm{my}}$Grid's service approach is that workflow invocation behavior should be independent of the workflow enactment service used. To facilitate peer review of novel results, it is important that other scientists be able to reproduce *in silico* experiments in their context and verify that their results confirm the reported novel results.

Executing workflows using different enactment services is given less emphasis in business workflows, which will typically be carefully negotiated and agreed by the businesses involved and executed in a fixed, known context. In contrast, a scientific workflow will be shared and evolved by a community and executed by many individual scientists using their favored workflow enactment service.

19.6.1 Reporting

Reporting the progress of a workflow is a complex task. Information about service invocation is unavailable in the general case. Defining how far a service is through a given invocation, so progress can be displayed, is nontrivial without the explicit modeling and monitoring of state. The migration of application services to the Grid's Web Service Resource Framework [100] is a solution that we are investigating.

The reporting mechanism in Taverna is a stream of events for each processing entity, with these events corresponding to state transitions of the service component. For example, a message is emitted when the service is first scheduled, when it has failed for the third time and is waiting to retry, etc. These message streams are collated into an XML document format and the results presented to the user in tabular form as shown in Figure 19.7.

The introduction of reporting in Taverna does not alter the workflow results. What it does alter is users' understanding of what is going on and
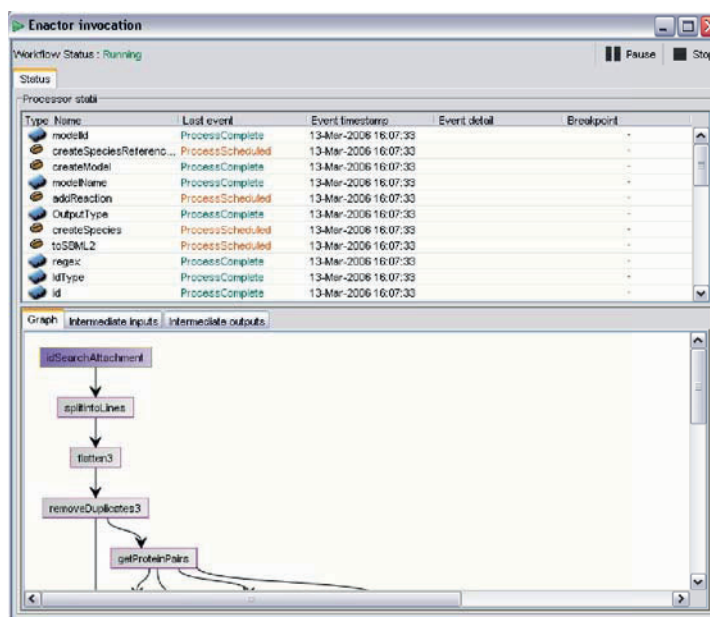
Figure 19.7:   Status information. When running a workflow, the Taverna workbench displays status information from the workflow enactor. For each Scufl processor, the last event is displayed along with the appropriate time and additional detail if available. This additional detail can include progress through an iteration (e.g. "item 2 of 6") and retry information. The status information also allows the selection of a processor and viewing of the relevant intermediate inputs and outputs. Each data item has been assigned a Life Science Identifier (LSID). More detailed trace information is also available using the "Process report" tab.

therefore their confidence that the system is doing what they want. Overall, the feedback from Taverna's initial users was that workflow execution without suitable monitoring was not acceptable. They were willing to accept workflows that occasionally failed; their experience with form-based Web services was that these were unreliable. However, workflow execution could not be a "black-box" service, users need feedback on what is happening, whether the workflow completed successfully or failed, and they need this recorded in logging records.

When a workflow may contain 50 or more processing components (e.g. Scufl processors), and each of these components can be retrying, using alternative implementations, etc., the complete state of a workflow is highly complex. Users require a visualization that allows them to see at a glance what is happening, acquire intermediate results where appropriate, and control the workflow progress manually should that be required.

19.7 Managing and Sharing Workflows and Their Results

As the use of workflows increases the ability to gather and generate data in large quantities, the storage of these data in an organized manner becomes essential for analysis within and between experiments. For scientists, workflows are the means to an end; their primary interest is in the results of experiments. This interest, however, goes beyond examining the results themselves and extends to the context within which those results exist. Specifically, the scientist will wish to know from where a particular result was derived, which key process was used, and what parameters were applied to that process. Thus, in addition to the raw data, we have devised a model of meta data describing the provenance of all aspects of the experiment: the data's derivation path, an audit trail of the services invoked, the context of the workflow, and the evidence of the knowledge outcomes as a result of its execution [494]. Another view is that it is the traditional who, where, when, what, and how questions applied to *in silico* science. These different aspects of provenance can be used for life scientists in different scenarios:

- to repeat a workflow execution by retrieving the "recipe" recorded in the provenance;
- to reproduce a data product by retrieving the intermediate results or inputs from which these data were derived;
- to assess the performance of a service that is invoked in different experiment runs at different times;
- to debug the failure of a workflow run, e.g. which service failed, when and why it failed etc.;
- to analyze the impacts of a service/database update on the experiment results, by comparing the provenance of repeated runs;
- to "smartly" rerun a workflow if a service is updated by using provenance to compute which part of a workflow is required to be rerun as a consequence of the update; and
- to aggregate provenance of a common data product that is produced in multiple runs.

We have adopted two key technologies for provenance collection:

- *Life Science Identifiers.* The description of the derivation of data necessitates reference to the data sets both inside and outside the control of $^{\mathrm{my}}$Grid. Bioinformatics has adopted view standards for the identification of data instead of using an ad hoc system of *accession numbers*. The recent Life Science Identifier (LSID) standard [93] provides a migration path from the legacy accession numbers to an identification scheme based on URIs.
- *Resource Description Framework (RDF).* The Dako data store has a fixed schema that reflects the common entities used in an e-Science experimental

life cycle not tied to any scientific discipline. The use of a fixed schema provides performance benefits. However, RDF's basic graph data model is well suited to the task of representing data derivation. The Knowledge Annotation and Verification of Experiments (KAVE) meta data store has a flexible schema due to its use of RDF. This allows statements to be added outside the fixed schema of the Dako data store, as is needed when providing subject-specific information. KAVE enables other components in ^{my}Grid to store statements about resources and later query those statements.

One can distinguish between provenance of the data and provenance of the process, although the two are linked. The primary task for data provenance is to allow the exploration of results and the determination of the derivation path for the result itself in terms of input data and intermediate results en route to the final value. "Side effect" information about how intermediate and final results have been obtained is generated and stored during workflow invocation. Thus the workflow engine produces not just results but also provenance meta data about those results. Side effect information is anything that could be recorded by some agent observing the workflow invocation, and it implicitly or explicitly links the inputs and outputs of each service operation within the workflow in some meaningful fashion. The associated component RDF Provenance Plug-in listens to the events of workflow execution and stores relevant statements using KAVE; for example, a name for a newly created data item or a meaningful link between the output of a service and the inputs that were used in its creation.

Process provenance is somewhat simpler than data provenance and is similar to traditional event logging. Knowledge provenance is the most advanced and contextual of the meta data results. Often a user does not need to see a full "blow by blow" account of the processes that executed during the workflow or a full account of the complete data-derivation path. Instead they wish to relate data outcomes across a group of processes annotating the relationships between outcomes with more semantically meaningful terms than "derived by." As each such provenance fingerprint is unique to the workflow and the user, a *provenance template* accompanies the Scufl document to be populated by the provenance capture component and stored in the KAVE.

19.8 Related Work

In life sciences there are many scientists who want an easy way of rapidly pulling together third-party services into prototypical *in silico* experiments. This contrasts with fields such as physics and astronomy, where the prime scenario involves carefully designed workflows linking applications to exploit computational Grid resources for *in silico* experiments that were previously impractical due to resource constraints.

Scientific workflow systems vary in terms of their intended scientific scope (the kinds of analyses supported), their technical scope (the kinds of resources that can be composed), their openness to incorporating new services, and whether or not they are open source. The strengths of Taverna are its ability to link together a significant range of autonomous bioinformatics services and its flexibility, particularly in terms of the metadata generated to help manage and share workflow results.

The Kepler workflow system [19, 20] has been developed for ecologists, geologists and biologists and is built on Ptolemy II, a mature application from electrical engineering [366]. Kepler's strengths include its library of Actors, which are mainly local applications, and its suite of Directors that provide flexible control strategies for the composition of Actors. The Triana [409] system was originally developed as a data analysis environment for a gravitational wave detection project. Like Taverna and Kepler, Triana is also data-flow oriented. It is aimed at CPU intensive applications, allowing scientists to compose their local applications and distribute the computation.

DiscoveryNet uses a proprietary workflow engine, and all services are wrapped to conform to a standard tabular data model. DiscoveryNet scientific workflows are used to allow scientists to plan, manage, share, and execute knowledge discovery and data analysis procedures [373]. In the Pegasus system [160], users provide a workflow template and artificial intelligence planning techniques are used to coordinate the execution of applications on a heterogeneous and changing set of computational resources. The emphasis is on the scheduling large numbers of jobs on a computational Grid, where there may be alternative strategies for calculating a user's result set.

The use of workflows for "programming in the large" to compose web services has led to significant interest in a standard workflow language, with BPEL[1] [24] a strong candidate, created through the agreed merge of IBM's WSFL [254] and Microsoft's XLANG [416]. One reason why Taverna workflows use Scufl rather than a potential standard is historical. In the initial stages of the $^{\mathrm{my}}$Grid project in 2001, BPEL did not exist. The more significant reason is conceptual. Initial experiments showed IBM's WSFL language did not match how our target users wanted to describe their *in silico* experiments [7]. WSFL forced users to think in terms of Web service ports and messages rather than passing data between bioservices.

---

[1] BPEL was originally termed BPEL4WS and is being promoted as a standard called WSBPEL through OASIS (Organization for the Advancement of Structured Information Standards), an international consortium for e-business standards.

19.9 Discussion and Future Directions

^{my}Grid set out to build a workflow environment to allow scientists to perform their current bioinformatics tasks in a more explicit, repeatable, and shareable manner:

- *Making tacit procedural knowledge explicit.* For at least the last 250 years, this has been recognized as essential in science. Each experiment must carry with it a detailed "methods" description to allow others both to validate the results and also reuse the experimental method. Our experience suggests that workflows allow this to be achieved for *in silico* experiments. They are formal, precise, and explicit, yet straightforward to explain to others.
- *Ease of automation.* Many of the analyzes we support have already been undertaken by scientists who orchestrate their applications by hand. Workflows can drastically reduce analysis time by automation. For example, Taverna workflows developed by the Williams–Beuren Syndrome team have reduced a manual task that took two weeks to be an automated task that typically takes just over two hours [397].
- *Appropriate level of abstraction.* Bioinformaticians have traditionally automated analyzes through the use of scripting languages such as PERL. These are notoriously difficult to understand, often because they can conflate the high-level orchestration at the application level with low-level "plumbing."

Taverna and the ^{my}Grid suite enables users to rapidly *interoperate* services. It does not support the semantic *integration* of the data outcomes of those services. We underestimated the amount of data integration and visualization provided by the existing Web-delivered applications. They often integrate information from many different analysis tools and provide cross-references to other resources. Accessing the analysis tool directly as a service circumvents this useful functionality. Although the scientist is presented with results in hours, not weeks, it now takes significant time to analyze the large amount of often fragmented results. A solution is complicated by the fact that the workflow environment does not "understand" the data and so cannot perform the data integration necessary. We have provided integration steps within workflows, written as scripts that integrate and render results, but these are specific to each workflow design. We are currently investigating a multi-pronged approach: (i) the use of Semantic Web technology to provide more generic solutions that can be reused between related workflows; (ii) appropriate workflow designs using shims and services under the control of the user to build data objects; and (iii) closing off the open world in situations where the workflows are known to orchestrate a limited number of services and will be permanent in nature, so it is worth the effort to build a more strongly typed model.

Since January 2006, the ^{my}Grid suite, including Taverna 2.0, has moved to a new phase. As part of the United Kindom's Open Middleware Infrastructure Institute (OMII-UK)(`http://www.omii.ac.uk`), ^{my}Grid is to be integrated with a range of Grid services and deployed in a common container with job submission services, monitoring services, and large-scale data management services. Focus is placed on the following:

- *Grid deployment.* Deploying the Taverna architecture within a Grid container, making the enactor a stateful service, and a server-side distributed service, and supporting stateful data repositories.
- *Improved security.* Authentication and authorization management for data, metadata and implementation of credentials for access control of services.
- *Revised execution and processor models.* Support of interactive applications, long running processes, control-based workflows, data flows with large data throughput, enhanced provenance collection, and credential handling. We already have a user interaction service that allows users to participate interactively with workflows.
- *Improved data and metadata management.* Incorporating better user-oriented result viewers and incorporating SRB and OGSA-DAI data implementations.
- *Integration with third-party platforms.* Examples are Toolbus and EGEE. We also plan to continue to interoperate with other workflow systems, specifically Kepler and the ActiveBPEL system emerging from UCL.
- *Extending services.* To execute over more domain services, such as the R suite, and over generic services such as GridSAM job submission.

The field of scientific workflows is rapidly evolving, and as a project in this area ^{my}Grid must also evolve. We engage different user communities (such as biological simulation), and new applications become available, as do novel service frameworks for deploying them. By working closely with our users, service providers, and other workflow projects, we continue to extend the basic core functionality to fulfill a wide range of uses.

Acknowledgments

The Triana Workflow Environment: Architecture and Applications

Ian Taylor, Matthew Shields, Ian Wang, and Andrew Harrison

20.1 Introduction

In this chapter, the Triana workflow environment is described. Triana focuses on supporting services within multiple environments, such as peer-to-peer (P2P) and the Grid, by integrating with various types of *middleware* toolkits. This approach differs from that of the last chapter, which gave an overview of Taverna, a system designed to support scientists using Grid technology to conduct *in silico* experiments in biology. Taverna focuses workflow at the Web services level and addresses concerns of how such services should be presented to its users.

Triana [429] is a workflow environment that consists of an intuitive graphical user interface (GUI) and an underlying subsystem, which allows integration with multiple services and interfaces. The GUI consists of two main sections, as shown in Figure 20.1: a *tool browser*, which employs a conventional file browser interface — the structure representing toolboxes analogous to directories in a standard file browser and the leaves (normally representing files) representing tools; and a *work surface*, which can be used to graphically connect tools to form a data-flow diagram. A user drags a desired tool (or service) from the tool browser, drops it onto the work surface, and connects tools together by dragging from an output port on one tool to an input port on the other, which results in cables being drawn to reflect the resulting data pipeline. Tools can be grouped to create aggregate or compound components (called *Group Units* in Triana) for simplifying the visualization of complex workflows, and groups can contain groups for recursive representation of the workflow.

The underlying subsystem consists of a collection of interfaces that bind to different types of *middleware* and services, including the Grid Application Toolkit (GAT) [13] and, in turn, its multiple bindings to Grid middleware, such as Grid Resource Allocation Manager (GRAM), GridFTP, and GridLab Resource Management System (GRMS); the Grid Application Prototype (GAP) interface [408] and its bindings to JXTA [64], P2PS [460], and
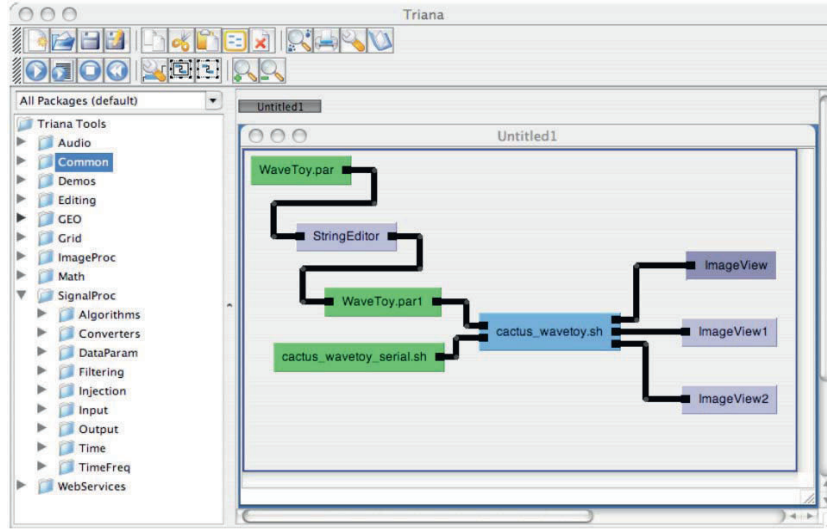
Figure 20.1: A mixed-component Triana workflow consisting of Grid file and job operations through proxies and Java components.

WSPeer [187]; and integration to Web services, WS-RF [100], and OGSA-DAI. The resulting integration means that Triana tools on the work surface can represent any service or primitives exposed by such middleware, and these tools can be interconnected to create mixed-component workflows. An illustration of this is provided in Figure 20.1, where we show a workflow that integrates job and file proxy components that interact with the GAT interface to access job submission (i.e., GRAM) and file transfer operations (GridFTP) and local Java components that provide editing and visualization capabilities. In this example, the local Java components are used to edit a parameter file, which is then staged on the Grid using the file proxy components and specified as an input file for a Grid job submission, which in this case happens to be a Cactus simulation (see Chapter 25). Local Java components are used to visualize the results from this job. Although this example shows the interaction between Grid jobs and local Java units, we have other scenarios that interconnect WS-RF services, P2P services, and local Java units.

In this chapter, we will take a detailed look at the Triana environment and discuss its components for interacting with Grids and P2P networks. We also focus on application examples and describe two specific examples of how workflows are generated, refined, and executed within the environment. The rest of this chapter is organized in the following way. In the next section, we relate Triana to other frameworks described in this book and elsewhere. We then give an overview of the main Triana components and illustrate the types of distributed component interactions that Triana facilitates. In Section

20.5, we discuss the workflow representations Triana uses, and in Section 20.6 how it has been used in a number of different ways by listing some projects that are using Triana and the functionality that they employ. In Sections 20.7 and 20.8, we present two case studies, which illustrate how Triana workflows are generated, modified, and executed within P2P environments and the Grid.

20.2 Relation to Other Frameworks

As we can see from some of the other chapters in this book, the Grid workflow sector is relatively crowded, with a number of different frameworks, languages, and representations for similar concepts. Part of the reason for this is that existing Grid workflow engines are often tied to the technologies employed by their parent projects and are not necessarily able to integrate new technologies effectively. Many of these projects contain elements very similar to Triana, albeit with a different terminology; for example, Triana *tasks* are conceptually the same as Kepler *actors* and Taverna *processors*. The Kepler project (Chapter 7) for supporting scientific workflows is a cross-project collaboration based on the Ptolemy II system [366]. The approach in Kepler/Ptolemy II is very similar to that of Triana in that the workflow is visually constructed from Java components, called actors, which can either be local processes or can invoke remote services such as Web services or a GridFTP transfer.

Taverna (Chapter 19) is a workbench for workflow composition and enactment developed as part of the myGrid [396] project, the focus of which is bioinformatic applications. Originally designed to execute Web service based workflows, Taverna can now interact with arbitrary services. ICENI (Chapter 24) is an environment for constructing applications using a graphical workflow tool together with distributed component repositories on computational Grids. ICENI employs coarser grained components than many of the other environments, generally focusing on large Grid-enabled application components.

The Chimera Virtual Data System (VDS) (Chapter 23) is a system for deriving data rather than generating them explicitly from a workflow. It combines a virtual data catalog, for representing data derivation procedures and derived data, with a virtual data language interpreter that translates user requests into data definition and query operations on the database. The user specifies the desired end result, and a workflow capable of generating that result is derived. If intermediate results are available, then these are used directly rather than being regenerated. Pegasus takes the abstract workflow generated by the Chimera system and maps it onto a Grid. Workflows are expressed in Chimera's Virtual Data Language (VDL) and are converted into Condor's DAGMan format for execution.

The current release of The Globus Alliance's CoG Kit includes a workflow tool called the Karajan Workflow Engine (Chapter 21). The workflow language

Karajan uses is an XML-based scripting language that includes declarative concurrency, support for control structures such as *for...next* and *while* loops, conditionals such as *if...then*, and support for all CoG-supported services, such as GridFTP or Globus job submission. The toolkit comes with a workflow editor for composing Karajan scripts and a workflow engine for executing them. Karajan workflow is aimed specifically at executing jobs in a Grid environment and does not have capabilities for local processes such as those available in Triana's local toolboxes. The main operations with which it concerns itself are job submission and file transfer, and these are represented as nodes in the script. The BPEL4WS (Chapter 14) language is a workflow language for choreographing the interaction between Web services. It is used in many projects in business workflow but is less common in scientific workflow systems.

20.3 Inside The Triana Framework

Triana was initially designed as a quick-look data analysis tool for the GEO 600 project [158] but has been subsequently extended into a number of arenas within the scientific community. Originally, workflows in Triana were constructed from Java tools and executed on the local machine or remotely using RMI. A large suite of over 500 Java tools has been developed, with toolboxes covering problem domains as diverse as signal, image, and audio processing and statistical analysis. More recently, Triana components have evolved into flexible proxies that can represent a number of local and distributed primitives. For example, a Triana unit can represent a Java object, a legacy code, a workflow, a WS-RF, P2P, or Web service, a Grid job, or a local or distributed file.

In essence, Triana is a data-flow system (see Chapter 11) for executing temporal workflows, where cables connecting the units represent the flow of data during execution. Control flow is also supported through special messages that trigger control between units. The cables can be used to represent different functionalities and can provide a convenient method for implementing plug-ins. For example, in our GAT implementation, described in Section 20.4.2, the cables represent GAT invocations, and the content of adjoining units provides the arguments to these calls. Therefore, two connected file units would result in a GAT *fileCopy* invocation, and the actual locations and protocols specified within the units indicate which GAT adapter should be used to make this transfer (e.g., HTTP, GridFTP, and so on).

Triana integrates components and services (see Chapter 12) as Triana units and therefore users visually interact with components that can be connected regardless of their underlying implementation. In a somewhat simplified perspective, Triana *components* are used to specify a part of a system rather than to imply a specific implementation methodology and its obvious object-oriented connotations. Triana components are simply units of

execution with defined interactions, which don't imply any notion of state or defined format for communication.

The representation of a Triana workflow is handled by specific Java reader and writing interfaces, which can be used to support multiple representations through a plug-in mechanism. This means that the actual workflow composition is somewhat independent of workflow language constraints and currently we have implementations for VDL (see Chapter 17) and DAG workflows (see Chapter 22). Such plug-ins can be dynamically activated at runtime, which means that Triana could be used as a translator between such representations to provide syntactic compatibility between systems.

20.4 Distributed Triana Workflows

Triana workflows are comprised of Triana components that accept, process, and output data. A component may be implemented as a Java method call on a local object or as an interface to one of a range of distributed processes or entities such as Grid jobs or Web services. We call components that represent a remote entity *distributed components* without suggesting that the remote entity represented describes itself as a component. These distributed components fall into two categories:

- *Grid-oriented components.* Grid-oriented components represent applications that are executed on the Grid via a Grid resource manager (such as GRAM, GRMS, or Condor/G) and the operations that support these applications, such as file transfer.
- *Service-oriented components.* Service-oriented components represent entities that can be invoked via a network interface, such as Web services or JXTA services.

Triana uses simplified APIs as its base for programming within both service-oriented and Grid-oriented environments. Specifically, the Grid Application Toolkit (GAT) API [13] developed during the GridLab project [175] is used for Grid-oriented components. The GAT is capable of implementing a number of different bindings to different types of middleware, and these can be dynamically switched at runtime to move across heterogeneous Grid environments without changing the application implementation. Section 20.4.2 discusses in detail our core interface to Grid-oriented software toolkits and services using the GAT. For our service-oriented components, we use the Grid Application Prototype (GAP) interface described in the next section. The GAT and GAP interfaces can be used simultaneously within a Triana application instance, enabling users to compose Triana components into workflows that represent elements from both traditional toolkits such as Globus 2.x and Web, WS-RF, or P2P services.

20.4.1 Service-Oriented Components

The Grid Application Prototype Interface (GAP Interface) is a simple interface for advertising and discovering entities within dynamic service-oriented networks. See [408] for a full description of the GAP. Essentially, the GAP uses a P2P-style pipe-based mechanism for communication. The pipe abstraction allows arbitrary protocols to be implemented as bindings to the GAP as long as they can fulfill the basic operations of *publish*, *find*, and *bind*. The GAP currently provides bindings to three different infrastructures:

- *P2PS*. P2PS [460] is lightweight P2P middleware capable of advertisement, discovery, and communication within ad hoc P2P networks. P2PS implements a subset of the functionality of JXTA using the pipe abstraction employed by JXTA but tailored for simplicity, efficiency, and stability.
- *Web services*. This binding allows applications to host and invoke Web services using standard discovery protocols such as UDDI [430] or dynamic P2P oriented discovery mechanisms such as P2PS.
- *JXTA*. JXTA [64] is a set of open protocols for discovery and communication within P2P networks. Originally developed by Sun Microsystems, JXTA is aimed at enabling any connected device, from a PDA to a server, to communicate in a P2P manner.

The GAP abstracts away the implementation detail of the various bindings. For example, service description takes different forms in the existing bindings — Web services use Web Service Definition Language (WSDL) [482], JXTA uses *service descriptors*, and P2PS simply uses named pipes. Likewise, transport and transfer protocols vary — Web services usually use HTTP over TCP/IP, while JXTA and P2PS are transport agnostic, allowing communication to traverse different protocols via the pipe abstraction. These peculiarities do not filter up through the GAP. From a user's perspective, a service is simply made available that provides some capability and can be invoked via the GAP. Furthermore, the use of the GAP as an umbrella to differing service-oriented infrastructures means that it is possible to seamlessly use applications developed on top of the GAP Interface across different networks just by switching the GAP binding used.

The most common GAP binding we use is the Web service binding. This is largely because of its support for Grid-based security, currently via the Grid Security Infrastructure (GSI), and because of the confluence of Web and Grid services, which means many Grid service interfaces are now being defined using Web service standards.

Web Service Integration

The GAP Web service binding is implemented using WSPeer [187]. WSPeer is focused on enabling simple, lightweight Web service management and does not

require the usual infrastructure associated with service hosting, such as a Web server and a service container. Furthermore, it allows an application to expose functionality as a Web service *on the fly*. As a result, WSPeer can operate under diverse conditions, making its binding to a P2P-oriented interface, such as the GAP, a straightforward task.
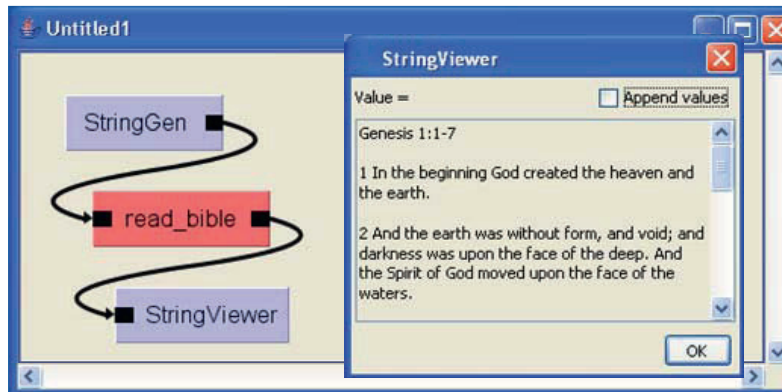
Figure 20.2: Web service and local tools on the Triana desktop.

From the perspective of Triana, the GAP enables diverse service infrastructures to be viewed in a common way. By wrapping a GAP service as a Triana component, the service is made available to the graphical workspace displaying optional input and output ports. Connections are drawn between components with cables, which usually denote data streams. On the workspace, local tools, remote services, and Grid jobs can coexist and be connected to one another. Figure 20.2 shows a combination of local Java tools interacting with a remote Web service. The local tools provide a means for inputting data into and reading output from the service component. The example in Figure 20.2 shows a simple string generator tool that passes Bible book, chapter, and verse information to the service, `read_bible`. The service returns the text from the specified section of the Bible, which is displayed using a simple string viewer tool. From the user's perspective, there is no difference between the components — they are simply visual components on the workspace.

WS-RF Integration

Triana interacts with its distributed resources by using the GAT and the GAP interfaces, which draw a clear distinction between Grid-based

and service-oriented interactions. This distinction divides our distributed interactions between simple application-level interfaces, like the GAT, where clear standardization efforts are currently under way (e.g., the SAGA GGF Research Group [374]) and service-based interfaces. Such a distinction, however, may well become less pronounced as service orientation is more widely adopted by Grid middleware in general.

Therefore, from a Grid service perspective, WSPeer also incorporates Web Service Resource Framework (WS-RF) [100] capabilities that enable Triana to handle stateful resources via services as well as employ the event-driven notification patterns supported by WS-Notification [316].

The WS-RF suite of specifications is based on the concept of a WS-Resource [319]. This is the combination of a resource identifier and an endpoint to a Web service that understands the identifier and can map it to some resource. The resource can be anything — a table in a database, a job, a subscription to a published topic, or a membership in a group of services. The aim of WS-RF is to allow this underlying resource to be made accessible and potentially be modified across multiple message exchanges with a Web service without associating the service itself with the state of the resource. In practice, this is achieved by placing the WS-Resource, serialized as a WS-Addressing [184] EndpointReference, into the header of the Simple Object Access Protocol (SOAP) message. Queries for properties of the underlying resource are implicitly mapped to the resource referenced in the WS-Resource. A WS-RF service advertises the *type* of resource it can handle through a schema document in the WSDL definition of the service. This schema document describes the properties (keys) that the resource type exposes and that can therefore be accessed or modified. When a client is in possession of a WS-Resource, it uses the properties keys declared in the WSDL to retrieve the associated values. These values in turn represent the state of the resource.

Although the underlying infrastructure to manage WS-RF and WS-Notification message exchange patterns is quite complex, Triana makes the process simple from a user's perspective. A WS-RF service can be discovered and imported in the same way ordinary Web services are. When a WS-RF service arrives in the user's toolbox, it is made up of the usual Web service operations that can be dragged onto the Triana worktop to be invoked. However, Triana allows an additional *context* to be associated with these WS-RF service operations in the workflow through a simple GUI. This *context* is not itself a WS-Resource, but the name associated at workflow design time with a WS-Resource that will be created or imported into the workflow at runtime. As WS-RF does not specify the mechanism for how a WS-Resource is created or returned to a client, it is impossible to write an all-purpose tool for creating/importing WS-Resources within a Triana workflow. A typical approach, however, is to employ a factory service. In this case, the factory service can become part of the workflow, feeding the WS-Resource into the context that is used as part of the invocation of a WS-RF service.

WS-RF Workflow

The application of WS-RF compliant services to workflows opens up certain possibilities. In particular, the WS-Resource construct can be used to reduce the need for sending large data sets as SOAP attachments or, worse, encoded as XML. Because the use of WS-Resources allows arbitrary resources to be exposed via a Web service, this can also pertain to data generated by a service (that is, output), allowing a service to return a WS-Resource to a service requester, as opposed to actual data. As a simple example, one can imagine an executable that has been wrapped as a Web service, that takes a file as input, and outputs another file after execution. Using standard Web services mechanisms, one could imagine this service with an operation that takes a byte array, or a SOAP attachment, as input and returns some similar data structure as output. In the case of large files, this can be expensive, especially if the file is being returned to the workflow enactment engine merely to be sent to the next node in the workflow thereafter. If this service is WS-RF compliant, however, then it can return a WS-Resource exposing the file as a resource and itself as the service from which properties of the file can be retrieved. There are a number of ways clients could be given access to the file resource; for example, the resource type may expose a property consisting of a URI that can be connected to directly, in order to read from a data stream. This is far more efficient that transferring data along with the XML and also allows the data to be pulled when (if) needed. If we extend this model to service operation inputs, then it allows us to create workflows in which references to data are passed directly between workflow components, bypassing the need to send data via the enactment engine. Further optimizations can be achieved by defining the properties of the file resource to reflect application-specific requirements. For example, certain services may only need to process parts of the file, in which case a property is exposed that returns just the relevant portion.

From a more general perspective, the widespread adoption of the WS-Addressing specification and the EndpointReference structure is potentially useful in terms of workflow. Although not fully standardized as yet, the use of WS-Addressing could pave the way for a generic means for services to reference each other — similar to the *anchor* tag in HTML — even services that are not specifically Web services. This in turn could lead to mechanisms for describing and deploying heterogeneous workflows — the kind of workflows Triana is capable of building — which are autonomous, running independently of continual controller intervention. Currently, workflows involving arbitrary services still require control and data to pass through the enactment engine at every stage of the workflow because there are no universally accepted means of transferring control or data directly to the next process in the flow.

While the widespread adoption of WS-Addressing should be considered a positive, its use is not always suitable for all situations. In fact, this criticism can be leveled at WS-RF. By combining an endpoint address with a resource

identifier, one is tightly coupling a service with a resource. This model can lead to an object-oriented approach in which WS-Resources are used as global pointers to specific resources located at certain addresses. Furthermore, it encourages the explicit modeling of entities that should be hidden behind the service interface. Both these conditions can lead to complex and fragile systems [261]. Specifically in the context of workflow, managing references that are explicitly tied to service endpoints can become cumbersome as the number of services involved grows. As a result, we are exploring other Web service frameworks, such as the Web Services Composite Application Framework (WS-CAF) [317] and the WS-Context [67] specification in particular, for the generation and enactment of Web service based workflows. WS-Context does not couple state with service endpoints. Instead, context is shared between multiple parties as a stateful conversation, and the interpretation of the context by individual services is left to the invisible implementation of the service.

Web service specification is a rapidly evolving area of development, making it almost impossible to develop code with confidence that it will have any longevity. In fact, we believe it is unlikely that WS-RF will survive in any meaningful form beyond 2007, although some of its ideas may be subsumed into other emerging standards. However, the experience gained with implementing it has left both WSPeer and Triana with flexible architectures for handling contextual message information in general, making them well suited for easily integrating new Web service specifications quickly.

20.4.2 Grid-Oriented Components

Components supporting the execution of code on Grid resources are provided within Triana using the GridLab GAT. The GridLab GAT is a simple API for accessing Grid services and resources. It enables applications to perform Grid tasks such as job submission and file transfer while remaining independent of the Grid middleware used to execute these tasks. The GridLab GAT employs an adapter-based architecture that allows different Grid middleware bindings to be plugged into the GAT, thereby enabling applications written to the GAT API to operate over a range of current and future Grid technologies. The application programmer also benefits from only having to learn a single Grid API, an idea currently being developed further through the SAGA Research Group [374].

At the core of the GAT is the concept of a job, the execution of code on a computational resource. The resource used to execute a job can be local or remote, depending on the GAT adapter used to create the job instance. As essential as job execution is the ability to interact with and relocate files, for example to prestage files in the execution directory of a job and to retrieve output files from a job. Different protocols for accessing and moving files, for example GridFTP and HTTP, can be handled via different GAT adapter instances.

The Visual GAT is the representation of GridLab GAT primitives as components within Triana workflows and the visualization of the data are dependencies between these components. The key Visual GAT components are:

- *Job component.* A job component represents the submission of a GAT job description to a resource broker. This job description includes information on the executable and arguments to be run, plus optional information such as the resource on which the job should be executed.
- *File component.* A file component represents a GAT-accessible file. The file is identified by a URI, which specifies the protocol used to access that file and its network location.

As with standard Triana components, the cables linking Visual GAT components represent data flow between those components. The semantics of this data flow depend on the context of the linked components. For example, the cable between two file components represents data flow from one file location to another; in other words, a file copy operation. Similarly, a cable from a file component to a job component indicates a prestaged file, and a cable from a non-Visual GAT component to a file component indicates a file write.

In Figure 20.1 we show a simple job submission workflow using a mixture of Visual GAT and standard Triana components. In this workflow, local Java components are used to create and view the data, while Visual GAT components are used to represent the prestaging and poststaging of these data and job submission. An equivalent workflow could be created without Visual GAT components; for example, by having specific GridFTP and Globus job submission components. However, although this approach is used within most visual workflow environments, the resulting workflow less accurately models the data flow between workflow components. Furthermore, non-Visual GAT workflows are often more complex and contain more redundancy than equivalent workflows employing Visual GAT components. These issues are discussed in much greater depth in [407].

20.5 Workflow Representation and Generation

A *component* in Triana is the unit of execution. It is the smallest granularity of work that can be executed and typically consists of a single algorithm, process, or service. Component structure in Triana, in common with many component-based systems such as the CCA, has a number of properties such as an identifying name, input and output "ports," a number of optional name/value parameters, and a proxy/reference to the part of the component that will actually be doing the work. In Triana, each component has a definition encoded in XML that specifies the name, input/output specifications, and parameters. The format is similar to WSDL [482], although

more succinct. These definitions are used to represent instance information about a component within the workflow language and component repositories. An example component definition can be seen below.

```
<tool>
    <name>Tangent</name>
    <description>Tangent of the input data</description>
    <inportnum>1</inportnum>
    <outportnum>1</outportnum>
    <input>
        <type>triana.types.GraphType</type>
        <type>triana.types.Const</type>
    </input>
    <output>...</output>
    <parameters>
        <param name="normPhaseReal" value="0.0"
            type="userAccessible"/>
        <param name="normPhaseImag" value="0.0"
            type="userAccessible"/>
    </parameters>
</tool>
```

The external representation of a Triana workflow is a simple XML document consisting of the individual participating component specifications and a list of parent/child relationships representing the connections. Hierarchical groupings are allowed, with subcomponents consisting of a number of assembled components and connections. A simple example taskgraph consisting of just two components can be seen below.

```
<tool>
    <toolname>taskgraph</toolname>
    <tasks>
        <task>
            <toolname>Sqrt</toolname>
            <package>Math.Functions</package>
            <inportnum>1</inportnum>
            <outportnum>1</outportnum>
            <input>
                <type>triana.types.GraphType</type>
                <type>triana.types.Const</type>
            </input>
            <output>...</output>
            <parameters>
            </parameters>
        </task>
        <task>
            <toolname>Cosine</toolname>
            <package>Math.Functions</package>
            ....
```

```
        </task>
        <connections>
            <connection>
                <source taskname="Cosine" node="0" />
                <target taskname="Sqrt" node="0" />
            </connection>
        </connections>
    </tasks>
</tool>
```

Triana can use other external workflow language representations, such as VDL, that are available through "pluggable" language readers and writers. These external workflow representations are mapped to Triana's internal object representation for execution by Triana. As long as a suitable mapping is available, the external representation will largely be a matter of preference until a standards-based workflow language has been agreed upon. Triana's XML language is not dissimilar to those used by other projects such as ICENI [153], Taverna/FreeFluo [326], and Ptolemy II [366] and should be interoperable.

A major difference between the Triana workflow language and other languages, such as BPEL4WS, is that our language has no explicit support for control constructs. Loops and execution branching in Triana are handled by specific components; i.e., Triana has a specific loop component that controls repeated execution over a subworkflow and a logical component that controls workflow branching. We believe that this approach is both simpler and more flexible in that it allows for a finer-grained degree of control over these constructs than can be achieved with a simple XML representation. Explicit support for constraint-based loops, such as *while* or an optimization loop, is often needed in scientific workflows but very difficult to represent. A more complicated programming language style representation would allow this but at the cost of ease-of-use considerations.

20.6 Current Triana Applications

This section outlines some of the projects currently using Triana and its related technologies, such as GAP Interface and P2PS. Triana itself is currently being developed as part of the GridOneD project.[1] The GridOneD project is in its second phase of funding. The initial focus of GridOneD was to develop components within Triana to support gravitational wave searches in collaboration with the GEO600 project [158], and this led to the development of GAP, P2PS, and other middleware. The second phase aims to extend Triana's support for gravitational wave searches and also to develop support for pulsar searches in collaboration with Manchester University and Jodrell

---

[1] `http://www.gridoned.org/`.

Bank. This support will employ Visual GAT components within Triana to submit data-analysis jobs across Grid resources.

Triana and its related technologies are being used in a range of external projects. The majority of these projects are using Triana to choreograph Web services. An example of this is Biodiversity World (Chapter 6), a collaboration between Cardiff, Reading, and Southampton universities and the Natural History Museum. The goal of Biodiversity World is to create a Grid-based problem-solving environment for collaborative exploration and analysis of global biodiversity patterns. Triana is providing the visual interface for connecting and enacting the services created by this project. Other examples of projects using Triana to choreograph Web services include Data Mining Grid [107], a project developing tools and services for deploying data-mining applications on the Grid; FAEHIM [8], a second data-mining-based project; and DIPSO [119], an environment for distributed, complex problem solving.

In terms of related technologies, the DARRT (Distributed Audio Rendering using Triana) project [105] at the Louisiana Center for Arts and Technology is exploring the use of Grid computing technologies towards sound computation and music synthesis, in particular using P2P workflow distribution within Triana. The SRSS (Scalable Robust Self-organizing Sensor networks) project [393] has been using the GAP and P2PS in simulating P2P networks within NS2 for researching lightweight discovery mechanisms. Triana is also being used for workflow generation and editing within the GENIUS Grid portal [157], part of the EGEE project.

20.7 Example 1: Distributing GAP Services

The GAP is an interface to a number of distributed services (e.g. P2PS, JXTA, WS-RF, or Web services). Services can be choreographed into Triana workflows for managing the control or data flow and dependencies between distributed services. However, Triana can also be used to locate and utilize a number of distributed *Triana service deployers* by using a distribution policy that enables the dynamic rewiring of the taskgraph at runtime in order to connect to these services. We have implemented two such distribution policies for parallel and pipelined execution. In both scenarios, on the client, a set of Triana units is selected and *grouped* to create a compound unit, and a distribution policy is applied to this group. In the parallel scenario, the subworkflow contained within the group is distributed across all available service deployers in order to duplicate that group capability across the resources. When data arrive they are farmed out to the various distributed services for parallel execution. In the pipelined scenario, the taskgraph is spliced vertically and parts of the group are distributed across the available resources.

These scenarios are based on the P2P-style discovery mechanisms that are exposed by the GAP interface, with implementations of these mechanisms

provided by the different GAP bindings. These scenarios can therefore work over WS-RF and Web services in the same way as for P2PS, as described in Section 20.4.1. We have used this mechanism in a number of scenarios [91, 406, 410, 411] for high-throughput applications, typically on local networks or clusters where we have control of the resources. Each application generally has a fixed set of data that are input into a group unit, which implements a data-processing algorithm, perhaps for searching a parameter space. Typically, the algorithms are CPU intensive and the parameter sets being searched can be divided and sent to parallel instances of the algorithm. We use the parallel distribution policy to discover and distribute the data to available resources for processing.

20.7.1 Workflow Generation

For these types of service-based scenarios, workflows are typically constructed from local units (Java or C) representing the algorithm for importing the data and for performing the parameter search. Such workflows are constructed and prototyped in a serial fashion and then distributed at runtime. The serial version of these algorithms can be complex. In one example [91], a template-matching algorithm for matching inspiral binaries in gravitational wave signals was constructed from more than fifty local Java units with a number of processing pipelines consisting of specific algorithms (e.g., FFT, correlation, complex conjugate, etc.) that were combined and processed further to give the desired result.

Once the algorithm is composed, the user can visually select the *processing* (CPU-intensive) section of the workflow and group it. This group can then be assigned the parallel distribution policy to indicate that it should be task-farmed to available resources. When data arrive at the group unit, they are passed out across the network to the discovered distributed services one data segment at a time. In this way, the individual services can process data segments in parallel.

20.7.2 Workflow Refinement

In this case, workflows are mapped from their locally specified serial version into a distributed workflow that connects to the available resources. This workflow refinement happens at runtime after the client has discovered the available services it can utilize. The workflow is annotated with proxy components to represent the available distributed services, and the workflow is rewired to direct data to these components. This results in the connectivity to the single local group being replaced by one-to-many connectivity from the client to the available remote services.

20.7.3 Workflow Execution

The distributed workflow created during the dynamic refinement process is used by the execution engine in order to be aware of the available services it can use during the execution phase of the workflow. The current algorithm simply passes the data out in parallel to the services and thereafter it passes data to services once they have completed their current data segment. This ensures a simple load-balancing mechanism during execution.

20.8 Example 2: The Visual GAT

In this section, we outline how Triana can be used to implement complex Grid workflows that combine the GridLab GAT capabilities discussed in Section 20.4.2 with interactive legacy application monitoring (using *gridMonSteer*, described in Section 20.8.1). The two scenarios presented below illustrate a fundamental shift in the perception of how legacy applications can be run on the Grid in that the workflow in each example as a whole is the Grid application rather than a monolithic legacy code. The legacy code is typically deployed multiple times within the workflow to conduct parameter sweeps or similar actions, and we allow interactive control in the wider context of the complete workflow.

In the examples presented in this section, we employ the use of a wrapper for integrating distributed legacy applications into complex *Visual GAT* workflows. In these workflows, decisions are made based on the current output state of the legacy application to steer the workflow (or application) to support the appropriate analysis required.

20.8.1 Integrating Legacy Applications

We have implemented a simple, nonintrusive legacy code or application wrapper, called *gridMonSteer* (GMS), which allows us to integrate noncustomized distributed applications within a workflow. GMS monitors the legacy application as it is executing and further allows application and/or workflow-level steering. GMS emerged from an ongoing collaboration investigating the integration of distributed Cactus simulations [167] (Chapter 25 within Triana workflows. Initially, a Grid-friendly Cactus thorn was developed to provide the distributed connectivity from Cactus to a Triana workflow component [168]. This component detected files output by Cactus and passed these into a running Triana workflow, which was used to visualize the simulation as it progressed (this was demonstrated in SuperComputing 2004). GMS is a generalization of this architecture, that allows the same kind of file detection for any application rather than one that is Cactus-specific.

GMS consists of an *application wrapper* that executes a legacy application and monitors specified directories for files that it creates and an *application*

controller, which is any application that exposes a defined Web service interface, enabling it to receive input from one or more application wrappers. The controller, in our case Triana, uses the dynamic deployment capabilities of WSPeer to expose a Web service interface that implements the gridMonSteer protocol for notification and delivery of the distributed files. The wrapper notifies the controller about new files that have been detected. The controller then selects files of interest and returns this list to the wrapper. Thereafter, the wrapper sends these files if and when they are rewritten or updated by the legacy application to the controller. Within the context of the Grid, the wrapper is typically the job submitted to the resource manager, with the executable of the actual legacy application that will be monitored being an argument of this job. Once started, the wrapper executes the legacy application and begins monitoring; for example, in the case of output files, it polls the output directory of the legacy application.

Communication between the wrapper and controller is always initiated by the wrapper. In other words, the controller plays the role of server by opening a listening port and the wrapper that of client in that it opens a per-request outgoing connection, thereby circumventing many NAT and firewall problems that exist within Grid environments. The principal benefit of the gridMonSteer architecture is that the wrapper executes in the same directory as the legacy application, allowing it to constantly monitor the application output files and immediately notify the controller of changes to these files. This approach allows the controller to monitor and respond to intermediate results for the legacy application in a timely manner not possible with other coarse-grained wrapping architectures, such as GEMCLA [229] and SOAPLab [381].

The next two sections describe a brief overview of the two scenarios that use GMS to integrate Cactus within a Triana workflow via the Visual GAT job submission component, described in Section 20.4.2. The breakdown of the process is illustrated through its generation, refinement and execution steps, described in Sections 20.8.4 – 20.8.6.

20.8.2 Executing and Monitoring Dynamic Legacy Applications

This first example was the final project review for the GridLab project. It demonstrated a wrapped GMS Cactus job that was executed within a Triana workflow. Triana was used to stage the files onto the Grid as input to the job, coordinate the job submission, and then interact with the running simulation by visualizing the results and steering it accordingly. The full demonstration is illustrated in Figure 20.3 and is described at length in [407].

Briefly, the scenario involves the following. The *WaveToy_medium.par*, represented using a Visual GAT file component, specifies the location of a Cactus parameter file from a Web server by using an HTTP address. This unit is connected to a local Java component, which results in the HTTP adapter being invoked by the GAT to make the HTTP-to-local transfer. (Conceptually, the GAT invocation is made at the cable level when both protocols on each
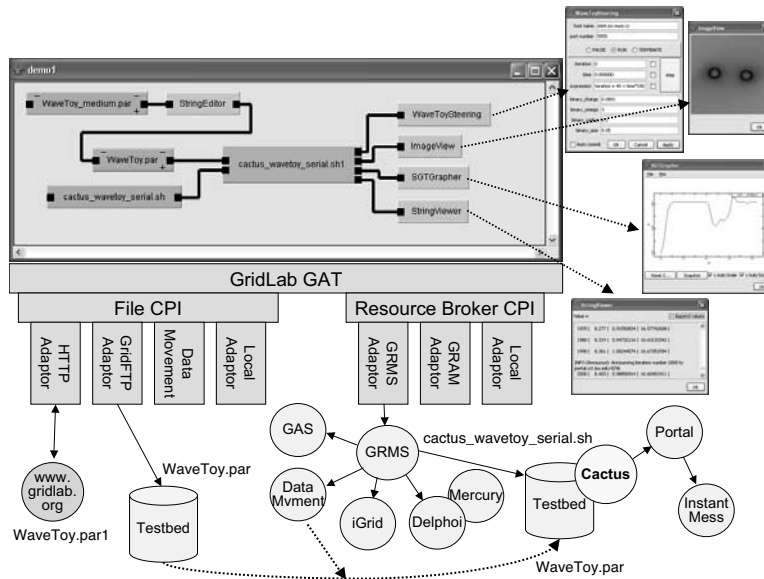
Figure 20.3: Graphical Grid programming through monitoring and steering a deployed Cactus simulation running.

side are defined.) The string editor unit displays this file for minor editing and then passes the contents to another GAT file component that represents a Grid-accessible location (a *gsiftp* address) for the parameter file. The data flow that results used the GridFTP GAT to write the file to a machine in Amsterdam. This file represents the first of two file dependencies for the Cactus job that is specified in the *cactus_wavetoy_serial.sh1* job component. The second dependency is the script that starts Cactus on the remote machine, *cactus_wavetoy_serial.sh*.

The job component in the GridLab review used the GRMS adapter to allow it to make the decision about where the actual job was run. This involved a number of other Gridlab services, including the GridLab Authentication Service (GAS), iGrid, Delphoi, and Mercury, and the GridLab Data Management service, resulting in the *WaveToy.par* and *cactus_wavetoy_serial.sh* files being copied into the location that GRMS chooses to execute the job, as illustrated in Figure 20.3. During execution, we used a custom unit, called *WaveToySteering*, to interact with the Cactus HTTP steering mechanism for changing run-time parameters. We visualized the output files gathered by GMS using the Triana *ImageViewer* tool to display the JPEG files of the simulation, the *SGTGrapher* Triana tool for viewing the wave amplitude of the signal against time from the live Cactus

simulation as it progressed, and the *StringViewer* tool to display standard output (*stdout*) from the simulation.

20.8.3 Dynamic Data-Dependent Simulations

Building from this simple scenario, we are currently in the process of defining more complex Cactus–Triana scenarios that adapt during execution depending on the analyses of data within the workflow. One scenario we are currently implementing involves monitoring a single Cactus simulation much like the scenario above, but instead of steering this Cactus simulation directly, we would monitor its data to watch out for specific features, such as an apparent horizon from a coalescing black hole binary system. Upon such a detection, rather than steering the application directly, we make a decision based on the stimuli or evolution of the application and dynamically instantiate a workflow to aid in the further investigation of this aspect.

Since this typically involves searching a parameter space, we want to perform multiple parallel job submissions of Cacti across the Grid of available resources to perform a distributed search across the parameter range. This could be implemented by dynamically writing a Visual GAT workflow to submit a number of Cacti across the Grid. When the Cacti finish their individual runs, they return the results to the main application, which enables it to steer the main Cactus simulation in the optimal direction and to visualize the results.

20.8.4 Workflow Generation

In both of these cases, the workflows can be specified graphically using simple Visual GAT primitives. In each case, the initial workflows are quite simple and hide the complexity of the multiple levels of refinement that can happen during execution.

20.8.5 Workflow Refinement

In both of these scenarios, Triana can refine the workflows in a number of different ways. In the first case, there are two levels of refinement, which were outlined during the scenario. The first involves converting the abstract Visual GAT workflow into a set of invocations that are appropriate for deployment. We described two mechanisms in the scenario for file transfer and job submission. The virtual GAT invocations result in runtime level refinement by dynamically choosing the appropriate Grid tool for the capability. So, for HTTP-to-local file transfer, an HTTP file adapter was used, and for Grid staging, a GridFTP file adapter was used. Similarly, for job submission, GRMS was chosen for its discovery and resource brokering capabilities.

A second-level refinement was made at application steering by allowing the location of the simulation to be dynamically fed into the *WaveToySteering*

unit, which could, in turn, tune the parameters in the simulation. The application-level refinement allows a user to alter the behavior of the simulation which in the case of the first scenario can result in different internal workflows taking place. In the second scenario, however, this is more apparent. Here, the result from one simulation is used to drive the workflow as a whole. The initial workflow is simple, but as events are detected, more workflows are spawned to analyze these events further and are then fed back into the workflow in order to steer the Cactus simulation.

20.8.6 Workflow Execution

The execution of both of these workflows uses the underlying GAT engine to coordinate the execution of the components and stage the files for the necessary transfer. Triana simply acts as a graphical interface to this underlying engine for the distributed functionality connecting these stages to the default local scheduler for execution of the local units where appropriate. Triana can also mix and match distributed services, local units, and GAT constructs and therefore acts as a manager or a bridge between the different engines for execution of the components.

20.9 Conclusion

In this chapter, we described the Triana workflow environment, which is capable of acting in heterogeneous Grid and P2P environments simultaneously. This is accomplished through the use of two lightweight application-level interfaces, called the GAP and the GAT, that allow integration with distributed services and Grid capabilities. The underlying bindings for these interfaces allow interaction through the GAP to JXTA, P2PS, and WSPeer (with its integration to Web services and WS-RF) and through the GAT to a host of Grid tools, such as GRAM, GridFTP, and GRMS. We described each of these bindings and outlined the underlying workflow language on which Triana is based. Finally, we presented two service-based and Grid-based examples to show how the workflow is generated, refined, and executed in each case.

20.10 Acknowledgments

Java CoG Kit Workflow

Gregor von Laszewski, Mihael Hategan, and Deepti Kodeboyina

21.1 Introduction

In order to satisfy the need for sophisticated experiment and simulation management solutions for the scientific user community, various frameworks must be provided. Such frameworks include APIs, services, templates, patterns, GUIs, command-line tools, and workflow systems that are specifically addressed towards the goal of assisting in the complex process of experiment and simulation management. Workflow by itself is just one of the ingredients for a successful experiment and simulation management tool.

The Java CoG Kit provides an extensive framework that helps in the creation of process management frameworks for Grid and non-Grid resource environments. Hence, process management in the Java CoG Kit can be defined using a Java API providing task sets, queues, graphs, and direct acyclic graphs (DAGs). An alternate solution is provided in a parallel extensible scripting language with an XML syntax (a native syntax is also simultaneously supported). Visualization and monitoring interfaces are provided for both solutions, with plans for developing more sophisticated but simple-to-use editors. However, in this chapter we will mostly focus on our workflow solutions. The Java CoG Kit workflow solutions are developed around an abstract, high-level, asynchronous task library that integrates the common Grid tasks: job submission, file transfer, and file operations.

The chapter is structured as follows. First, we provide an overview of the Java CoG Kit and its evolution, which led to an integrated approach to Grid computing. We present the task abstractions library, which is necessary for a flexible Grid workflow system. Next, we provide an overview of the different workflow solutions that are supported by the Java CoG Kit. Our main section focuses on only one of these solutions, in the form of a parallel scripting language that supports an XML syntax for easy integration with other tools, as well as a native, more human-oriented syntax. Additionally, a workflow repository of components is also presented, which allows sharing of workflows between multiple participants and dynamic modification of

workflows. We exemplify the use of the workflow system with a simple, conceptual application. We conclude the chapter with ongoing research activities.

21.1.1 Overview of the Java CoG Kit

One of the goals of the Java Commodity Grid (CoG) Kit is to allow Grid users, Grid application developers, and Grid administrators to easily use, program, and administer grids from a high-level framework. The Java CoG Kit leverages the portability and availability of a wide range of libraries associated with the Java framework, while promoting easy and rapid Grid application development. The Java CoG Kit started with the development of a client-side and partial server-side implementation of the classic Globus (Globus Toolkit 2.x) libraries under the name of "jglobus." Today jglobus includes, among other libraries, Java implementations of the Grid Security Infrastructure (GSI) libraries, GridFTP, myProxy, and GRAM. The jglobus library is a core component of both Globus Toolkit 3 and Globus Toolkit 4 and a major contribution of the Java CoG Kit to the Grid effort.

Today, the Java CoG Kit provides rich concepts and functionality to support process management that goes beyond that of the Globus Toolkit. One of the concepts that has proven to be useful in protecting the user from frequent changes in the standards development is the concept of abstractions and providers. Through simple abstractions, we have built a layer on top of the Grid middleware that satisfies many users by giving them access to functions such as file transfer or job submission. These functions hide much of the internal complexity present within the Grid middleware. Furthermore, it projects the ability to reuse commodity protocols and services for process execution and file transfer instead of only relying on Grid protocols. In order to integrate new services, all a developer has to do is define a relatively simple set of providers that follow a standard interface definition. In addition to the abstraction and provider concept, the Java CoG Kit also provides user-friendly graphical tools, workflows, and support for portal developers.

Hence, the Java CoG Kit integrates a variety of concepts to address the needs posed by the development of a flexible Grid upperware toolkit as depicted in Figure 21.1. End users will be able to access the Grid through standalone applications, a desktop, or a portal. Command-line tools allow users to define workflow scripts easily. Programming is achieved through services, abstractions, APIs, and workflows. Additionally, we integrate commodity tools, protocols, approaches, and methodologies, while accessing the Grid through commodity technologies and Grid toolkits. Through this integrated approach, the Java CoG Kit provides significant enhancements to the Globus Toolkit. Hence, the Java CoG Kit provides a much needed add-on functionality to Grid developers and users while focusing on the integration of Grid patterns through the availability of a toolkit targeted for the development of Grid upperware.
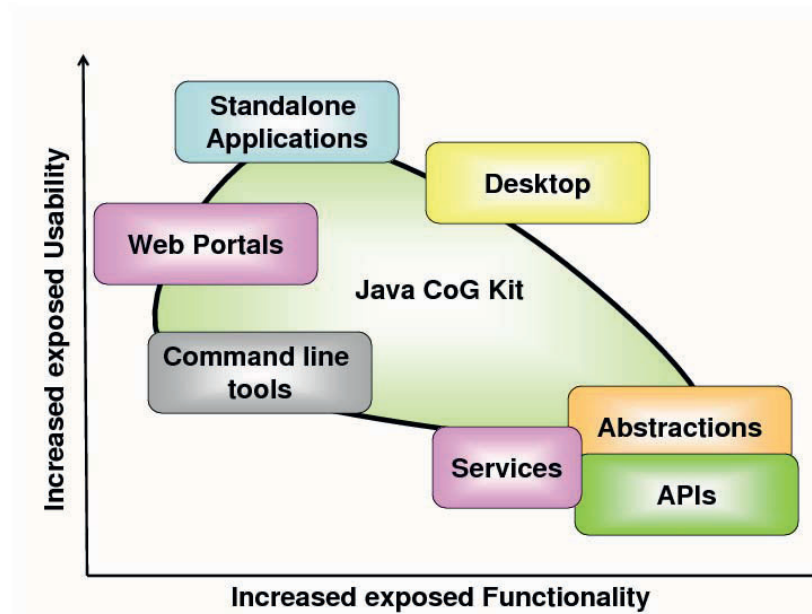
Figure 21.1: An integrated approach.

21.1.2 Abstractions and Providers

The architecture of the Java CoG Kit is derived from a layered module concept that allows easier maintenance and bridges the gap between applications and the Grid middleware 21.2. It allows for the easy integration of enhancements developed by the community. One of the strengths of the toolkit is that it is based on the abstractions and providers model.

We have identified a number of useful basic and advanced abstractions that help in the development of Grid applications. These abstractions include job executions, file transfers, workflow abstractions, and job queues and can be used by higher-level abstractions for rapid prototyping. As the Java CoG Kit is extensible, users can include their own abstractions and enhance its functionality.

We introduced the concept of Grid providers that allow a variety of Grid middleware to be integrated into the Java CoG Kit. The abstractions allow the developer to choose at runtime where Grid middleware services tasks related to job submission and file transfer will be submitted. This capability is enabled through customized dynamic class loading, thus facilitating late binding against an existing production Grid.
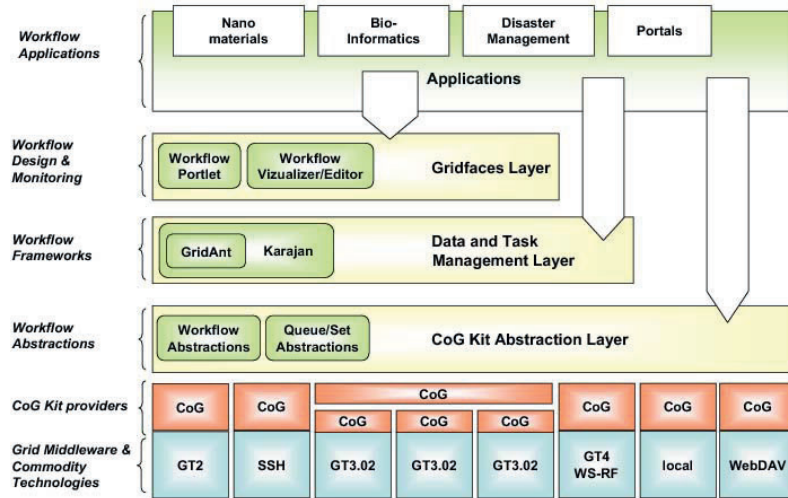
Figure 21.2: A layered architecture.

21.1.3 Workflow Concepts in the Java CoG Kit

The origin of all workflow concepts of the Java CoG Kit, including the adaptive workflow framework is based on the work described in [443], which defines a dynamically adapting scheduling algorithm that chooses optimized algorithms based on performance measurements to identify a set of resources that fulfill a given high-level function such as a matrix multiplication. The task of conducting the matrix multiplication is specified at the time the workflow is specified. However, its instantiation and the appropriate choice of algorithm are conducted during runtime. Other important evolutionary steps are the development of GECCO [448], which included dynamic fault detections and a workflow debugger; and GridAnt [446], which uses the commodity tool Ant to manage Grid tasks.

Today, the Java CoG Kit contains a number of workflow concepts that have been shaped by our long collaborations with experimental scientists [444, 445]. We evolved several concepts as part of the Java CoG Kit workflow framework. These concepts include (a) abstractions for queuing systems and workflow graphs with simple dependencies, as found in [443]; (b) event-based notifications and monitoring as found in [443, 447, 448, 452]; (c) elementary faulttolerant features; (d) a simple execution pattern [448], now termed *cog*

pattern; (e) hierarchical graphs [448]; (f) structured control flow with loops and conditions [449]; (g) visual interfaces to augment the workflow execution [443]; (h) an adaptive workflow framework; and (i) workflow component repositories.

21.1.4 Lessons Learned from GridAnt

The Apache Ant project presents certain characteristics that seem to make it suitable as a workflow language and engine. Features such as native dependency structured build targets make it easy to write declarative dependency-based workflows.

Ant is designed around the concept of targets and tasks. Targets exist only as top-level entities. Dependencies between targets are specified using target names as handles. Targets in Ant are executed sequentially, without the possibility to exploit the parallelism inherent in a dependency-based specification. Targets are composed of tasks that are generally executed sequentially. Parallel execution of tasks can be achieved with the `<parallel>` task. It executes its nested tasks in parallel, synchronously. Globally scoped immutable properties can be used to provide value abstractions. Conditional execution is achieved at the target level based on property values. Iterations are not possible in Ant.

GridAnt, invented by Gregor von Laszewski, is an extension to the Apache Ant build system, which adds the following features:

- the `<gridExecute>` and `<gridTransfer>` tasks, allowing job submission and file transfers using the Java CoG Kit abstractions API;
- a `<gridAuthenticate>` task, which launches a GSI proxy certificate initialization window;
- a generic progress viewer, which can visualize target dependencies and track the state of the execution of each target (Figure 21.3); and
- Partial iteration capabilities. Full support for iterations featuring iteration variables was impossible due to the immutable nature of Ant properties, an aspect that was deeply ingrained into multiple areas of the Ant engine.

The use of the Ant engine posed the following problems that limited the possibility of implementing complex workflows:

- Inability to concurrently execute targets.
- Lack of full iteration support.
- Difficulties in expressing conditional execution.
- Scalability limitations in parallelism, due to the extensive use of native threads, leading to memory exhaustion when running workflows describing large numbers of parallel tasks.
- Difficulties in the ability to consistently capture the state of the execution of a particular workflow, leading to an inability to add checkpointing and resuming abilities to Ant.
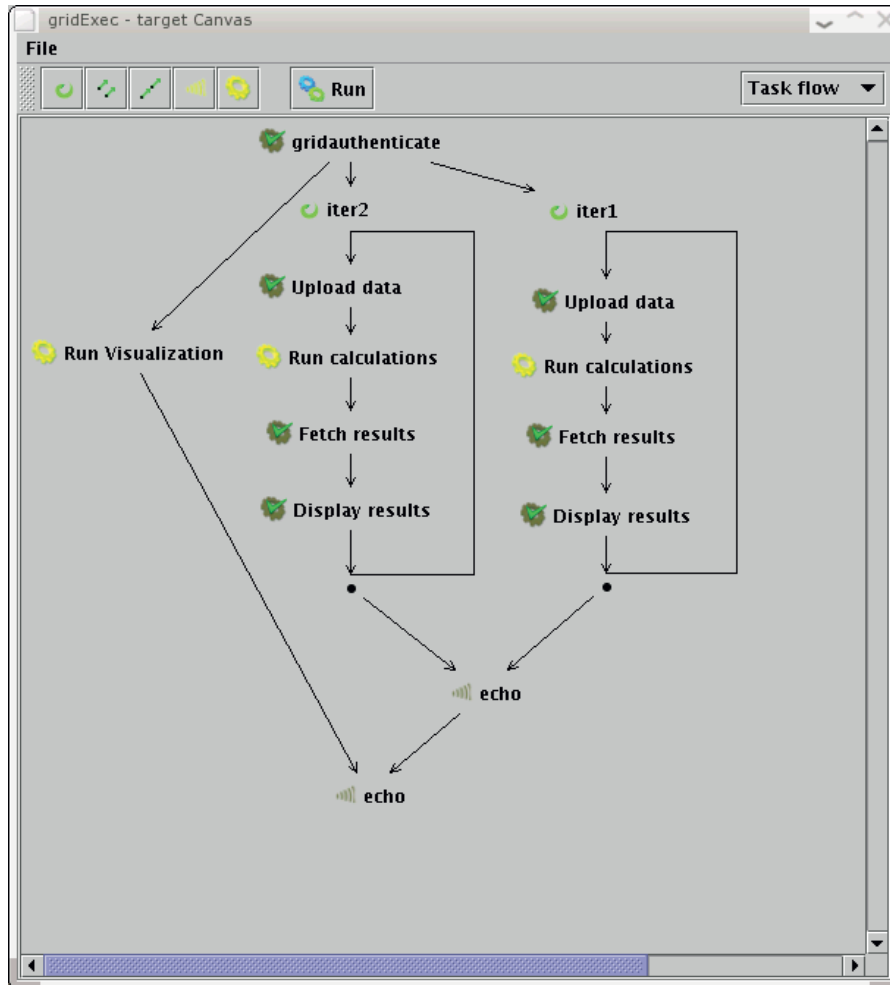
Figure 21.3: GridAnt viewer.

- The authors of Ant favored verbose specifications and made abstractions, parameterized execution, or self-contained extensibility difficult.

All of these disadvantages motivated us to develop a more streamlined and powerful workflow framework.

21.2 The Java CoG Kit Karajan Workflow Framework

Karajan was designed and implemented when it became apparent that the shortcoming of GridAnt could not be avoided without a complete re-design of

the engine. It provides a cleaner and clearer separation between its composing parts that have been specifically designed to address the shortcomings of our previous systems. The name "Karajan" originates from the name of a famous conductor of the Berlin symphony orchestra. However, it does not yet contain some of the features we provided in earlier workflow systems [443,448], notably a workflow engine dealing with workflow congestion and elementary debugging support. However, for many scientific applications the current version of the Java CoG Kit Karajan framework will be sufficient.

21.2.1 Architecture

The architecture of the Java CoG Kit Karajan framework is displayed in Figure 21.4. It contains the workflow engine that interfaces with high-level components, namely a visualization component that provides a visual representation of the workflow structure and allows monitoring of the execution, a checkpointing subsystem that allows the checkpointing of the current state of the workflow, and a workflow service that allows the execution of workflows on behalf of a user. A number of convenience libraries enable the workflow engine to access specific functionalities such as a task library to enable access to Grid services, a forms library to enable the dynamic creation of forms as part of workflow tasks, a Java library to extend the workflow language with elements based on Java classes, and a core library that includes convenience abstractions used within the workflow engine.

The language structure specification is designed carefully, so it can be syntactically formulated in two ways. One possibility is to use an XML-based syntax that has its origin from GridAnt but is significantly enhanced with features such as control structures. The other way is based on the desire to have more simplified syntax for scripting that includes such features as the replacement of XML begin and end tags with simple brackets. This syntax is significantly shorter than the XML syntax and provides the script designer with a rapid prototyping mechanism. The languages can be transformed into each other.

The workflow execution engine employs a lightweight threading in order to support large-scale workflows efficiently.

The philosophy of Karajan is based on the definition of hierarchical workflow components. However, instead of just supporting direct acyclic graphs (DAGs), a much more powerful internal implementation is provided that is also reflected within the language structure. Hence we provide primitives for generic sequential and parallel execution, sequential and parallel iterations, conditional execution and functional abstraction. At the same time, we provide support for common data types, such as lists and maps, that are specifically targeted to support parameter studies.

The Grid interface is enabled with the help of the Java CoG Kit abstractions API that we introduced earlier. Through the use of the provider concept, which provides a mechanism to interact with tasks by defining specific
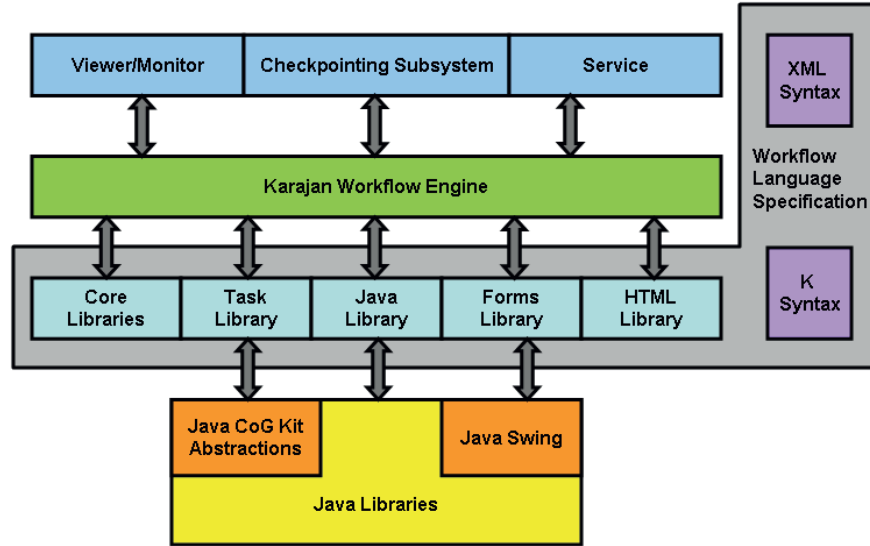
Figure 21.4: The components of the Java CoG Kit Karajan module build a sophisticated workflow system.

task handlers for different Grid middleware and services, the decision of how a particular task is to be executed can be deferred until the task is mapped onto a specific resource during runtime. This makes it possible to focus on the definition of tasks while deferring the actual instantiation and mapping of the component once onto a resource during runtime. The actual mapping can be performed through the use of a simple scheduler example that is included within the Karajan framework. This example also demonstrates that it will be easy to integrate user-defined scheduling algorithms and make the Karajan workflow framework an ideal candidate for enhancements through the user community.

Based on this flexibility, the Karajan workflows can provide interoperability between multiple grids. One of the fundamental problems of the Grid is that through deployment variations we cannot assume that the same Grid middleware version is available everywhere. However, with the Java CoG Kit and the Karajan workflow, we can formulate workflows despite the fact that the underlying resources use different versions of Grid services and standards. Consequently, interoperability is considered an elementary necessary feature of Grid workflows.

Karajan provides user-directed and global fault tolerance. Through user-directed fault tolerance, special library elements can be employed to ignore faults, restart faulting blocks, trap faults, and provide individual actions, or specify dynamically scoped error handlers. At the global level, timed or program-directed checkpointing can be used.

One of the important differences from other workflow frameworks is that Karajan can be extended both through parameterized user-defined workflow elements (functions) and/or by implementing new workflow elements in Java.

21.2.2 Language Design

The Karajan language is a declarative style language, but with strict evaluation.

Variables can be used in Karajan, but the scoping rules restrict the possibility of concurrent destructive updates. Each execution element, which is similar to a function, has its own scope, both for its arguments and its body. Variables defined in parent scopes can be read if they fall within the same lexical scope but are not written to. Attempting to write to a variable will create a new binding for that variable, which will shadow a variable with the same name in any parent scope. Furthermore, parallel elements will create separate scopes for each concurrent thread.

Iteration elements can be used to execute a given set of elements sequentially or in parallel for a given set of items. Iterations can be used for both actions and data. From a data-centered perspective, iterations are equivalent to multiple issues of the same parameterized data. Unrolling iterations manually while consecutively replacing the iteration variable with the respective values produces the same result as using iterations. The direct consequence is that Karajan elements support multiple return values by default.

Karajan supports parameterized abstractions, similar to function definitions. However, Karajan provides extended functionality in terms of concurrency for workflow element definitions. Besides strict evaluation in which all arguments are evaluated before the element is invoked, it is possible to define workflow elements that evaluate the arguments in parallel with the body of the definition. If an argument that is needed by the body thread is not yet evaluated, the body thread suspends execution waiting for the argument thread to evaluate the particular argument. The parallel element evaluation can achieve a result similar to the use of generators in other languages. Nonetheless, generators require that special semantics be used in the definition of generators, while the Karajan parallel element allows any other function to be used as a generator (in part due to the multiple return values that are natural in Karajan).

Dataflow equivalence is provided between sequential elements and their parallel counterparts. Values are returned in lexical order, regardless of the order in which they are evaluated. It is, however, also possible to use versions of the parallel primitives which return values in the exact order in which they are evaluated.

Due to their nature and structure, the parallel composition elements in Karajan provide and promote locality with respect to concurrency. Combined with the recursive structure of the language, this allows for concurrent threads

to be expressed declaratively. Concurrency thus becomes an aspect of the program rather than a separate entity.

A number of other helpful concurrent constructs are also available. Futures can be used to represent the result of a computation that is not yet available. Syntactically identical to variables, futures will cause the execution of the thread that attempts to access their value to be suspended until the value is available. Channels can be used as future lists. Similar to futures, values in channels can be accessed as they become available, otherwise causing the accessing thread to be suspended.

21.2.3 Execution Engine

The execution engine supports lightweight threading, which provides concurrency scalability, with less impact on resources than is the case with native threads. The engine does not excel in terms of performance. Nonetheless, the overall impact on Grid workflows is little since the limitations are caused mostly by the security libraries, most of the overhead CPU time being spent during authentication, data encryption/decryption, and signature verification in the Grid libraries.

The lightweight threading engine, given the same resources, allows somewhere in the range of two orders of magnitude more concurrent threads when compared with the use of native threading. The engine uses a mix of cooperative and preemptive multithreading. By default, a small number of worker threads are started and used to execute workflows. When the existing worker threads become blocked executing lengthy operations, new threads are progressively added up to a certain limit, in order to keep the latency low. The cooperative threading relies on the asynchronous implementation of time-consuming operations, in particular the Grid tasks.

The language specification provides a recursive definition of a set of transformations on data and actions (sideeffects) — workflow elements. The execution of a Karajan workflow consists of the execution of a root element that receives an initial data environment. The root element in turn executes all its subelements, which will recursively execute their subelements and so on. Elements do not have an internal state. The state is maintained in the data environment, which is continuously moved around and transformed. Elements can create private spaces in the data environment, which are maintained for as long as an element and its sub-elements complete execution. Parallel execution is accompanied by the creation of new environments. Since the new environments only share data that are private to parent elements, the concurrent execution of elements cannot cause concurrent destructive writing into the environment. This ensures the consistency of low-level program data when concurrent execution is involved.

The execution engine also allows the state to be checkpointed, either automatically at preconfigured time intervals or manually at specific points in the workflow. The state of a workflow consists of all the currently executing

workflow elements and the data environment on which they are executing. This information alone is sufficient to restore the execution at a later time if severe failures occur (loss of power, hardware failure, and others).

21.2.4 Task Library

In Karajan, the task library provides the main means of interfacing with Grid middleware. The task library is built on top of the Java CoG Kit abstractions API. As the scope of the abstractions API was discussed earlier, we will focus on the functionality provided by the task library.

The binding of tasks to resources and protocols can either be done explicitly or delegated to a scheduler. Sometimes it is necessary to separate tasks that can be executed on any resources that provide certain capabilities from tasks that must be executed on specific resources. Therefore a mix of explicit and scheduled tasks can also be employed. For example, a workflow may involve fetching data from a given resource, processing it as quickly as possible on all available computation resources, and then moving the resulting data on to another predefined resource. Using purely abstract or purely concrete workflows may prevent the ability to express such a workflow.

Schedulers in the task library can be configured with a set of accepted protocols and a set of resources (although schedulers that dynamically fetch resource information are also possible but not currently implemented). Tasks that are then not explicitly bound to a specific resource or protocol are forwarded to the scheduler, which assigns them resources and protocols based on policies. An unbound (abstract) task is composed from a type ($type$) and a specification ($spec$): $T_u = (type, spec)$. The type describes the type of task: execution, file transfer, information query,[1] or file operation. A bound task is a task associated with a resource (r) and a protocol (p): $T_b = (type, spec, r, p)$. Resources can support zero or more protocols for each service type: $r = \{s | s = (type, p)\}$. Assuming that only one service of the same type and protocol exists for every resource, then the pair $(type, r, p)$ uniquely identifies a service for a resource. Consequently, given a bound task $(type, spec, r, p)$ and a resource r defined above, the task is unambiguously defined. The duty of the task scheduler is to maintain load information for resources and produce bound tasks from unbound tasks $S : R, (type, spec) \rightarrow (type, spec, r, p)$, where R is the set of all resources available to the scheduler.

Additionally, there might exist the need to group several tasks on the same resource. A mechanism exists in the task library to request that the scheduler to supply a virtual resource that can be used to partially bind tasks for the purpose of indicating that they must be scheduled on the same resource. The virtual resource allocations can be nested if grouping on more than one resource is needed at the same time. As an example, suppose that a

---

[1] Due to the many changes in Globus MDS, the Information queries are not yet implemented.

job must be executed somewhere, and two resulting files must be transferred on another, unique machine. Without grouping, there would be no guarantee that the transfers would have the source files on the same machine as the one on which the job is executed or that the transfers would have the same machine as destination.

The task library, through the Java CoG Kit abstractions API, uses an asynchronous task execution mechanism, which minimizes the number of resources (threads) created by the execution of tasks. Combined with the lightweight threading engine of Karajan, it allows for higher scalability than synchronous, native thread-based implementations.

21.2.5 The Service

The Karajan service is designed to accomplish the task of writing distributed Karajan workflows.

The service works together with a *remote* library element in order to provide a mechanism through which parts of Karajan programs can be detached from the current interpreter and sent to the service for execution, while preserving (most) of the semantics of the language.

Built around a flexible communication layer, the service allows configuration of the mode in which remote invocations are handled. It is possible to configure, on a host or domain basis, whether persistent connections, callback connections, or polling is to be used. Such configuration is intended to allow control between performance and resource usage, and not the least, the ability to use the service from behind a firewall. The current implementation is built on top of a GSI/SSL transport mechanism, allowing GSI authentication and data privacy and encryption.

Two major modes of operation are supported:

- *Personal.* In personal mode, the service is bound to a unique GSI identity. Once authenticated, a user has unrestricted access to all Karajan libraries.
- *Shared.* The shared mode is designed for multiple users. Authorization is done using Globus Gridmap files (a simple form of an access control list). Tight security restrictions are placed on various aspects of the workflow. Only authorized data types are permitted, and certain library functions are not available, in order to prevent the possibility of privilege escalation. In shared mode, a special local provider can be used, enabling Grid mapped job submission and possibly other operations.

The service allows the use of both remote and local libraries and abstractions. The use of local libraries enables a workflow to reuse libraries that are not part of the service distribution, while the use of remote libraries may allow system-customized interfaces to local resources. With remote libraries, system administrators can implement system-dependent functionality and expose a common interface, allowing workflows to be written in a configuration-independent fashion.

21.2.6 Examples

In Figure 21.5, we present a simple workflow that concurrently executes two jobs and transfers their output to the client machine. It makes use of the scheduling capabilities of Karajan. The `<parallel>` element executes its subelements (in this case the two `<allocateHost>` elements) in parallel and waits for their completion. The `<allocateHost>` element allows the grouping of tasks on a single host, represented by variables `one` and `two`, respectively. It executes its sub-elements in sequential order. The `<task:execute>` and `<task:transfer>` elements interface with the CoG Kit abstraction library in order to provide job submission and file-transfer capabilities. The duty of finding the appropriate services for submitting the execution and transfer requests is left to the scheduler.

The scheduler and resource definition file are presented in Figure 21.6. In this particular case, the resources used are composed of two hosts, each with an execution and a file-transfer service.

```
<karajan>
    <import file="cogkit.xml"/>
    <import file="scheduler.xml"/>
    <parallel>
        <allocateHost name="one">
           <task:execute  executable="/bin/example"
                          stdout="example1.out" host="{one}"/>
           <task:transfer srchost="{one}" srcfile="example1.out"
                          desthost="localhost"/>
        </allocateHost>
        <allocateHost name="two">
            <task:execute  executable="/bin/example"
                           stdout="example1.out" host="{two}"/>
           <task:transfer srchost="{two}" srcfile="example1.out"
                          desthost="localhost"/>
        </allocateHost>
    </parallel>
</karajan>
```

Figure 21.5: A simple workflow that uses a simple scheduler defined in Fig. 21.6.

21.2.7 Repository

The Workflow component repository [450] is a service used to store, retrieve, and search for components that can be integrated into a Java CoG Kit workflow. The repository service promotes reusability of components that can

```
<karajan>
    <scheduler type="default">
        <resources>
            <host name="host1.example.org">
                <service type="execution" provider="gt2"
                        uri="host1.example.org"/>
                <service type="file-transfer" provider="gsiftp"
                        uri="host1.example.org"/>
            </host>
            <host name="host2.example.org">
                <service type="execution" provider="gt2"
                        uri="host2.example.org"/>
                <service type="file-transfer" provider="gsiftp"
                        uri="host2.example.org"/>
            </host>
        </resources>
    </scheduler>
</karajan>
```

Figure 21.6: A scheduler and resource-definition example that is reused in Figure 21.5.

be maintained either by an individual researcher or by a shared community of peers with similar interests.

The aim in designing a workflow repository was to dynamically include workflow components or provide the ability to modify the components while a workflow is in progress. Remote access to the repository is also an important consideration in order to utilize the components of the workflow system in a collaborative environment by providing remote workflow component storage and access to distributed group members. The repository subsystem is still in the first stage of development and only provides indispensable features. It enables persistence for workflows that are executed by storing them either at a local embedded repository or a remote repository based on the user's preference. The components within the repository have metadata associated with them. Versioning and timestamps of a workflow component can be used to distinguish between the components modified over time. Independent user groups may create and maintain their own repositories, which contain components with related information. However, when these groups pool their resources with groups from other domains of science, categories or namespaces are used for distinction. Provenance information for components will in the future guide the selection of components.

The Java CoG Kit Karajan workflow framework allows the dynamic inclusion of workflow components through the use of an include statement. The include statement fetches the component from the repository and evaluates the contents at runtime. Components include a number of attributes

and are defined through a simple XML specification. These attributes are name, short description, description, license, author, code, signature, version, date entered, and date modified. Additionally it is possible to add user-defined attributes. The predefined attributes allow provenance of the component information.

The repository architecture and design follow those of the abstraction and provider model defined and promoted within the Java CoG Kit. Hence it is possible to use a variety of data stores to host such a repository by developing different providers. Sharing of the repository can be enabled by starting up a network server. In this mode, the repository can operate as a centralized component share system for a group of users.

We chose to implement a provider for relational databases specifically based on Apache Derby [25]. A provider based on a relational data store has the advantage of well-defined transaction management, regular backup of data, a built-in server for remote access, user management, and the possibility of replication of the component data in a distributed database. It is foreseeable that providers for other implementations including object-oriented databases and XML databases could be developed. One may ask why it is not enough to provide one solution? The answer is based on the ease of deployment of such a service in a wide variety of user communities. Setting up and maintaining a relational database or a WebDAV server is typically more difficult than simply providing a flat file system. As our tool serves also as middleware, we want to give Grid architects the freedom to choose or develop a provider that plugs into the overall architectural design of their Grid application. It also makes it possible to address scalability issues.

The repository provides these obvious functionalities to its users: loading, saving, listing, searching, and retrieving workflow components in the form of an XML file along with the metadata. Other functions involve definition of new attributes (metadata) for the components, removal of attributes for the components, listing current attributes and user management for the repository. Besides the access provided to the repository API, a command-line tool exists to interface with the repository in a straight forward fashion through a UNIX-like command.

The example shown in Figure 21.7 shows how to integrate a predefined workflow component called "gaussian" into a Karajan workflow. Here, the program "gaussian," which is used to compute thermochemical values in the chemistry domain, is internally called using the element here to invoke the command on a remote computing server. To do so, an input file is transferred to the remote server and executed, and the output is copied back to the local system.

There are a number of predefined elements that can be used to access the repository via the workflow. These constitute the base repository library and are stored in "repository.xml." Once this file is included in a workflow, we can use the appropriate repository functions that have been defined in the library which in turn call the repository API using Java elements provided

by Karajan. One such is "repository:get," which retrieves a component from the repository located at "dblocation". The "provider" is a local embedded database that is created and maintained using Apache Derby, and the component is stored in the file as indicated by "filename."

```
<karajan>
    <include file="cogkit.xml"/>
    <include file="repository.xml"/>
    <include>
        <repository:get component="gaussianChem"
provider="local:derby"
                        dblocation="/home/Admin/repositoryext"/>
    </include>
    <task:transfer srcfile="H2O_B3SP.gjf"
                   destfile="H2O_B3SP.gjf"
                   desthost="hot.mcs.anl.gov">
    <gaussian inputFile="H2O_B3SP.gjf"
              nodes=2
              checkpoint file="H2O.chk"
              host="hot.mcs.anl.gov">
    <task:transfer srcfile="H2O_B3SP.log" srchost="hot.mcs.anl.gov"
                   destfile="gaussian.out">
</karajan>
```

Figure 21.7: An example that demonstrates the inclusion of an element called *gaussian* that is defined in a workflow repository.

21.3 Workflow Support for Experiment Management

The Java CoG Kit group has also prototyped a tool that introduces a framework for experiment management that simplifies the user's interaction with Grid environments. We have developed a service that allows the individual scientist to manage a large number of tasks, as is typically found in experiment management. Our service includes the ability to conduct application state notifications. Similar to the definition of standard output and standard error, we have defined standard status, which allows us to conduct application status notifications. We have tested our tool with a large number of long-running experiments and shown its usability [453].

21.4 Conclusion

In this chapter, we introduced a subset of frameworks that provide experiment management support within the Java CoG Kit. We have focused explicitly

on workflow solutions and motivated the Karajan workflow framework. The framework can be used to specify workflows through a sophisticated XML scripting language as well as an equivalent more user-friendly language that we termed K. In contrast to other systems we not only support hierarchical workflows based on DAGs but also have the ability to use control structures such as if, while, and parallel in order to express easy concurrency. The language itself is extensible through defining elements, and through simple data structures it allows easy specification of parameter studies. The workflows can be visualized through our simple visualization engine, which also allows monitoring of state changes of the workflow in real time. The workflows can actually be modified during runtime through two mechanisms. The first is through the definition of elements that can be deposited in a workflow repository that gets called during runtime. The second is through the specification of schedulers that allow the dynamic association of resources to tasks. The execution of the workflows can either be conducted through the instantiation of a workflow on the users client or can be executed on behalf of the user on a service. This service will in the future also allow a more distributed execution model with loosely coupled networks of workflow engines. Hence, the execution of independent workflows acting as agents for users will be one of our focus areas. Furthermore, we will extend our workflow visualizer to integrate components stored in the repository to enable a dynamically extensible workflow composition tool. We have, however, put great emphasize on the fact that the workflow can also be started from the command line and we will provide future enhancements. At present we have demonstrated with our tool that we can successfully start tens of thousands of jobs due to our scalability-oriented threading mechanisms that are part of the Karajan core engine. We will be using this engine to do modeling of the management of urban water distribution systems [451].

Acknowledgement

22

Workflow Management in Condor

Peter Couvares, Tevfik Kosar, Alain Roy, Jeff Weber, and Kent Wenger

22.1 Introduction

The Condor project began in 1988 and has evolved into a feature-rich batch
system that targets high-throughput computing; that is, Condor ([262], [414])
focuses on providing reliable access to computing over long periods of time
instead of highly tuned, high-performance computing for short periods of time
or a small number of applications.

Many Condor users have not only long-running jobs but complex sequences
of jobs, or workflows, that they wish to run. In the late 1990s, we began
development of DAGMan (Directed Acyclic Graph Manager), which allows
users to submit large workflows to Condor. As with Condor, the focus has been
on reliability. DAGMan has a simple interface that allows many, but certainly
not all, types of workflows to be expressed. We have found through years
of experience running production workflows with our users that solving the
"simple" problems can be surprisingly complex. In the first half of this chapter,
we therefore provide a conceptual (and almost chronological) development of
DAGMan to illustrate the complexities that arise in running workflows in
production environments.

In the past several years, Condor has expanded its focus from running jobs
on local clusters of computers (or *pools* in Condor terminology) to running jobs
in distributed Grid environments. Along with the additional complexities in
running jobs came greater challenges in transferring data to and from the job
execution sites. We have developed Stork [244], which treats data placement
with the same concern that Condor treats job execution.

With a combination of DAGMan, Condor, and Stork, users can create
large, complex workflows that reliably "get the job done" in a Grid
environment. In the rest of this chapter, we explore DAGMan and Stork
(Condor has been covered in detail elsewhere).

22.2 DAGMan Design Principles

The goal of DAGMan is to automate the submission and management of complex workflows involving many jobs, with a focus on reliability and fault tolerance in the face of a variety of errors. Workflow management includes not only job submission and monitoring but job preparation, cleanup, throttling, retry, and other actions necessary to ensure the good health of important workflows. Note that DAGMan addresses workflow execution but does not directly address workflow generation or workflow refinement. Instead, DAGMan is an underlying execution environment that can be used by higher-level workflow systems that perform generation and refinement.

DAGMan attempts to overcome or work around as many execution errors as possible, and in the face of errors it cannot overcome, it endeavors to allow the user to resolve the problem manually and then resume the workflow from the point where it last left off. This can be thought of as a "checkpointing" of the workflow, just as some batch systems provide checkpointing of jobs.

Notably, the majority of DAGMan's features—and even some of its specific semantics—were not originally envisioned but rather are the product of years of collaboration with active users. The experience gained from the needs and problems of production science applications has driven most DAGMan development over the past six years.

The fundamental design principles of DAGMan are as follows:

- DAGMan sits as a layer "above" the batch system in the software stack. DAGMan utilizes the batch system's standard API and logs in order to submit, query, and manipulate jobs, and does not directly interact with jobs independently.[1]
- DAGMan reads the logs of the underlying batch system to follow the status of submitted jobs rather than invoking interactive tools or service APIs. Reliance on simpler, file-based I/O allows DAGMan's own implementation to be simpler, more scalable and reliable across many platforms, and therefore more robust. For example, if DAGMan has crashed while the underlying batch system continues to run jobs, DAGMan can recover its state upon restart and there is no concern about missing callbacks or gathering information if the batch system is temporarily unavailable: It is all in the log file.

---

[1] Note that DAGMan assumes the batch system guarantees that it will not "lose" jobs after they have been successfully submitted. Currently, if the job is lost by the batch system after being successfully submitted by DAGMan, DAGMan will wait indefinitely for the status of the job in the queue to change. An explicit query for the status of submitted jobs (as opposed to waiting for the batch system to record job status changes) may be necessary to address this. Also, if a job languishes in the queue forever, DAGMan currently is not able to "timeout" and remove the job and mark it as failed. When removing jobs, detecting and responding to the failure of a remove operation (leaving a job "stuck" in the queue) is an interesting question.

- DAGMan has no persistent state of its own—its runtime state is built entirely from its input files and from the information gleaned by reading logs provided by the batch system about the history of the jobs it has submitted.

22.3 DAGMan Details

22.3.1 DAGMan Basics

DAGMan allows users to express job dependencies as arbitrary *directed acyclic graphs*, or DAGs. In the simplest case, DAGMan can be used to ensure that two jobs execute sequentially—for example, that job B is not submitted until job A has completed successfully.

Like all graphs, a DAGMan DAG consists of nodes and arcs. Each node represents a single instance of a batch job to be executed, and each arc represents the execution order to be enforced between two nodes. Unlike more complex systems such as those discussed in [476], arcs merely indicate the order in which the jobs must run.

If an arc points from node N_p to N_c, we say that N_p is the *parent* of N_c and N_c is the *child* of N_p (see Figure 22.1). A parent node must complete successfully before any of its child nodes can be started. Note that each node can have any whole number of parents or children (including zero). DAGMan does not require that DAGs be fully connected.

Why does DAGMan require a directed acyclic graph instead of an arbitrary graph? The graph is directed in order to express the sequence in which jobs must run. Likewise, the graph is acyclic to ensure that DAGMan will not run indefinitely. In practice, we find either that most workflows we encounter do not require loops or that the loops can be unrolled into an acyclic graph.

DAGMan seeks to run as many jobs as possible in parallel, given the constraints of their parent/child relationships. For example, in for the DAG in Figure 22.2, DAGMan will initially submit both N_1 and N_5 to Condor, allowing them to execute in parallel if there are sufficient computers available. After N_1 completes successfully, DAGMan will submit both N_2 and N_3 to the batch system, allowing them to execute in parallel with each other, and with N_5 if it has not completed already. When both N_2 and N_3 have finished successfully, DAGMan will submit N_4. If N_5 and N_4 both complete successfully, the DAG will have completed, and DAGMan will exit successfully.

Earlier we defined a node's parent and child relationships. In describing DAGs, it can also be useful to define a node's *sibling* as any node that shares the same set of parent nodes (including the empty set). Although sibling relationships are not represented explicitly inside DAGMan, they are important because sibling nodes become "runable" simultaneously when their parents complete successfully. In Figure 22.2, N_1 and N_5 are siblings with no parents, and N_2 and N_3 are siblings that share N_1 as a parent.
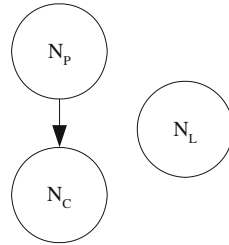
Figure 22.1: The relationship between parents and children. N_p is the parent of N_c. N_L is lonely and has no parents or children.

In practice, however, DAGMan submits individual jobs to the batch scheduler one at a time and makes no guarantees about the precise order in which it will submit the jobs of nodes that are ready to run. In other words, N_1 and N_5 may be submitted to the batch system in any order.

It is also important to remember that, once submitted, the batch system is free to run jobs in its queue in any order it chooses. N_5 may run after N_4, despite being submitted to the queue earlier. Additionally, the jobs may not be run in parallel if there are insufficient computing resources for all parallel jobs.
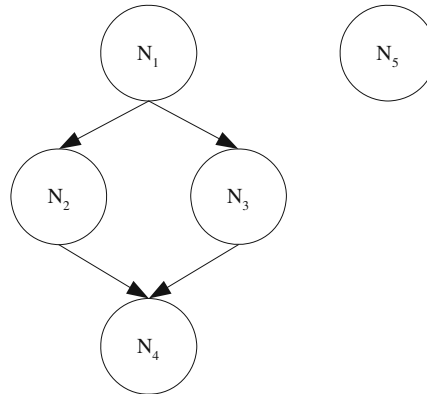
Figure 22.2: This "diamond" DAG illustrates parent and child links. N_1 must complete successfully, then both N_2 and N_3 can execute in parallel. Only when both of them have finished successfully can N_4 begin execution. N_5 is a disconnected node and can execute in parallel with all of the other nodes.

While running, DAGMan keeps a list in its memory of all jobs in the DAG, their parent/child relationships, and their current status. Given this
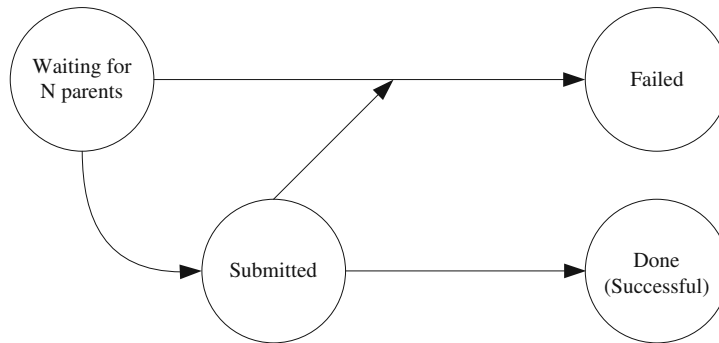
Figure 22.3: The state transition diagram for each node of a DAG. See the text for details.

information, DAGMan submits jobs to the batch system when appropriate, and continues until either the DAG is complete or no more forward progress can be made due to failed jobs. In the latter case, DAGMan creates a list of failed jobs along with the reasons for their failure and produces a *rescue DAG* file.

A rescue DAG is a special DAG that represents the state of a previously partially completed DAG such that the original DAG can be restarted where it left off without repeating any successfully completed work. The rescue DAG is an exact copy of the original input DAG, except that all previous nodes that have successfully completed are marked as done.

When DAGMan is restarted with a rescue DAG, it reconstructs the state of the previous DAG. Internally, DAGMan keeps track of the current status of each node. Figure 22.3 shows the basic state diagram of a DAG node. Note that this is simplified, and we will expand the diagram in the rest of this chapter.

When DAGMan starts, it marks each node as "waiting" and initializes a waiting count (N) for the node equal to its number of parents. In the case of a rescue DAG, DAGMan sets the waiting count equal to the number of parents that are not already marked as "done."

A node's waiting count represents the number of its parents that have yet to complete successfully and are therefore preventing it from being submitted. Only when a node's waiting count reaches zero can DAGMan submit the job associated with the node. If the job is submitted successfully, DAGMan marks the node as "submitted." If the job submission fails for any reason, the node is marked as "failed."

When DAGMan detects that a job has left the batch system queue, it marks the node as "done" if the job exited successfully, otherwise it marks the node as "failed." Success is determined by the exit code of the program:

If it is zero, then the job exited successfully, otherwise it failed. (But see the description of post scripts in Section 22.3.2 for a modification of this.)

When a job is marked "done," the waiting count of all its children is decremented by one. Any node whose waiting count reaches zero is submitted to the batch scheduler as described earlier.

22.3.2 DAGMan Complications

So far, the description of a DAGMan DAG is not very interesting: We execute jobs and maintain the order in which they must execute, while allowing parallelism when it is possible. Unfortunately, this is insufficient in real environments, which have many complications and sources of errors.

Complication: Setup, Cleanup, or Interpretation of a Node

The first complication occurs when using executables that are not easily modified to run in a distributed computing environment and therefore need a setup or cleanup step to occur before or after the job. For example, before a job is run, data may need to be staged from a tape archive or uncompressed. While this step could be placed in a separate DAG node, this may cause unnecessary overhead because the DAG node will be submitted and scheduled as a separate job by the batch system instead of running immediately on the local computer.

DAGMan provides the ability to run a program before a job is submitted (a *pre script*) or after a job completes (a *post script*). These programs should be lightweight because they are executed on the same computer from which the DAG was submitted and a large DAG may execute many of these scripts. (But see the discussion on throttling for our solution to preventing too many of these scripts from running simultaneously.)

Running these scripts adds complexity to the state diagram in Figure 22.3. The changes needed to support scripts are shown in Figure 22.4. Once a job is allowed to run, it can optionally run a pre script. After the job has run, it can optionally run a post script.

Note that if the scripts fail, the node is considered to have failed, just as if the job itself had failed. There is one interesting case to note that is not easily represented in Figure 22.4: If a node has a post script, it will never go directly into the failed state but will always run the post script. In this way, the post script can decide if a job has really failed or not. It can do some analysis beyond DAGMan's ability to decide if a node should be considered to have succeeded or failed based on whether the exit code is zero or not, perhaps by examining the output of the job or by accepting alternate exit codes. This significantly enhances the ability of DAGMan to work with existing code.

Some users have discovered an interesting way to use pre scripts. They create pre scripts that rewrite the node's job description file to change how the job runs. This can be used for at least two purposes. First, it can create
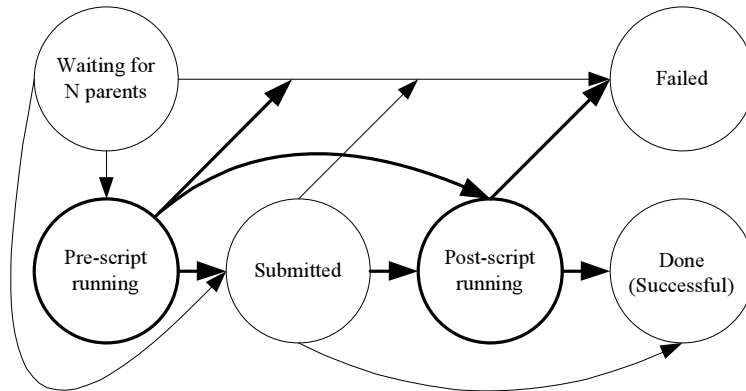
Figure 22.4: A state diagram for executing a single DAG node. Unlike Figure 22.3, this diagram adds the ability to run pre scripts and post scripts. The differences from Figure 22.3 are noted in bold.

conditional DAGs by allowing a runtime decision that changes the DAG. For example, consider Figure 22.5. If N_1 succeeds, then the prescript for N_3 will rewrite N_3 to an empty job—perhaps running the /bin/true command.[1] In this way, only N_2 will run after N_1 succeeds. Similarly, if N_1 fails, then only N_3 will run. While a more generic facility for conditional DAGs may be desirable, it would add complexity, and simple conditional DAGs can be created in this way.

A second use for pre scripts is to do last-minute planning. For example, when submitting jobs to Condor-G (which allows jobs to be submitted to remote grid sites instead of the local batch system), users can specify exactly the Grid site at which they wish their jobs to run. A pre script can decide what Grid site should be used, rewrite the job description, and the job will run there.

Complication: Throttling

All of the mechanisms described so far work very well. Unfortunately, the real world applies additional constraints. Imagine a DAG that can have one thousand jobs running simultaneously, and each of them has a pre script and a post script. When DAGMan can submit the jobs, it will start up one thousand nearly simultaneous pre scripts and then submit one thousand jobs nearly simultaneously. Running that many pre scripts may cause an unacceptable load on the submission machine, and submitting that many jobs to the underlying batch submission system may also strain its capacity.

---

[1] In recent versions of Condor, the job can be edited to contain "$noop\_job = true$" which immediately terminates the job successfully.
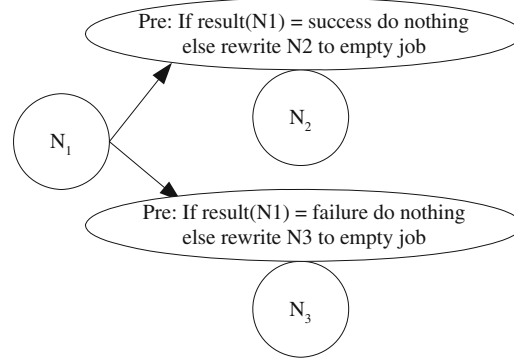
Figure 22.5: An example conditional DAG.

For this reason, DAGMan can throttle the number of pre scripts, jobs, or post scripts that may run at any time. This results in another modification to our state diagram for running a single node, as shown in Figure 22.6. For example, a node will not leave the pre script ready state unless there are fewer pre scripts running than the throttle allows.

DAGMan can also throttle the number of jobs that it submits to the batch system to avoid submitting more jobs than can be handled by the batch system. This is a good example of a surprising additional constraint: We did not realize that DAGs might be able to submit so many jobs that the number of idle jobs could overwhelm the batch system.

Complication: Unreliable Applications or Subsystems

Some applications are not robust—it is not uncommon to find a program that sometimes fails to run on the first attempt but completes successfully if given another chance. Sometimes it is due to a program error and sometimes due to interactions with the environment, such as a flaky networked file system. Ideally, problems such as this would always be fixed before trying to run the program. Unfortunately, this is not always possible, perhaps because the program is closed-source or because of time constraints.

To cope with unreliable programs or environments, DAGMan provides the ability to retry a node if it fails. Users specify how many times the node should be retried before deciding that it has actually failed. When a node is retried, the node's pre script is also run again. In some cases, a user wants to retry multiple times unless some catastrophic error occurs. DAGMan handles this with the "retry unless-exit" feature, which will retry a job unless it exits with a particular value. One place this might be useful is planning. Imagine a pre script that decides where a job should be run. Retry might be set to 10 to allow the job to be run at ten different sites, but if there is some catastrophic
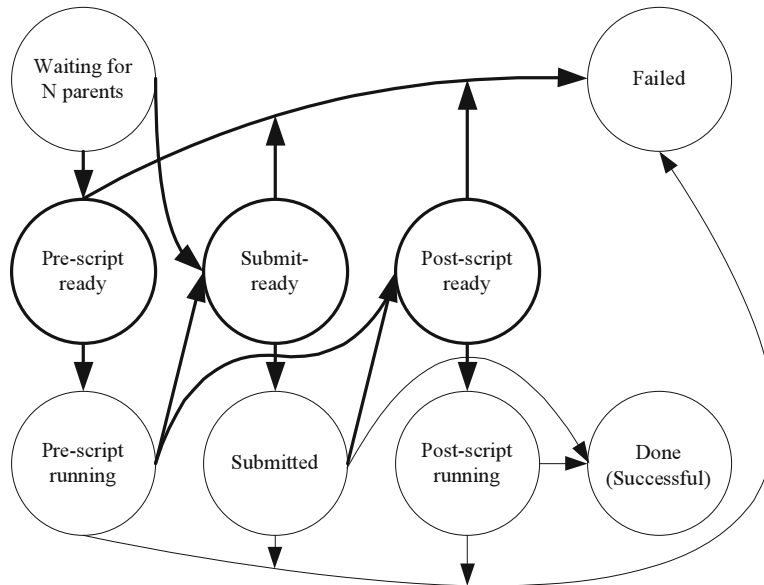
Figure 22.6: A state diagram for executing a single DAG node. In addition to the state in Figure 22.4, this diagram adds DAGMan's ability to throttle pre scripts, jobs, and post scripts. The differences from Figure 22.4 are noted in bold.

error, then the pre script can exit with a specific value that indicates "do not retry." Adding the ability to retry the job results in one final change to our state diagram, as shown in Figure 22.7.

22.3.3 Additional DAGMan Details

Submission Failures

When submitting a job to the underlying batch system, sometimes the job submission itself (not the job execution) will fail for reasons that are not the fault of the user. Usually, this is due to heavy use of the batch system, and it becomes temporarily unavailable to accept submissions. DAGMan will retry the job submission up to six times (by default, but this can be changed), increasing the amount of time between job submissions exponentially in order to increase the chance of a successful submission. If the job continues to fail to submit successfully, Condor will mark the job submission as failed.

Running DAGMan Robustly

What happens if the machine on which DAGMan is running crashes? Although DAGMan would no longer continue to submit jobs, existing jobs
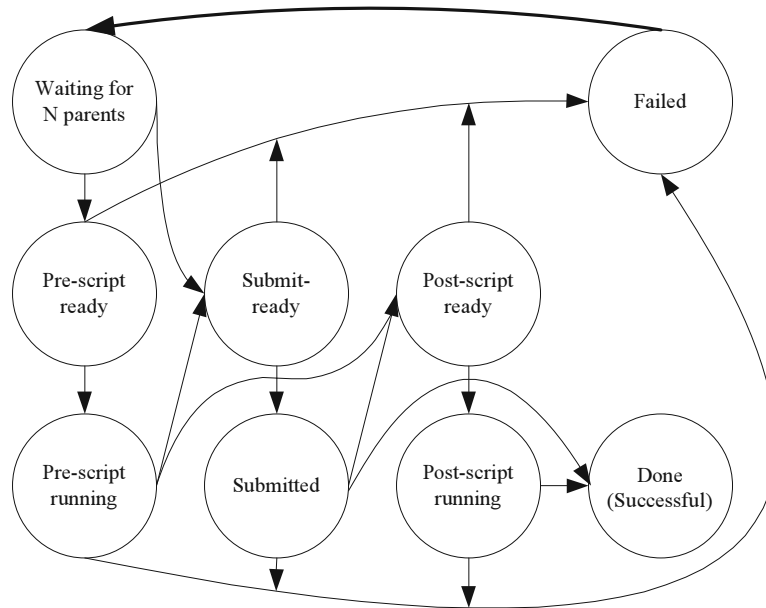
Figure 22.7: The complete state diagram for executing a single DAG node. The single difference from Figure 22.6 is noted in bold.

continue running, but it would be nice if DAGMan could be restarted so that it could continue to make forward progress. Ideally, DAGMan should handle as much as possible for the user, so we run DAGMan as a Condor job. This means that if the machine crashes, when it restarts Condor will restart the DAGMan process, which will recover the state of its execution from persistent log files, and will resume operation. This sort of robustness is essential in allowing users to run large sets of jobs in a "hands-off" fashion.

Recursive DAGs

A DAG node can submit any valid jobs, including submitting another DAG. This allows the creation of DAGs with conditional branches in them. The DAG node can make a choice, then submit an independent DAG based on the result of that choice. This can allow very complex DAGs to be executed. Unfortunately, it also makes it harder to debug a DAG. For an alternative to recursive DAGs, see Section 22.7.

22.3.4 Describing a DAG

It is the user's responsibility to provide DAGMan with a description of each job in the format of the underlying batch scheduler. For Condor, this means

associating each node with a "submit file" describing the job to be executed. DAGMan ultimately uses this file to submit the job to the batch scheduler using the standard submission interface.

Users describe a DAG by listing each node and the relationships between nodes. A sample DAG description is shown in Figure 22.8.

```
Job N1 submit-n1
Job N2 submid-n2
Job N3 submid-n3
Job N4 submid-n4
Job N5 submid-n5

Parent N1    Child N2 N3
Parent N2 N3 Child N4

Retry N1 5

Script PRE  N5 uncompress-data
Script POST N5 uncompress-data
```

Figure 22.8: How a user might describe the diamond DAG from Figure 22.2. In this description, node N1 can be retried five times, and none of the other nodes are retried if they fail. Node N5 has both a pre script and a post script

22.3.5 DAGMan Experience

DAGMan has been used extensively with the Condor batch job scheduling system. We have found that our implementation of the DAGMan easily scales to large DAGs of around 1000 nodes without throttling and DAGs of around 100,000 nodes with throttling. We believe it could scale much further than that if necessary. Because DAGMan can manage DAGs of this scale and because we find that the greatest bottleneck is in the underlying batch job submission system capabilities, we have not expended effort to optimize it to work with DAGs larger than 100,000 nodes.

DAGMan has been used in a wide variety of production environments. We will provide two examples here.

Within the Condor Project, we have created a Basic Local Alignment Search Tool (BLAST) [53] analysis service for the Biological Magnetic Resonance Data Bank at the University of Wisconsin–Madison [55]. BLAST finds regions of local similarity between nucleotide or protein sequences. Local researchers do weekly queries against databases that are updated every week. Our service takes a list of sequences to query and creates a pair of DAGs to perform the queries, as illustrated in Figure 22.9. The first DAG performs

the setup, creates a second DAG that does the queries (the number of nodes in this DAG varies, so it is dynamically created), and then assembles the results. These DAGs are used differently: The first DAG uses dependencies to order the jobs that are run, while the second DAG has completely independent nodes, and DAGMan is used for reliable execution and throttling. On average, the second DAG has approximately 1000 nodes, but we have experimented with as many as 200,000 nodes. This service has run on a weekly basis for more than two years with little human supervision.
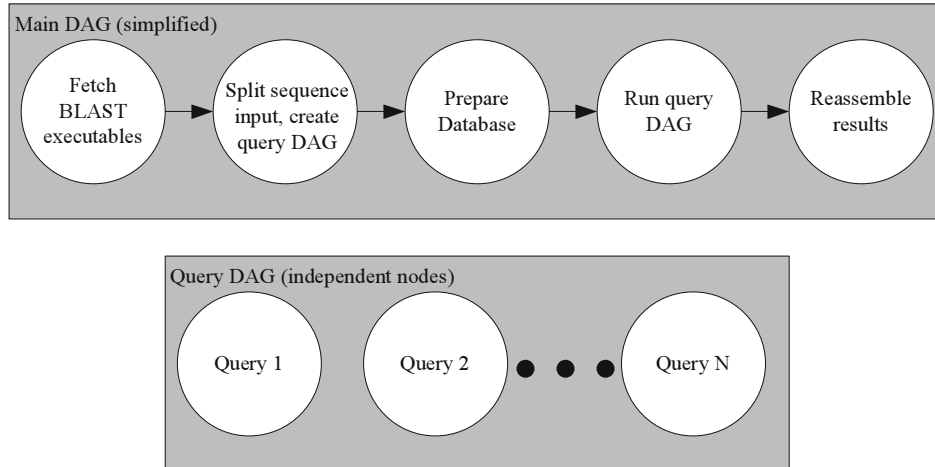
Figure 22.9: The pair of DAGs used to run BLAST jobs. The query DAG is created by the main DAG. See the text for details.

The Virtual Data System (VDS) (see Chapter 23 or [148]) builds on top of DAGMan and Condor-G. Users provide a description of what data are available and how the data can be transformed, then request the data they need. The VDS creates a DAG that fetches and transforms data as needed, while tracking the provenance of the data. As part of the DAG creation and execution, the VDS uses planning to decide which Grid sites should perform the transformations. The VDS has been used for a wide variety of applications, including high-energy physics event simulation, finding galaxy clusters, and genome analysis.

22.4 Implementation Status

DAGMan has been freely distributed as part of the Condor software since 1999. It has been used for numerous large projects and is stable. It is available for a wide variety of Unix platforms, Microsoft Windows, and Mac OS X.

22.5 Interaction with Condor

Condor is a high-throughput batch job scheduler. Because it has been covered in detail elsewhere ([262], [414]), we only briefly review it here.

Condor was originally designed to utilize CPU cycles on computers that would otherwise be idle, such as desktop computers that are unused but turned on overnight. However, Condor has expanded its scope and now works well with dedicated computers and Grid systems. Condor's ability to interact with a Grid system called Condor-G [151] allows Condor to submit jobs to Globus [166] (versions 2, 3, and 4), NorduGrid, Oracle, LSF, PBS, and even remote Condor installations (referred to as Condor-C).

Condor and Condor-G emphasize reliability. If Condor crashes, it will continue running the jobs when it restarts. Condor can provide job checkpointing and migration to facilitate recovery when execution computers fail. Condor-G provides elaborate recovery schemes to deal with network outages and remote Grid site failures.

DAGMan is built to use Condor for job execution, and it can submit jobs to both the local batch system and remote Grid systems with equal ease. We have created many workflows using DAGMan that execute in a Grid environment.

22.6 Integration with Stork

22.6.1 An Introduction to Stork

Just as computation and network resources need to be carefully scheduled and managed, the scheduling of data placement activities across distributed computing systems is crucial, as the access to data is generally the main bottleneck for data-intensive applications. This is especially the case when accessing very large data stores using mechanical tape storage systems.

The most common approaches to solving the problems of data placement are either interactive data movement or the creation of simple scripts. Generally, scripts cannot adapt to a dynamically changing distributed computing environment: They do not have the privileges of a job, they do not get scheduled, and they do not have any automation or fault-tolerance capabilities. Furthermore, most scripts typically require close monitoring throughout the life of the process and frequent manual intervention.

Data placement activities must be first-class citizens in the distributed computing environments, just like computational jobs. Data placement jobs need to be queued, scheduled, monitored, and even checkpointed. Most importantly, users require that their data placement jobs omplete successfully without human monitoring and intervention.

Furthermore, data placement jobs should be treated differently from computational jobs, as they have different semantics and different characteristics. For example, if the transfer of a large file fails, we may not

simply want to restart the job and retransfer the whole file. Rather, we may prefer to transfer only the remaining part of the file. Similarly, if a transfer using one protocol fails, we may want to try alternate protocols supported by the source and destination hosts to perform the transfer. We may want to dynamically tune network parameters or decide concurrency levels for unique combinations of the data source, destination, and protocol. A traditional computational job scheduler does not handle these cases. For this reason, data placement jobs and computational jobs should be differentiated, and each should be submitted to specialized schedulers that understand their semantics.

We have designed and implemented the first batch scheduler specialized for data placement: Stork [244]. This scheduler implements techniques specific to queuing, scheduling, and optimization of data placement jobs and provides a level of abstraction between the user applications and the underlying data transfer and storage resources.

A production-quality Stork is bundled with Condor releases. Additionally, research into new features is continuing in parallel.

22.6.2 Data Placement Job Types

Under Stork, data placement jobs are categorized into the following three types:

- *Transfer.* This job type is for transferring a complete or partial file from one physical location to another. This can include a get or put operation or a third-party transfer. Stork supports a variety of data transfer protocols and storage systems, including local file system, GridFTP, FTP, HTTP, NeST, SRB, dCache, CASTOR, and UniTree. Furthermore, sites can create new transfer modules using the Stork modular API.
- *Allocate.* This job type is used for allocating storage space at the destination site, allocating network bandwidth, or establishing a light path on the route from source to destination. It deals with all resource allocations required for the placement of the data.
- *Release.* This job type is used for releasing the corresponding allocated resource.

22.6.3 Flexible Job Representation

Stork uses the ClassAd [367] job description language to represent the data placement jobs. The ClassAd language provides a very flexible and extensible data model that can be used to represent arbitrary services and constraints.

Below are three sample data placement (DaP) requests:

```
[
  dap_type      = "allocate";
  dest_host     = "houdini.example.com";
  size          = "200MB";
```

```
  duration     = "60 minutes";
  allocation_id = 1;
]


[
  dap_type = "transfer";
  src_url  = "file:///data/example.dat";
  dest_url = "nest://houdini.example.com/data/example.dat";
]


[
  dap_type      = "release";
  dest_host     = "houdini.example.com";
  allocation_id = 1;
]
```

The first request is to allocate 200 MB of disk space for 1 hour on a NeST server. The second request is to transfer a file from the local file system to the allocated space on the NeST server. The third request is to deallocate the previously allocated space.

22.6.4 Fault Tolerance

Data placement applications must operate in an imperfect environment. Data servers may be unavailable for many reasons. Remote and local networks may encounter outages or congestion. Computers may crash, including the Stork server host itself. Stork is equipped to deal with a variety of data placement faults, which can be configured at both the system and job levels.

For transient environment faults, data placement jobs that fail can be retried after a small delay. The number of retries allowed is configurable.

For longer-term faults associated with a particular data server, Stork can also retry a failed transfer using a list of alternate data protocols. If in the previous example the host houdini.example.com is also running a plain FTP server, the corresponding transfer job could be augmented to retry with plain FTP.

```
[
  dap_type      = "transfer";
  src_url       = "file:///data/example.dat";
  dest_url      = "nest://houdini.example.com/data/example.dat";
  alt_protocols = "file_ftp";
]
```

22.6.5 Interaction with DAGMan

Condor's DAGMan workflow manager has been extended to work with Stork. In addition to specifying computational jobs, data placement jobs can be

specified, and DAGMan will submit them to Stork for execution. This allows straightforward execution of workflows that include data transfer.

A simple example of how Stork can be used with DAGMan appears in Figure 22.10. This DAG transfers data to a Grid site using Stork, executes the job at the Grid site using Condor-G, and then transfers the output data back to the submission site using Stork. This DAG could easily be enhanced to allow space allocation before the data transfers, or it could have multiple data transfers.
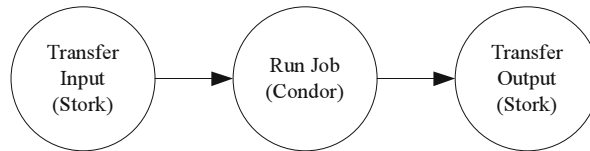
Figure 22.10: A simple DAG that includes Stork.

22.6.6 Interaction with Heterogeneous Resources

Stork acts like an I/O control system between the user applications and the underlying protocols and data storage servers. It provides complete modularity and extendibility. Users can add support for their favorite storage system, data transport protocol, or middleware very easily. This is a crucial feature in a system designed to work in a heterogeneous distributed environment. The users or applications cannot expect all storage systems to support the same interfaces to talk to each other. Furthermore, it is becoming increasingly difficult for applications to talk to all the different storage systems, protocols, and middleware. There needs to be a negotiating system between them that can interact with those systems easily and even perform protocol translation. Stork has been developed with these capabilities as requirements. The modularity of Stork allows users to insert a plug-in to support any storage system, protocol, or middleware easily.

Stork supports several data transfer protocols, including:

- FTP [362]
- GridFTP [9]
- HTTP [141]
- DiskRouter [242]

Stork supports several data storage systems, including:

- SRB [41]
- UniTree [71]

- NeST [45]
- dCache [109]
- CASTOR [82]

Stork maintains a library of plugable "data placement" modules. These modules are executed by data placement job requests. Modules can perform interprotocol translations using either a memory buffer or third-party transfers whenever available (such as GridFTP). For example, if a user requests a data transfer between two remote systems, Stork can often perform the transfer without saving the file to disk. Interprotocol translations are not yet supported between all systems or protocols, but they are available for the major use cases we have encountered so far.

In order to transfer data between systems for which direct interprotocol translation is not supported, two consecutive Stork jobs can be used instead. The first Stork job performs the transfer from the source storage system to the local disk cache of Stork, and the second Stork job performs the transfer from the local disk cache of Stork to the destination storage system.

22.6.7 Modular API

While the Stork server is a single process, the data transfers, allocations, etc., are performed by the separate *modules*. The module application program interface is simple enough for sites to write their own modules as needed. For example, each data transfer module is executed with the following argument list:

src_url dest_url arguments ...

Thus, to write a new module that transfers data from the *foo* protocol to the *bar* protocol, a new module is created with the name `stork.transfer.foo-bar`. Modules need only be executable programs, and may even be written as shell scripts. Furthermore, module binding is performed at runtime, enabling sites to create new modules without restarting the Stork server.

22.6.8 Performance Enhancements

Stork has seen several recent performance enhancements. The first is that multiple data placements may now be specified in a single submission file instead of multiple files. This optimization is significant when transferring many files to Stork because it eliminates extra invocations of the *stork_submit* command, which can be surprisingly time consuming when transferring tens of thousands of files.

The second enhancement was integration of the GridFTP client *globus-url-copy* into Stork server. When doing many simultaneous GridFTP file transfers,

this saves considerable time and reduces the total number of processes in the system.

Finally, Stork is now able to execute an arbitrary program when the active job queue size falls below a configurable level. This is envisioned as a simple but high-performance alternative to managing very large data placement workflows with DAGMan: It will allow Stork users to limit the rate at which they submit jobs so that Stork is not overwhelmed, while ensuring that Stork has sufficient work to do at any given time.

22.6.9 Implementation Status

Stork is available, with all features described so far, as part of the Condor distribution. It is available on Linux and will be available for other platforms in the future. Users outside of the Condor Project are just beginning to use Stork in production, and we hope to have more in the near future.

22.6.10 Active Research

Research on Stork is active, and much of it can be found in [243]. Research includes:

- Extending data placement types to include interaction with a metadata catalog.
- Experimentation with scheduling techniques other than first-in, first-out. This includes not only traditional scheduling techniques such as shortest job first or multilevel queue priority scheduling but also scheduling based on management of storage space to ensure that storage space is not exceeded by the data transfers. In addition, scheduling of connection management is important when there are many simultaneous connections to a server.
- Runtime adaptation can be performed to tune the network parameters for a transfer to minimize transfer time. This is discussed in further detail in [241].
- Research has been done to enable Stork to detect problems such as servers that are unreliable in a variety of ways and then base scheduling decisions on this knowledge. More details are in [240].

22.7 Future Directions

There are several promising areas for future work with DAGMan and Stork. For DAGMan, we would like to explore methods (probably utilizing ClassAds [367]) to allow different conditions for deciding when a node should execute. Today, a node executes when all of its parents have finished with an exit code of 0, but allowing more complex conditions would allow conditional

execution (equivalent to if-then-else) and partial execution (a DAG that finishes when a certain percentage of nodes have completed).

We also would like to support dymamic DAGs, which are DAGs that can change on-the-fly, based on user input. This has been frequently requested by users and is particularly useful when DAGMan is used by a higher-level scheduling system that may change plans in reaction to current conditions.

For Stork, we are exploring ways to make it more scalable and more reliable. We are also investigating methods to use matchmaking, similar to that in Condor, to select which data transfers should be run and which sites they should transfer data to.

22.8 Conclusions

DAGMan is a reliable workflow management system. Although the workflows it supports are relatively simple, there are many complexities that were discovered as we used DAGMan through the years, such as the need for flexible methods for retrying and throttling. As a result of our experience, DAGMan has found favor with many users in production environments, and software has been created that relies on DAGMan for execution. Used with Condor, Condor-G, and Stork, DAGMan is a powerful tool for workflow execution in Grid environments.

23

Pegasus: Mapping Large-Scale Workflows to Distributed Resources

Ewa Deelman, Gaurang Mehta, Gurmeet Singh, Mei-Hui Su, and
Karan Vahi

23.1 Introduction

Many scientific advances today are derived from analyzing large amounts
of data. The computations themselves can be very complex and consume
significant resources. Scientific efforts are also not conducted by individual
scientists; rather, they rely on collaborations that encompass many researchers
from various organizations. The analysis is often composed of several
individual application components designed by different scientists. To describe
the desired analysis, the components are assembled in a workflow where
the dependencies between them are defined and the data needed for the
analysis are identified. To support the scale of the applications, many resources
are needed in order to provide adequate performance. These resources are
often drawn from a heterogeneous pool of geographically distributed compute
and data resources. Running large-scale, collaborative applications in such
environments has many challenges. Among them are systematic management
of the applications, their components, and the data, as well as successful and
efficient execution on the distributed resources.

In order to manage the process of application development and execution,
it is often convenient to separate the two concerns. For example, the
application can be developed independently of the target execution system
using a high-level representation. Then, once the target execution environment
is identified, the application can be mapped onto it.

In this chapter, we describe a system, Pegasus (Planning for Execution
in Grids) [111, 116], which given a workflow instance and information about
the available resources generates an appropriate executable workflow. Pegasus
enables scientists to design workflows at the application level without the need
to worry about the actual execution environment, be it a Grid [147], a set of
Condor pools [262], or a local machine. Pegasus is a flexible framework that
can be tailored to the performance needs of a variety of applications and
execution environments. In this chapter, we describe Pegasus's functionality
and summarize results of using Pegasus in dedicated and shared execution

environments. We present the optimizations that are made to improve the overall workflow performance. We also describe some of the applications that use Pegasus today.

23.2 Workflow Generation for Pegasus

In our work, we distinguish between three different forms of a workflow: a template, an instance, and an executable workflow. A *workflow template* is a skeleton of a computation; it describes the various application components that are part of the scientific analysis, as well as the dependencies between the components. The workflow template provides the general structure of the analysis but does not identify the data or the resources that are necessary to obtain the desired results. Templates can be designed collaboratively by a group of scientists. Once the template is agreed upon, it can be stored in a library for future use.

A scientist needs to provide data to the workflow template in order to fully specify the analysis. The template and the input data together form a *workflow instance* (also known as an *abstract workflow*). In this form, the workflow uniquely identifies the analysis but does not contain information about the resources that will be used to execute the computations. The workflow instance is portable; it can be mapped to a variety of execution environments.

The workflow that includes all the necessary resource information is an *executable workflow* (also known as a *concrete workflow*). This workflow identifies the resources where each workflow task will be executed, provides tasks for staging data in and out of the computations (including the identification of specific data replicas), and any other tasks needed for data and computation management (such as data registration or creating an execution directory at the execution site).

In general, a given workflow can be a mixture of a template, instance, and executable. For example, portions of the workflow can be refined while others are being executed. However, in practice, the workflow is often just in one particular form. It is possible that users may develop workflow templates and instances ahead of a particular experiment and then only during the experiment's runtime are the workflows executed.

In our work, we have used a variety of different methods to create a workflow instance. The first technique is appropriate for application developers who are comfortable with the notions of workflows and have experience in designing executable workflows (workflows already tied to a particular set of resources). They may choose to design the workflow instances directly according to a predefined schema. Another method uses Chimera [148] to build the workflow instances, based on the user-provided partial logical workflow descriptions specified in the Virtual Data Language (VDL) (Chapter 17). We also had initial experiences in using Triana (Chapter 20) where

the system created workflow instances using the graphical user interface. Workflow instances may also be constructed using assistance from intelligent workflow editors such as the Composition Analysis Tool (CAT) [237]. CAT uses formal planning techniques and ontologies to support flexible mixed-initiative workflow composition that can critique partial workflows composed by users and offer suggestions to fix composition errors and to complete the workflow templates. When using the CAT software, an input data selector component uses the Metadata Catalog Service (MCS) [386] to populate the workflow template with the necessary data. MCS performs a mapping from specific metadata attributes to particular data instances. The three methods of constructing the workflow instance can be viewed as appropriate for different circumstances and scientists' backgrounds, from those very familiar with the details of the execution environment to those that wish to reason solely at the application level.

In any case, all three workflow creation methods result in a workflow instance that needs to be mapped onto the available resources to facilitate execution. The workflow mapping problem can be defined as finding a mapping of tasks to resources that minimizes the overall workflow execution time. The workflow execution consists of both the running time of the tasks and the data transfer tasks that stage data in and out of the computation.

Since the execution environment can be very dynamic, and the resources are shared among many users, it is impossible to optimize the workflow from the point of view of execution ahead of time. In fact, one may want to make decisions about the execution locations and the access to a particular (possibly replicated) data set as late as possible.

23.3 Pegasus and the Target Workflow Execution Environment

Pegasus is a framework that allows the mapping of workflow instances onto a set of distributed resources such as Grid [147] or a Condor pool [262]. The mapping process not only involves finding the appropriate resources for the tasks but also may include some workflow restructuring geared toward improving the performance of the overall workflow. In order to adapt to a dynamic environment, Pegasus may also map only portions of a workflow at a time.

23.3.1 Target Execution System Overview

In order to understand the functionality of Pegasus, it is important to describe the environment in which the workflows are to be executed. We assume that the environment is a set of heterogeneous hosts connected via a network, often a wide-area network. The hosts can be single-processor machines, multiprocessor clusters, or high-performance parallel systems.

In order to be able to schedule jobs remotely, Pegasus needs a job submission interface and the ability to automatically stage data to locations accessible from the computational resources.
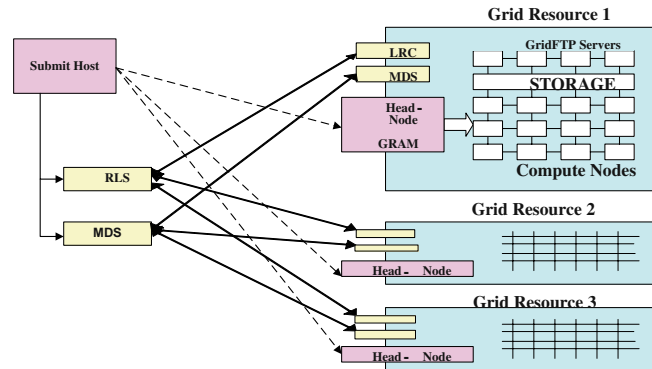
Figure 23.1: An example execution host configuration.

Figure 23.1 shows a typical execution system, with a head node visible to the network, additional hosts that form a pool of resources, and a storage system. In order to be able to schedule jobs remotely, the resource needs to have appropriate software deployed that can provide information about the resources, stage data, and accept job submissions. In our work, we use the Globus Toolkit [144] to provide these functionalities.

In order to use Pegasus in such an environment, a resource (named the submit host), which could be a user's desktop, needs to be set up to provide Pegasus, DAGMan, and Condor-G [152]. The latter two systems provide the workflow execution engine (Chapter 22) and the capability to remotely submit jobs to a variety of Globus-based resources. The submit host also maintains information about the user's application software installed at the remote sites (in the Transformation Catalog (TC) [114], described below) and about the execution hosts of interest to the user (in the Site Catalog, described below). The submit host can also serve as a local execution platform for small workflows or for debugging purposes.

23.3.2 Information Services and Catalogs

Pegasus interfaces with various catalogs to discover the data locations, executable locations and the available resources and their characteristics. Pegasus reduces the workflow based on the available intermediate data products since it is possible that some of the analysis was already conducted and does not need to be repeated. Once the workflow is reduced, Pegasus locates the available resources and input data and maps the workflow

components onto the distributed resources. Pegasus can use various mapping algorithms and can also map only portions of the workflow at a time to adjust the mapping to the dynamic environment. We describe Pegasus' s functionality in more detail in the following subsections.

Replica Catalog

Pegasus uses a replica catalog to dynamically gather information about the locations of the data products that are required by the workflows. Input data products are usually raw data files that cannot be produced as part of a workflow execution and are needed to be present before the workflow execution starts. Cataloging of the intermediate data products and final data products helps in data reuse and allows reduction of parts of the workflow if data already exist. This is described in detail in Section 23.4. A replica catalog stores the logical to physical filename mappings. In addition, it stores attributes of the physical filenames, such as the handle of the site where the physical file resides. We use the Globus Replica Location Service (RLS) [88] as our default replica catalog implementation. RLS is a distributed replica management system consisting of local catalogs that contain information about logical to physical filename mappings and distributed indexes that summarize the local catalog content.

Transformation Catalog

Pegasus interfaces with a transformation catalog to gather information about where the transformations are installed on the Grid sites. The transformations are the executables corresponding to the jobs in the workflow instance. Similarly to the replica catalog, the transformation catalog stores the logical to physical filename mappings.

   In addition to the installed executables, the Transformation Catalog can also store the location of statically linked executables (as part of the physical mapping) that can be staged to remote sites as part of workflow execution. The staging of executables is described in detail in Section 23.4. The catalog also stores a variety of attributes associated with the executables, such as the target operating system, compiler used, memory needed, etc. The default implementation of the transformation catalog is a database with a predefined schema.

Site Catalog

Pegasus interfaces with a *Site Catalog* to gather information about the layout of remote sites. It stores both static and dynamic information. The static information includes information such as:

- The GridFTP [9] servers that stage data in and out of the site

- The GRAM [102] jobmanagers that submit jobs to the local scheduler running on the Grid site
- The scratch directories to which data can be staged to as part of workflow execution
- The storage directories to which the final data products can be staged

  The dynamic information includes information such as:

- The number of available processors
- The amount of available memory
- The amount of available disk space

The site catalog can be populated using data provided by the Globus Monitoring and Discovery Service (MDS) [101] and additional information provided by the user or site administrator. The site catalog can also be populated using a Grid-specific catalog such as GridCAT [174] for OSG [328].

23.4 Pegasus and Workflow Refinement

Pegasus transforms a workflow instance to an executable (concrete) workflow through a series of refinements. The workflow instance (Figure 23.2 shows an example) is composed of tasks described in terms of logical transformations and logical input and output filenames. The workflow instance is independent of the resources. The goal of Pegasus is to find a good mapping of the tasks to the available resources necessary for execution. Figure 23.3 depicts the steps taken by Pegasus during the workflow refinement process.
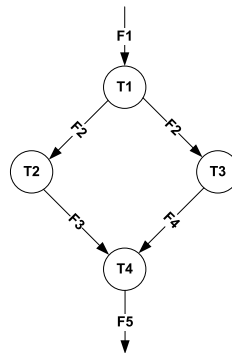
Figure 23.2: An example workflow instance composed of four tasks. Ti stands for a logical transformation (task). Fi is a logical filename.

Defining the set of available and accessible resources. First, Pegasus consults the Site Catalog to check which resources are available. Additionally,

Pegasus may try to authenticate to these resources using the user's credentials. Thus, the possible set of resources may be reduced to a minimum.

Workflow reduction. The next step may modify the structure of the workflow instance based on the available data products. Pegasus consults the replica catalog to determine which intermediate data products are already available. Based on this information, Pegasus may reduce the workflow to contain only the tasks necessary to generate the final data products. In the extreme case, if the final data are already available, no tasks will be scheduled except perhaps a data transfer (to the user-specified location) and a registration of the desired data products. If we consider the workflow instance in Figure 23.2 and suppose that the replica catalog indicates that files $F3$ and $F4$ are available, then the reduced workflow instance would consist only of one task, $T4$.

Resource selection. At this point, we have the minimal workflow instance in terms of the number of tasks. The workflow reduction was made based on the assumption that it is more efficient to access the data than to recompute them. Given the minimal workflow, a site (resource) selection is performed. This selection can be done based on the available resources and their characteristics, as well as the location of the required input data. The type of site selection performed is customizable as a plugable component within Pegasus. The system incorporates a choice of a few standard selection algorithms: random, round-robin, group-based,[1] and min-min [54]. These algorithms can be applied to the selection of the execution site as well as the selection of the data replicas. The selection algorithms make use of information available in the Site Catalog (resource characteristics), Transformation Catalog (the location of the application software components), and replica catalog (the location of the replicated data). It is also possible to delay data replica selection until a later point, in which case the replica catalog is not consulted at this time. Additionally, users may wish to add their own algorithms geared toward their application domain and execution environment. These algorithms may also rely on additional or different information services that can be plugged into Pegasus as well.

Task clustering. Pegasus provides an option to cluster jobs together in cases where a number of small-granularity jobs are destined for the same computational resource. During clustering, we consider only independent tasks, so that they can be viewed by the remote execution system as a single entity. The task clusters can be executed on a remote system either in a sequence or if feasible and appropriate they can be executed using MPI [389] in a master/slave mode. In the latter case, an initial number of processors is

---

[1] In group-based site selection, the jobs in the workflow instance are tagged prior to the mapping by Pegasus. The nodes with the same tags belong to a single group. All the jobs in the same group are scheduled by Pegasus to the same site. The tagging can be user-defined or performed automatically based on the performance characteristics of the workflow components.

requested and the clustered tasks are sent to the remote site as the constituent task execution is completed.
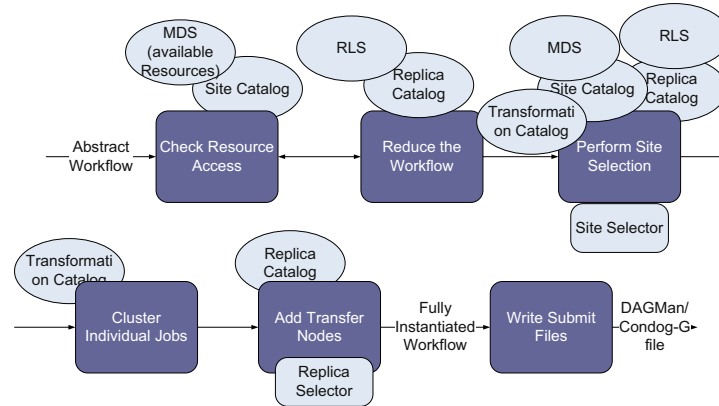
Figure 23.3: Pegasus's logic.

Executable staging. Pegasus provides the ability to stage the executables to a remote Grid site. The executable may need to be staged either if it is not installed there or the version of the executable installed is an out-of-date version. Currently, Pegasus enables only the transfer of statically linked executables. Pegasus supports the following staging of executable scenarios:

- *Installed* executables are always preferred—only sites that have the executables installed are selected for running the workflow.
- *Staged* executables are always staged to the remote Grid sites, irrespective of whether they are installed there or not.

Once Pegasus determines that an executable is to be staged for a particular computation job, it adds the executable file as an input to that particular job. The transfer of the executable is then handled in the same manner as the transfer of other data files required by the computation job.

Adding data stage-in, stage-out and registration tasks. The workflow instance contains only nodes representing computations. Since the workflow can be executed across multiple platforms and since data need to be staged in and out of the computations, Pegasus augments the workflow with tasks that explicitly perform the necessary data transfers. If, during site selection, data replica selection was not performed, it can be done at this point. Again, users have the option of using Pegasus-provided algorithms or suppling their own. These algorithms are used to determine which of possibly many data replicas will be used as a data access location. Once the location is determined, a data stage-in node is placed in the workflow and a dependency to the corresponding computation is added. Additionally, where appropriate, intermediate and final

data may be registered in the data catalogs, such as the replica catalog or a metadata catalog, to enable subsequent data discovery and reuse. The data registration is also represented explicitly by the addition of registration tasks and dependencies to the workflow.

Following the example of Figure 23.2, where we have a reduced workflow containing only task $T4$, the executable workflow would look as depicted in Figure 23.4. The workflow consists of three stage-in tasks, which stage the two input files to the resource R (selected by Pegasus) and stage the executable $T4$. There is an additional job following the executable staging, which sets the x bit on the staged executable (the x bit on a file is usually not preserved while transferring a file using the Grid transfer tools). Then, the task $T4$ is to be executed followed by a transfer of the data file $F5$ to the user-specified location U. Finally, the output file is registered in the RLS. It is important to note that the executable workflow in the figure is a "plan" that needs to be given to a workflow execution engine to execute.
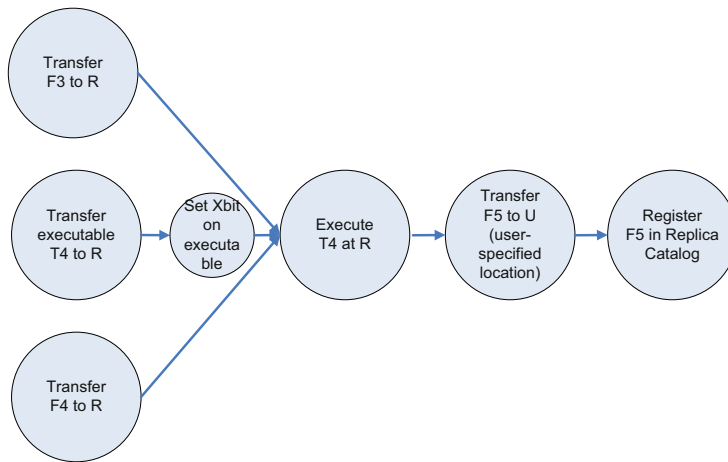
Figure 23.4: Executable workflow corresponding to the reduced workflow instance in Figure 23.2.

Submit file generation. At this point, all the compute resource and data selection has been performed and the workflow has the structure corresponding to the ultimate execution structure, which includes computation, data transfer, and registration. The final step is to write it out in a form that can be interpreted by a workflow execution engine such as, for example, DAGMan (Chapter 22). Once this has been accomplished, the resulting submit files can be given to DAGMan and Condor-G for execution. DAGMan will follow the dependencies in the workflow and submit available tasks to Condor-G, which in turn will dispatch the tasks to the target resources.

The sequence of refinements depicted in Figure 23.3 is currently static, but one can imagine constructing the sequence dynamically based on user and/or application requirements.

23.5 Workflow Execution

Executing a workflow involves submitting the ready tasks to the remote resources, monitoring their execution, and analyzing the dependencies in the workflow. A workflow execution engine needs to be able to parse the workflow description, interface with the remote resources, monitor the status of currently executing tasks, and act on task completion events by identifying new tasks that have become ready for execution as a result of past completions. Condor DAGMan [97] is such a workflow execution engine. It can execute a task graph on the machines in a Condor pool or using Condor-G on a set of resources that are accessible via the Globus Resource Allocation Manager (GRAM) [102] (for example, Grid systems that use PBS [189] or LSF [271] as a front-end scheduler). The combination of DAGMan, Condor-G, and GRAM allows the execution of an executable workflow generated by Pegasus in a Grid environment composed of heterogeneous resources.

23.6 Adapting the Workflow Mapping to a Dynamic Execution Environment

In dynamic execution environments, resources can come and go suddenly. This poses a problem as well as an opportunity for workflow management systems. It is a problem in that one cannot plan too far into the future because the resources the planner assumed would be available may become inaccessible or overloaded at the time when tasks are sent to them. A dynamic environment can potentially also be an opportunity in that new resources may come online or become lightly loaded. This enables the workflow management system to take advantage of the newly available resources, provided the workflow management system can adapt to the changes.

In the case of Pegasus, we can adapt to the execution environment by mapping only portions of the workflow at a time (also known as *deferred mapping*). Currently, we support only a static partitioning scheme, where the workflow is partitioned ahead of time [116] and then Pegasus is called on each partition in turn to map the portions of the workflow defined within the partition. The dependencies between the partial workflows (or subworkflows) reflect the original dependencies between the tasks of the workflow instance. Pegasus then maps the partial workflows following these dependencies. The original workflow is partitioned according to a specified partitioning algorithm. The result is a workflow, where the elements are partial workflows.
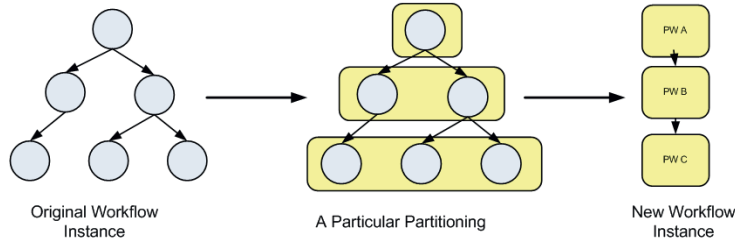
Figure 23.5: Partitioning of the workflow for deferred mapping.

The particular partitioning algorithms shown in Figure 23.5 simply partition the workflow based on the level of the node in the workflow instance. The partitioner is a plugable component, where various partitioning algorithms can be used depending on the type of workflow and the dynamic nature of the resources. Once the partitioning is performed, Pegasus maps and submits the partial workflows to DAGMan.
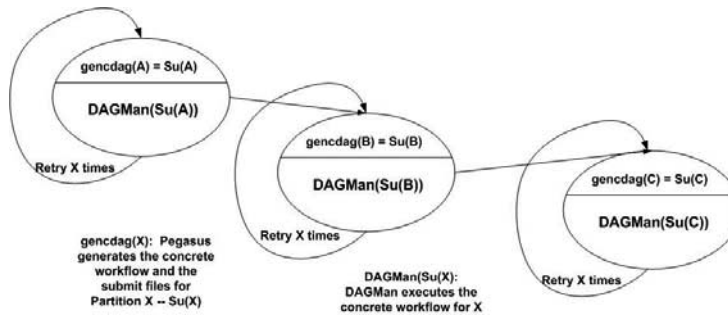
Figure 23.6: Deferred workflow mapping. A meta workflow given to DAGMan (instance # 1)for execution.

Figure 23.6 illustrates the deferred mapping process. It shows a *Meta Workflow* that is generated based on the partitioning in Figure 23.5. This Meta Workflow is given to DAGMan and starts the refinement and execution of the application workflow instance. Given this Meta Workflow, DAGMan (instance #1) first calls Pegasus (shown as a call gencdag) on one partition of the workflow instance, partition A. Pegasus then generates the executable workflow and produces the submit files necessary for the execution of that workflow through DAGMan; these files are named $Su(A)$ in Figure23.6. Now the first instance of DAGMan calls a new instance of DAGMan (instance #2) with the submit files $Su(A)$. This is reflected in the $DAGMan(Su(A))$ node in Figure23.6; it is a nested call to DAGMan within DAGMan. Once the second instance of DAGMan concludes successfully, implying that the

executable workflow corresponding to the partial workflow instance A has successfully executed, the first instance of DAGMan calls Pegasus with the workflow instance B, and the process repeats until all the partitions of the workflow are refined to their executable form and executed.

23.6.1 Partition-Level Failure Recovery

Pegasus and DAGMan can be a powerful combination that enables a certain degree of remapping in case of failure. As explained above, in the Meta DAG, each task consists of a workflow partition mapping step followed by a DAGMan execution of the mapped workflow. If either of these steps fails due to a mapping failure or due to the execution, the entire task can be retried by DAGMan. An example of a situation where this is particularly useful is shown in Figure 23.7. We start with a partition containing a subworkflow in the shape of a diamond, consisting of four tasks. As mentioned before, Pegasus reduces the workflow based on the available data products. In this case, Pegasus found that files $f2$ and $f3$ are already available. Because the two files are available, tasks B and C do not need to be executed and consequently neither does task A. The resulting executable workflow is shown next. It consists of four nodes, the first two stage in files $f2$ and $f3$ to the execution location $R1$. Then task D is to be executed at location $R1$, and finally the data are to be staged out to the user-specified location. Given this mapping, DAGMan proceeds with the execution of the workflow. Let's assume that file $f2$ is successfully staged in, but for some reason there is a failure when accessing or transferring $f3$ and the data transfer software returns an error. Given this failure, the DAGMan execution of the partition fails, as does the entire original Meta Workflow node representing the refinement and execution of the partition. Upon this failure, the Meta DAG node is resubmitted for execution (by the Condor retry mechanism) and the refinement ($gencdag(A)$ and $execution(Su(A))$ is redone. In the final step we see the executable workflow that resulted from the Pegasus/gencdag mapping. We notice that Pegasus took into account that $f2$ was already successfully staged in and at the same time the reduction step did not reduce task C because $f3$ needs to be regenerated (assuming there was only one copy of $f3$ available). In this case, we also assume that $f1$ is available; thus task A still does not need to be executed. Given this new mapping, DAGMan is invoked again to perform the execution.

23.7 Optimizing Workflow Performance with Pegasus

The resources used in workflow execution are often autonomous, heterogeneous, shared in nature, and generally use batch queuing systems such as PBS, LSF, Condor, etc., for resource management. Each task in the workflow is submitted to a job queue maintained by the resource management system. All the tasks in the job queue are considered independent of each other, and the
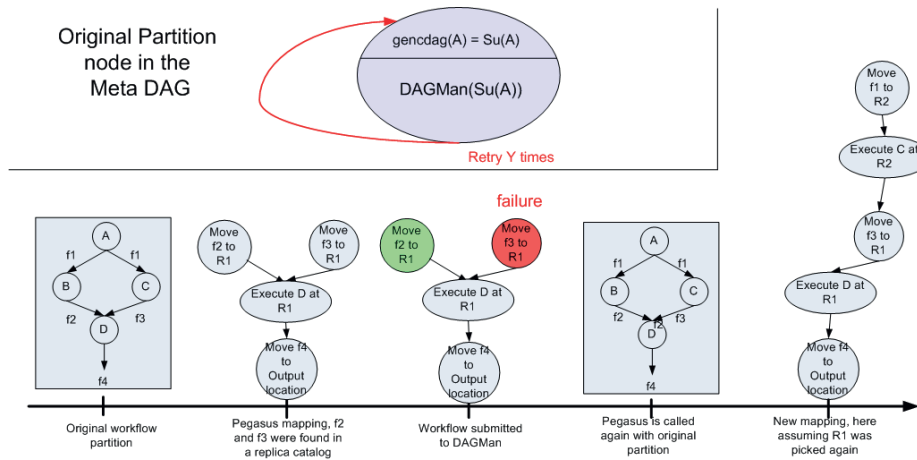
Figure 23.7: Recovery from failure. The top left shows the Meta DAG node that is being refined and executed. The bottom of the figure shows (left to right) the progression of the refinement and execution process.

remote scheduler generally does not support precedence constraints between the tasks in the queue. The performance of a resource is optimized with respect to metrics of interest to the resource provider, and tasks from the job queue are scheduled based on local policies that are not always made public. In some cases, the start time of a job depends on the time already spent by the job in the queue. The parallelism of the executable workflow is often further inhibited by policies of Grid resources that effectively limit the number of tasks that can be submitted or executed concurrently by a user.

The approach used in Pegasus for improving the workflow performance in this type of execution environment is to use *placeholders*. A placeholder is a unit of work that is submitted to the queue of a Grid resource and can be used for executing multiple tasks when the placeholder starts executing. This placeholder can be a simple shell script that is given a list of tasks to execute sequentially. It can also be implemented as an MPI wrapper program that executes all its constituent tasks in a master/slave mode on multiple worker nodes. In this particular mode of use, there is a tight binding between the tasks and the placeholders. The set of tasks that would be executed on a placeholder is known at the time the placeholder is submitted to the job queue of the target resource.

A placeholder improves the workflow performance by decoupling the definition of a task in the workflow from a task as seen by the scheduler of a remote resource. Each placeholder appears as a task to the remote scheduler. By being able to execute multiple workflow tasks using a single placeholder, we can reduce the uncertainty associated with the execution of the workflow. The workflow performance is improved because the queue wait time of the

placeholder can be amortized over the various tasks that can be executed using this placeholder. Furthermore, it can be used to overcome the constraints on concurrency since multiple tasks can be executed in parallel using a smaller number of placeholders. This approach of using placeholders works well for fine-granularity workflows where the runtime of the tasks in the workflow is on the order of seconds or minutes. For this class of workflows, the execution overhead of queue wait time can be much more than the runtime of the tasks in the workflow. Thus, placeholders can be used to reduce this execution overhead and improve the workflow performance [116]. Using placeholders has the effect of restructuring the executable workflow. Tasks in the workflow are grouped or clustered together, and each cluster is executed using a single placeholder.

The approach used for clustering tasks in Pegasus is to cluster tasks that are at the same depth in the workflow. A level is assigned to each task in the workflow. The tasks in the workflow that do not have any predecessors are assigned level 1. Tasks that become ready for execution when the tasks at level 1 complete are assigned level 2, and so on. An important property of this level assignment is that the tasks at the same level are independent of each other and can be part of a single cluster. Tasks at the same level are clustered based on a clustering factor provided by the user. The placeholder implementation to be used for the cluster (sequential or MPI-based) is also indicated by the user. The clustering factor and the placeholder implementation are specified by the user in the Transformation Catalog (Section 23.3.2). Task clustering in Pegasus is performed after the tasks have been mapped to the execution resources, and each cluster contains tasks that have been mapped to the same resource. Thus the mapping drives the clustering in Pegasus and not the other way around. This is an important distinction since we only want to cluster tasks based on the characteristics of the target resource. A resource that has been dedicated for the user might not require any clustering, whereas a resource that provides a best-effort service may provide good performance only with task clustering.

In order to illustrate the performance benefit of using clustering, we describe the execution of a Montage workflow (Chapter 3) with and without clustering on the NCSA TeraGrid Linux cluster [412], containing 890 Intel Itanium 2 processors. The Montage workflow used for experimentation created a two square degree mosaic of the M16 region of the sky. The original executable workflow without any clustering had 1513 tasks and took 104 minutes to complete. We clustered tasks in the workflow with a clustering factor of 60, where each cluster contained 60 tasks at the same level. The restructured workflow contained 35 clusters and completed in 40 minutes, a 61% improvement over the original completion time [116]. The placeholder implementation used was a simple shell script that executed the tasks in the cluster sequentially.

While we have implemented a simple clustering strategy, more sophisticated algorithms are possible that can cluster tasks with precedence

constraints between them. It is important to note that the placeholder implementation should then be intelligent enough to ensure that the tasks in the cluster are executed in the right order. The placeholders that we have mentioned are passive in the sense that the list of tasks to be executed is provided as an input data to the placeholder and is known before the placeholder is submitted to the resource queue. Placeholders can also be active in the sense that they are not tightly bound to any particular task or set of tasks but instead can query for work to a central server or manager when the placeholder starts execution [358]. For example, Condor provides a placeholder implementation known as Glide-in [96] that can be used to start Condor daemons on remote resources. The Condor daemons query a central manager for tasks to be executed. For the duration that the Glide-in placeholder is executing, the remote resources appear to be part of a Condor pool. Pegasus provides support for generating executable workflows that can be executed using these temporary resources. However, the Glide-in job has to be initiated manually, and the automation of this provisioning step is the subject of future research.

We are not limited to the choice of active and passive placeholders. In fact, both can be used simultaneously. We have conducted experiments [234] that show the performance achieved using Pegasus when tasks are clustered and executed on resources acquired using Condor Glide-ins. We conducted experiments using the Montage workflow, and the resulting performance compares favorably with an MPI-based implementation (Chapter 3). The experiments show that using the proper set of optimizations, the performance achieved using the workflow paradigm could be comparable to the performance achieved using other parallel programming paradigms that are not Grid-aware.

23.8 Applications

Pegasus is currently distributed as part of the GriphyN Virtual Data System (vds.isi.edu). In addition to Pegasus, VDS contains an abstract planner [148] that can generate a workflow instance based on a VDL description (Chapter 17). VDS also contains a variety of Grid utilities, including data transfer tools, workflow visualization tools, and kickstart, which is used to launch the executable at the resource and gathers provenance information. The provenance information can then be stored in the Provenance Tracking Catalog for future mining.

A large number of scientific groups use Pegasus, the VDS tools, and DAGMan to map and execute their scientific workflows in Grid environments. Following are some example applications.

Bioinformatics and biology. One of the most important bioinformatics applications is BLAST (Basic Local Alignment Search Tool), which consists of a set of sequence comparison algorithms that are used to search sequence

databases for optimal local alignments to a query. Scientists used Pegasus to perform two major runs. One consisted of 60 genomes and the other 450 genomes, each composed of 4000 sequences. The runs produced on the order of 10,000 jobs and approximately 70 GB of data. A speedup of 5–20 times was achieved using Pegasus not because of algorithmic changes but because the nodes of the cluster were used efficiently by keeping the submission of the jobs to the cluster constant—basically automating the analysis process.

Tomography. In this application, 3D structures are derived from a series of 2D electron microscopic projection images (Chapter 8). Tomography allows the reconstruction and detailed structural analysis of complex structures, such as synapses, and large structures, such as dendritic spines. The tomography application is characterized by the acquisition, generation and processing of extremely large amounts of data, upward of 200 GB per run.

Astronomy. Astronomy applications are often characterized by a large number of short jobs. Among such applications are Montage and Galaxy Morphology. Montage is a Grid-capable astronomical mosaicking application (Chapter 3, [234]). It is used to reproject, background match, and finally mosaic many image plates into a single image. Pegasus is used in Montage as part of a service targeting the astronomy community. In Montage, Pegasus was able to improve the performance of the application by 61% (Section 23.7, [116, 234]). The Galaxy Morphology application [115] is used to investigate the dynamical state of galaxy clusters and to explore galaxy evolution inside the context of large-scale structure.

High-energy physics. Applications such as CMS [112] fall into the category of workflows that contain few long-running jobs. In one of the CMS runs, over the course of seven days, 678 jobs of 250 events each were submitted using Pegasus and DAGMan. From these jobs, 167,500 events were successfully produced using approximately 350 CPU-days of computing power and producing approximately 200 GB of simulated data.

Gravitational-Wave physics. The Laser Interferometer Gravitational Wave Observatory (LIGO) (Chapter 4) [6, 113] is a distributed network of interferometers whose mission is to detect and measure gravitational waves predicted by general relativity, Einstein's theory of gravity. In a Pegasus run conducted at the SuperComputing 2002 conference, over 58 pulsar searches were performed, resulting in a total of 330 tasks, 469 data transfers executed, and 330 output files. The total runtime was 11:24:35.

Earthquake Science. Within the Southern California Earthquake Center (SCEC), Pegasus is being used in the CyberShake Project ([276], Chapter 10), whose goal is to calculate Probabilistic Seismic Hazard curves for several sites in the Los Angeles area. The hazard curves in this study are generated using 3D ground motion simulations rather than empirically derived attenuation relationships. SCEC is running hundreds of analyses, some of which run over a period of several days. Pegasus was recently used to schedule SCEC workflows onto the TeraGrid resources [412]. During a period of 23 days in the fall of 2005, over 260,000 jobs, which used a combined 1.8 CPU-years, were executed.

23.9 Related Work

There are many workflow management systems for Grid environments. Triana (Chapter 20) is a visual workflow composition system where the workflow components can be service-oriented or Grid-oriented. It uses the Grid Application Toolkit (GAT) created by GridLab (www.gridlab.org) for distributing the workflow components across Grids. ICENI (Imperial College e-Science Network Infrastructure) (Chapter 24) is a system for workflow specification and enactment on Grids. The user creates an abstract workflow in an XML-based language. The ICENI system is responsible for making the workflow concrete by finding suitable implementations of the components in the workflow, mapping the components to appropriate resources, and monitoring the instantiation of the concrete workflow on the mapped resources. Once a schedule for the workflow has been computed, the ICENI system tries to reserve the resources at the desired time by negotiating with the resource provider. Taverna (Chapter 19) is the workflow management system in the myGrid project. The Taverna workbench allows users to graphically build, edit, and browse workflows. However, it is a domain-specific system and the workflows are limited to the specification and execution of ad hoc *in silico* experiments using bioinformatics resources. These resources might include information repositories or computational analysis tools providing a Web service based or custom interface. Workflows are enacted by the FreeFluo enactment engine, and progress can be monitored from the Taverna workbench. GridAnt (Chapter 21) is a client-side workflow management system that can be used for executing workflows on Grid resources. It extends Ant, an existing commodity tool for controlling build processes in Java, by adding additional components for authenticating, querying and transferring data between Grid resources. Furthermore, it provides a graphical visualization tool for monitoring the progress of the workflow execution. GridAnt is similar in functionality to the Condor DAGMan workflow manager (Chapter 22).

GridFlow [76], a Grid workflow management system, uses a graphical user interface for composing workflows. It assumes a hierarchical Grid structure consisting of local Grids managed by the Titan resource management and a global Grid that is an ensemble of local Grids. GridFlow simulates workflow execution on the global Grid in order to find a near optimal schedule. The best workflow schedule is enacted on the local Grids using ARMS agents. Unicore plus [431] is a project to develop a Grid infrastructure and a computing portal for users to access the Grid resources seamlessly. The Unicore job model supports directed acyclic graphs with temporal dependencies. The Unicore graphical tools allow a user to create a job flow that is then serialized by a Unicore user client and sent to a server for enactment. The server is responsible for dependency management and execution of the jobs on the target resources. Gridbus [489] is another workflow management system that allows users to specify workflows using a simple XML-based workflow

language. A workflow coordinator (WCO) is responsible for monitoring the status of the tasks in the workflow and activating the child tasks when they become eligible. An event service server (ESS) is used for notification purposes. Active tasks register their status with the ESS, which in turn notifies the WCO. Based on the status received from the ESS, WCO may activate the child tasks (similar to DAGMan functionality). It allows users to specify execution resources for each task in the workflow. Alternatively, it is also able to discover resources using Grid information services. Askalon (Chapter 27) is a Grid application development and computing environment that supports the development and optimization of workflow applications over Grid resources. It uses an XML-based AGWL (Abstract Grid Workflow Language) for specifying workflows. It supports a rich set of constructs for expressing sequence, parallelism, choice, and iteration constructs. It includes mechanisms for monitoring workflow execution and dynamic rescheduling in order to optimize workflow performance. Kepler (Chapter 7) is another project for composing and executing scientific workflows. It provides a graphical user interface for composing workflows. A workflow in Kepler is composed of independent actors communicating through well-defined interfaces. An actor represents parameterized operations that act on an input to produce an output. The execution order and the communication mechanisms of the actors in the workflow are defined in a director object. In order to support execution over Grid resources, Kepler has defined a set of Grid actors for access authentication, file coping, job execution, job monitoring, execution reporting, storage access, data discovery, and service discovery.

23.10 Conclusions

In this chapter, we described the Pegasus system, which can be used to map large-scale workflows onto Grid resources. Pegasus is a flexible framework that enables the plugging in of a variety of components from information services and catalogs to resource and data selection algorithms. Pegasus has been used in several applications and has mapped the workflows onto a diverse set of resources from local machines, to Condor pools, to high-performance TeraGrid systems. Pegasus is only one of the tools in the workflow life cycle; other tools are used to generate workflow instances, execute workflows, and record provenance. Pegasus relies on the existing Grid infrastructure to provide the information necessary for the planning process.

Even with today's application successes, many research issues remain open not only for Pegasus but other workflow management systems as well. Some issues touch upon improving workflow performance by developing better application and resource performance models, which in turn can help improve the planning process. The performance models are also necessary for accurate and cost-efficient resource provisioning.

In terms of execution and the interplay between the planner and the workflow engine, more research needs to target fault tolerance. As we discussed, Pegasus has some fault-tolerant capabilities; however, the issue of fault tolerance across workflow management systems is a greater one and involves a dialogue between workflow composition, workflow planning, and the workflow execution components.

Debugging is also a major issue, especially in environments such as the Grid, where errors are hard to detect and categorize. Additional complexity stems from the gap between what the user specified (possibly a very high-level analysis specification) and what is actually executed (a very low-level detailed directive to the Grid). Bridging this gap can be a significant challenge.

Finally, most of the workflow systems today involve a user specifying the workflow in its entirety and then the workflow management systems bringing it to execution. Providing support for interactive workflows poses great challenges where the interactions with the users need to be predictable in terms of time scale. Thus, real-time performance and quality-of-service (QoS) guarantees are becoming very important.

Acknowledgements

ICENI

A. Stephen McGough, William Lee, Jeremy Cohen, Eleftheria Katsiri, and John Darlington

24.1 Introduction

Performing large-scale science is becoming increasingly complex. Scientists have resorted to the use of computing tools to enable and automate their experimental process. As acceptance of the technology grows, it will become commonplace that computational experiments will involve larger data sets, more computational resources, and scientists (often referred to as *e-Scientists*) distributed across geographical and organizational boundaries. We see the *Grid* paradigm as an abstraction to a large collection of distributed heterogeneous resources, including computational, storage, and instrument elements, controlled and shared by different organizations. *Grid computing* should facilitate the e-Scientist's ability to run applications in a transparent manner.

As scientists become more confident with the new emerging Grid paradigm, their requirements and expectations of the Grid are increasing. Initially their requirements were to deploy simple applications over the Grid, often by manual selection and invocation of the resources used. However, this is now evolving into the need to deploy the whole e-Science process onto the Grid without the need for intervention by the e-Scientist except where desired. The e-Scientist may then interact with the running application by monitoring its progress or, if possible, visualizing the output and/or steering its execution.

To manage the complexity of these e-Science processes, we need a means to describe them. Each individual task needs to be specified along with the way in which multiple tasks interact (often referred to as a *workflow*). However, the type and level of this description may vary dramatically, depending on the scientist's domain knowledge. To this end, we define two realms for workflow description:

- *e-Scientist's conceptual workflow.* This describes the tasks identified by the e-Scientist and the interactions required between these tasks.

- *Middleware workflow.* This is the actual set of tasks that are required to produce the e-Science workflow along with the interactions between them.

In the simplest case, these two workflow descriptions will match, though this is not always the case. For example, suppose that the e-Scientist wishes to manipulate a database. In the conceptual workflow, there may appear tasks such as "Record result in database" or "Retrieve result from database." However, these are unlikely to appear as separate computational tasks between software interacting on different resources, though they may appear as interactions between a computational task and a database.

We propose an abstract architecture (referred to as a *workflow pipeline*) that is used to automatically progress an e-Scientist's conceptual workflow into a middleware workflow and then through to an enacted workflow upon the Grid. Our architecture provides for flexible deployment, charging, execution performance, and reliability. These are addressed both at the specification level and at the realization and execution levels.

24.1.1 Definitions

We use the term *application* to denote a composition of components and services defined using a *workflow language.* A *component* is an indivisible unit of execution in which all the contextual, functional, and behavioral aspects are made explicit. These include the implemented functional interfaces, behavioral description (e.g., performance characteristics), resource dependencies, and runtime requirements. While multiple *components* might exhibit the same functional interface, they can be independently implemented, yielding different behavior [284]. An application therefore consists of a number of components linked together in a manner that performs the overall task required by the e-Scientist. Figure 24.1 illustrates a simple application.
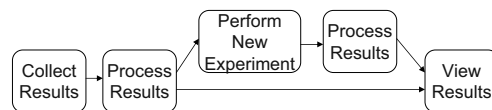
Figure 24.1: A simple application consisting of components connected into a workflow.

Without loss of generality, a component may be viewed as an atomic unit or the composition of a collection of other components — a *supercomponent.* These collective components allow lower-level details to be encapsulated and higher-level functionality to be exposed. In our definition, a *component* is an abstract entity that awaits composition by *application authors* and subsequent deployment into an *execution environment.* We make no assumptions in this work as to the exclusivity of a service on a resource.

A deployed *component* is referred to as a *service*. A *service* is a realized manifestation of the states and functions provided by the underlying *resource* and the *component* implementation. It presents a standardized and interoperable set of renderings (e.g., Web services) of the exposed functions. The definition does not mandate a particular *component* implementation architecture. It only requires the use of an interoperable set of protocols (e.g., SOAP) and data models (e.g., WSDL, XSD schema). This allows *applications* to be composed with both abstract *components* that need to be provisioned and anchored *services* that exist irrespective of the lifetime of the *application*. It is the role of the *workflow pipeline* to realize an *application* by performing component implementation selection, resource allocation, deployment, and orchestration.

We define *application execution time* as the time required to execute the application. This is the time from the start of the first component until the end of the last component. We define *workflow pipeline execution time* as the active time spent within the workflow pipeline.

24.1.2 Background

Figure 24.2 illustrates the layers within a workflow pipeline based on the pipeline developed at the UK Workflow Workshop 2003. Below, we outline the functionality of each of these layers:
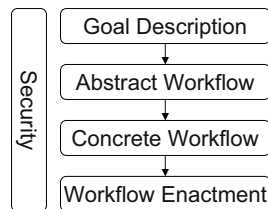
Figure 24.2: The layers within the workflow pipeline.

- *Goal description.* This is the e-Scientist's conceptual workflow.
- *Abstract workflow.* The conceptual workflow is mapped down into an abstract workflow described in terms of its meaning [284] — what each component will do rather than how it will achieve it — along with how the components are interconnected, for example, a linear equation solver taking inputs from an experimental device and sending the results back to the user.
- *Concrete workflow.* Each component is matched with specific implementation and resource instances. Data exchange formats are defined and reservations are made. For example, a Cholesky solver will be used, that

has an interface that takes a matrix of 32 bit integers and calls a method on an interface that takes an array of 32 bit integers.

- *Workflow enactment.* The instantiation of this workflow onto the Grid. The execution of all the components and the coordination of this process.

The model above is too restrictive for use within the Grid. Rarely will a workflow pass cleanly through these four stages. It may be desirable to keep parts of the workflow abstract at the same time as others are concrete (or even enacted). Likewise, parts of the workflow may revert back to previous states due to the ever-changing conditions within the Grid (resource failure/availability). Thus our workflow pipeline merges the abstract and concrete stages into the *realized workflow* stage and allows parts of the workflow in the enactment stage to revert back to this new stage. The workflow may be altered in light of the changing Grid, while executing components may be migrated in order to better achieve the desired results of both the e-Scientist and the owners of the resources.

24.1.3 Requirements for an End-To-End Workflow Pipeline

Many coupling frameworks and component-based languages have been specified for dealing with workflows. Some of the better known ones include the Bespoke Framework Generator (BFG) [417], Business Process Execution Language (BPEL) [24], and Abstract Grid Workflow Language (AGWL) [138]. However, although all the works above are very significant in the field, they do not provide complete solutions for capturing all the requirements to provide a fully integrated end-to-end pipeline. Here we define the higher-level functionalities that we are building into ICENI II. Where appropriate, we are integrating these features into the existing standards, such as BPEL.

- *An information-rich environment.* Although many component and workflow models allow for the annotation of components, this tends to be at the syntactic level. We advocate the collection and use of higher-level annotation (semantic) information on components and the workflow language. In previous work [284, 285], we have shown the annotation of components in terms of *meaning*, *behavior*, and *implementation*, where multiple implementations may share the same behavior and multiple behaviors may share the same meaning. This information is key for the following high-level services.
- *Tools for capturing and manipulating the rich annotation.* Without appropriate tooling to achieve the collection of this information, most component developers will only provide the minimum amount of information for the component to be valid. We see the development of such tooling as key to the success of this work.
- *Tacit scientific realm annotation.* e-Scientists have a vast knowledge of the scientific realm in which they work. This information can be captured and used to provide appropriate annotation on the meaning of a workflow.

- *Component encapsulation and abstraction.*   By allowing application developers to design their applications in terms of meaning, this removes the implementation tie-in. Applications can be deployed to the most appropriate set of resources using the most appropriate implementations — perhaps ones of which the application author was unaware' – thus making the application much more portable.
- *Workflow validation.* There is little point in attempting to execute a workflow if it will fail at a later stage due to incorrect construction. Therefore, the workflow is validated at both the *syntactic* and *semantic* levels before it is executed.
- *The use of problem-solving environments.* Problem-solving environments [134] are capable of taking environment-specific information and using this to reason on the validity of a workflow at both the syntactic and semantic levels. This may also be used to determine if manipulations of the workflow will semantically alter its meaning.
- *A general coupling framework.* At a naive level, components may be coupled together by defining components with appropriate "in" ports and "out" ports and binding the appropriate ports. This, however, assumes that the components are distinct enough and designed to be combined in this manner. Often, scientific components will have high levels of inter-dependence and/or generate output that is in an inappropriate format for direct connections. By using of component annotation, it is possible to couple components closely together with high levels of interdependence. Shims (translation components) can also be automatically inserted into workflows as appropriate.
- *Component hierarchies.* An application may itself be considered as a component deployable within a larger workflow as a *supercomponent.* Most workflow systems treat these supercomponents as a black box — all information about the components within the supercomponent and the workflow description is lost to the higher-level workflow. A proper rendering of the supercomponent needs to be used within the workflow language.
- *The use of coordinated forms.* In order to be complete, we advocate that the architecture should also support the notion of a *coordination form.* In previous related work, Darlington et al. [104], used a structured coordination language and proposed different functional skeletons representing different parallel behavior. The skeletons can be composed to produce a high-level parallel abstract type. We are developing coordination forms that provide abstractions in component coupling and especially in the area of scientific model coupling.

24.1.4 ICENI

ICENI (Imperial College e-Science Networked Infrastructure) [283] originated from the research activities of Professor John Darlington and colleagues in

the 1970s and early 1980s in the development and exploitation of functional languages. The growth of applied parallel computing activities at Imperial College demonstrated a fundamental need for a software environment to enable the use of complex resources by the average scientist. This requirement became even more apparent with the growth and adoption of Grid computing within the United Kingdom (a significantly more complex environment than a single parallel machine) to enable computer-based research — e-research. The enduring goal of ICENI is to increase the effectiveness and applicability of high-performance methods and infrastructure across a whole range of application areas in science, engineering, medicine, industry, commerce, and society.

Our focus within ICENI therefore has three major elements: prototyping the services and their interfaces necessary to build service-oriented Grid middleware; developing an augmented component programming model to support Grid applications; and exploring the information needed to annotate the services and software to enable effective decision making about component placement within a Grid.

ICENI has now had exposure in the wider Grid community for nearly five years. The GENIE project has used ICENI in order to Grid-enable their unified Earth System Model, which has allowed them to vastly reduce their runtime [185]. The e-Protein project is using ICENI to control their gene annotation workflows [323]. The Immunology Grid project is using ICENI in areas of molecular medicine and immunology [212]. The RealityGrid project has used ICENI in order to coordinate the deployment, steering, and visualization of its workflows [368].

We are currently revising the ICENI architecture, as ICENI II, in light of our experiences with these projects. The proposed architecture is derived from our previous work with the ICENI pipeline along with the changing trends within the wider community. We abide by state-of-the-art software technology standards. We have adopted Web services as a distributed component middleware technology, as they have a set of features that are well suited to our needs.

24.1.5 Related Work

Many groups are developing systems to deal with workflows [490]. Here we highlight a number of projects that we feel are significant and compare them with our approach.

The *Component Architecture Toolkit* [442] shares the common vision of a component programming model. It follows the Common Component Architecture (CCA) specification [30], in which components expose a set of typed "ports." Components can then be joined to form larger applications by connecting type-compatible "ports." The system has been implemented in both HPC++ [63] and Java. The motivation is to provide a common set of familiar APIs to component developers so that the underlying

network intricacies can be abstracted. Resource selection in the Component Architecture Toolkit is tightly coupled with the composition and control tools and often driven by the end user. The component implementations are not designed to interoperate.

Triana [408] is an integrated and generic workflow-based graphical problem-solving environment that handles a range of distributed elements such as Grid jobs, Web services, and P2P communication. The distributed components considered by Triana fall into two categories: Grid-oriented and service-oriented. Furthermore, Triana and the Visual GAT represent explicitly Grid operations such as job submission and file transfer (by the introduction and use of "job components" and "file components"). Although Triana can be integrated with and operate over a range of Grid technologies, such as resource managers and data management systems, it only focuses on the implementation of an environment for the specification and execution of a component workflow. This means that other aspects such as optimization, interaction with resource managers, and data management systems are not addressed by the framework.

Taverna [473] is the workflow editor environment for the MyGrid [396] project. Taverna and the MyGrid middleware are designed for performing *in silico* experiments in biology. We see Taverna as a good example of the need to provide scientific domain-specific environments and as such feel this fits in well with our specification layer. However, the MyGrid middleware lacks the facility to perform scheduling; instead, it looks up anchored services that the user can compose.

Kepler [19] is similar to Taverna, providing a workflow editing environment with the ability to invoke Web services. Although Kepler provides a useful way to model workflows, it lacks the ability to adapt these workflows in the presence of changes in the Grid.

The Virtual Data System (VDS) [29] (formally Chimera) is a set of tools for expressing, executing, and tracking the results of workflows. VDS aims to abstract the workflow from the details of implementation, such as file location and details of the programs to be deployed. As such, VDS fits well with our notion of abstract workflows that are realized down to executing components through an automated system.

The Open Grid Services Architecture (OGSA) [325] working group within Open Grid Forum (OGF) [324] (formally the Global Grid Forum (GGF) [159]) is chartered to define an architecture for the Grid. Although OGSA has yet to address the issues of workflow fully within its work, there is great synergy between our work and that of OGSA due to the active participation within this working group.

In Section 24.2 we present the abstract architecture for our workflow pipeline followed by a more detailed breakdown of the three levels in Sections 24.3, 24.4 and 24.5. We illustrate how this architecture can be used in collaborative environments in Section 24.6 before concluding in Section 24.7.

24.2 The Workflow Pipeline
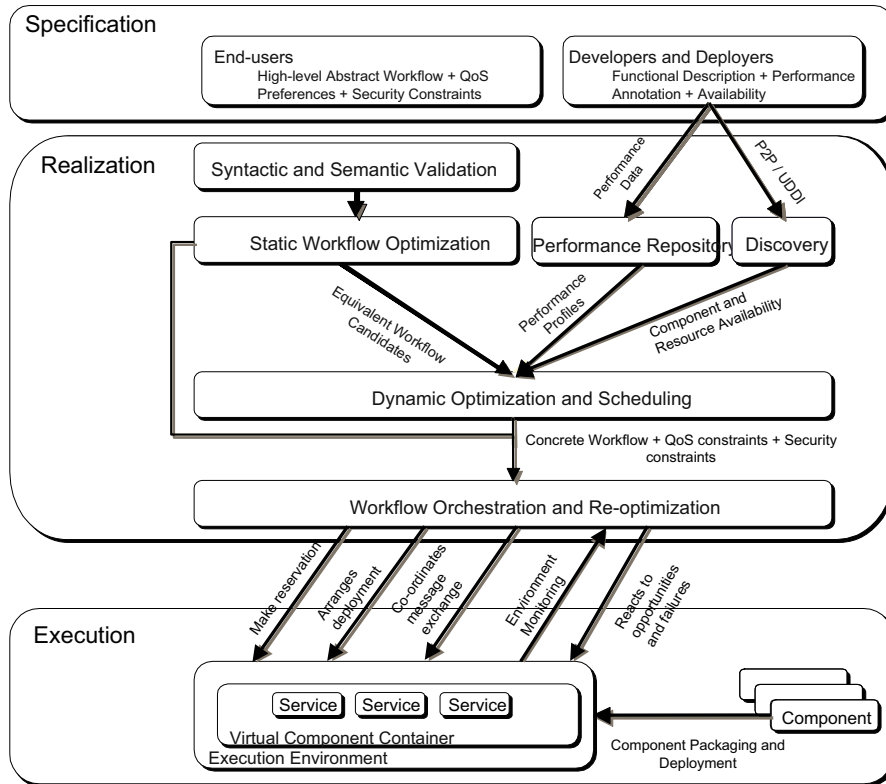
Figure 24.3: The workflow pipeline.

Here we define the architecture for our workflow pipeline. The stages of the workflow pipeline can be grouped into three main areas — *Specification*, *Realization*, and *Execution* (Figure 24.3). Below is an overview of each of the stages, which are discussed in greater detail in subsequent sections:

Specification: (goal description). This stage deals with the generation of the workflow: the language used, the representation of the workflow to the e-Scientist, and how the e-Scientist specifies any requirements on the execution of the workflow (quality of service). e-Scientists may never see or be aware of the underlying language of the workflow, nor indeed that they are specifying a workflow. This stage will produce an abstract workflow.

Realization (abstract and concrete workflows). The aim of this stage is to thoroughly validate the workflow and then map its elements to concrete resources and implementations in preparation for execution. This

is a nontrivial process that may be both computationally intensive and time-consuming. As a result, this stage begins by carrying out various optimizations to the abstract workflow.

Execution (workflow enactment). This is the execution of the workflow (or parts of it) on Grid resources. This is not just a process of deploying component implementations to resources. The execution stage includes a middleware layer tasked with monitoring the progress of components, providing an environment (container) in which the components can execute, and providing functionality to migrate components as appropriate.

24.3 Specification

The specification stage can be seen as the e-Scientists' opportunity to specify the application that they wish to achieve. Key concepts in this stage are *end-user programming*, *workflow reuse*, *code reuse*, and *adaptability*.

End-user programming. End users are typically not specialists in Grid computer software or architectures, nor should they be forced to become so. e-Scientists should be able to specify their requirements in a format directly applicable to their field of expertise, which is then mapped to the workflow language. Eres et al. [134] have shown that scientists can develop their ideas in systems such as Matlab [423], which can then submit this work to the Grid. Triana [408] shows an example how a well-tailored interface can be used to assist a bioscientist in using the Grid. This is of course the ideal scenario. As there are many scientific domains, we cannot hope to develop suitable specification environments tailored to all. We are instead developing exemplar environments for the scientific domains with which we are involved, such as the e-Protein project [106]. Experts in other areas are encouraged to develop suitable environments. These specification environments are assumed to be as expressive as a workflow environment, with the ability to edit and manipulate specifications as well as archive them for future use or editing, thus providing *workflow reuse* and *adaptability*.

Code reuse. There exists much valuable legacy code and numerous software libraries, many of which are specialized for particular architectures. This legacy code needs to be wrapped within component technology so that the e-Scientist can compose it along with new code as part of a workflow.

24.3.1 Workflow Validation

Workflow validation is aimed at reducing the chance that an invalid workflow will commence on the Grid. It is of no benefit to submit a workflow that will fail or return incorrect results to the user. By using the rich annotation information we collect in ICENI, the abstract workflows can be validated before being submitted to the Grid. At the most simple level, this can be to check that the workflow can actually be fulfilled at a syntactic level, though
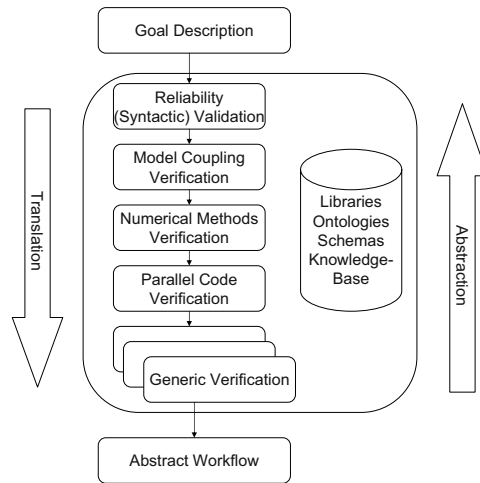
Figure 24.4: Specification and deployment.

it may also be used in an attempt to ensure that the result of an application has a valid meaning (semantic level). Figure 24.4 defines some of the layers of validation that can be performed. We are currently developing [333] a semantic validation service for the GENIE project [171].

Validation of the workflow starts by checking the syntactic correctness of the workflow. The workflow will fail if a component that requires an input on a port cannot receive that input — we assume that data coming out of a component can be discarded. Any mandatory input port that is not connected is a syntactic failure, and the workflow is returned as unexecutable. It is also necessary to check that the output data type at one end of an interconnection matches the input expected at the other end. Existing "component knowledge" can also be utilized when testing for syntactic correctness. It may be known that a specific component outputs a matrix of size $x \times y$, while the component receiving the input of this matrix requires a matrix of dimensions $s \times t$ (where $x \neq s$ or $y \neq t$), again a syntactic failure.

Once a workflow has passed the initial set of syntactic correctness tests, it can be assumed that it is capable of passing into the scheduling stage. However, this is no guarantee that it is carrying out the set of tasks that the composing scientist intended. At the semantic correctness testing stage, the aim is to use knowledge about common component groupings, sets of scientific methods that cannot successfully interact, and other domain-dependent knowledge to diagnose situations where a scientist may have made an error within his or her workflow definition. This stage may be considered akin to a grammar checker for component-based e-Science applications.

It is not always possible to perform this stage due to a lack of model information about the components that are used within the application.

However, significant work has been done with modeling mathematical operations in order to achieve this result [273].

24.4 Realization

The resources and software implementations now need to be selected before the workflow can be enacted on the Grid. In the simplest case, components need only be assigned to resources and implementations. However, to make the best use of the resources in the Grid and to satisfy the e-Scientist's requirements, a process of scheduling (brokering) is employed.

Most scheduling algorithms do not attempt to search the entire problem space to find the optimal concrete mapping of components or processes to resources since this is an NP-hard problem. Instead, heuristics are used to approximate the optimal solution. Our aim is to map component implementations to resources, forming an execution plan that is efficient, both in terms of application execution time, and in terms of workflow execution pipeline time. Not all components need be deployed at the same time: just-in-time scheduling of components and the use of advanced reservations help to make more optimal use of the available resources for all users of the Grid.

24.4.1 Resource Discovery

In Grid computing, services are increasingly used to model resources for e-Science activities. Resources include not only computational systems but also assets (data storage and instruments) and knowledge (database, advice, and brokers). The service encapsulation of resources provides a high-level abstraction of the functional capabilities offered by the visualized resources. This level of abstraction permits the e-Scientists to focus on the functional composition of services to create applications, while allowing Grid middleware to handle the complex mapping and instantiation of software components onto available resources.

At each stage of the pipeline, user and system actions are guided by the resource information made available by resources and services. At the specification phase, users are largely interested in the abstract functional interfaces of the services available for composition. In the context of Web services, the functional interface is often described in the Web Services Description Language (WSDL). Clients can discover services implementing a particular interface defined in WSDL by the name of the port type. This syntactic approach limits the query result to an exact match because WSDL lacks any inheritance model. Services that differ syntactically but offer equivalent functionality are missed by this mode of discovery.

The Semantic Web technologies and the standardization of ontology languages such as the Web Ontology Language OWL [331] lay the foundation for a reasoned approach in logically describing relationships between concepts

and their properties. The development of the OWL-S Ontology [332] is an effort to model the constituents of a Web service with the OWL constructs. It consists of the main concepts *ServiceProfile*, *ServiceModel*, and *ServiceGrounding*. They respectively model the capabilities of a service in the form of *precondition, input, output and effects*, a detailed perspective of how the service operates as a *process*, and finally the mapping between the abstract profile and a concrete specification. The semantic service description augments the syntactic definition of messages defined in a WSDL document in relation to the constituent parts of the conceptual description.

This ontological approach is far superior to discovering services by the name of the type. Clients can take advantage of automated reasoners to query a service registry by the subconcept relationship (e.g., biconjugate gradient is a specific method for solving linear equations), equivalence (e.g., job submission services capable of executing the x86 or Intel instruction set), and instance checking (e.g., is this service a job submission service capable of executing the x86 instruction set?) The use of semantic information in the discovery process has been demonstrated in many e-Science exemplars [87, 267]. Although this approach has many obvious advantages the expressiveness of the language has the apparent penalty of loss of efficiency due to its computational complexity. In the realm of the Grid, where knowledge about the services is vast and distributed, a trade-off between expressiveness, efficiency and completeness [188] often has to be made. The usability of a modeling language presents another barrier to using ontologies to describe resources and services.

The ICENI workflow pipeline is agnostic to the mechanisms for information dissemination and gathering. Information can be stored in a centralized registry such as a UDDI [430] directory, frequently used in Web services, or disseminated across a peer-to-peer network [310]. The decision is driven by the nature of the information, such as frequency of change, natural distribution across the network, trust and the need for authoritative verification.

In particular for performance optimization, resource information such as hardware configuration, CPU load, I/O load, available network bandwidth, memory, and storage are the main parameters for the scheduling decision. While static information is best suited to be stored and queried from a central registry, dynamic information that the scheduler might need to build up a historical profile of the resource is more suited to being delivered in a publish–subscribe push model.

24.4.2 Static Workflow Optimization

The Static Workflow Optimization service is responsible for pruning and manipulation of the workflow in order to preoptimize the runtime execution. Using static information about the components, this service accepts as input an application's workflow and produces a preoptimized workflow that is expected to execute more efficiently in the Grid. This stage performs best when an abstract workflow describes the components in terms of their meaning.

Here we do not consider the dynamic load on systems within the Grid. The manipulations that can be performed include:

- *Reordering of components.* It may be possible to reorder some of the components within a workflow to improve efficiency.
- *Insertion of additional components.* This allows translation components to be added into the workflow to convert the output from one component into the desired format for the next component.
- *Workflow substitution*: A workflow may contain a subset of components that is semantically equivalent to an alternative subset that is known to be more efficient.
- *Pruning redundant components*: Workflows, especially those that are composed from supercomponents, may contain components that serve no purpose toward the final result. These components can be identified and removed.
- *Component substitution*: It may be possible to use information about the data that will arrive at the component to select the most appropriate implementation. For example, for a finite-element solver, with an input matrix that is sparse and diagonally dominant, it would be more appropriate to select a Conjugate Gradient Solver over a Jacobi Solver.

24.4.3 Prescheduling Runtime Optimization

Using complex scheduling algorithms to attempt to schedule several components over what may be millions of resources is in itself an NP-hard problem. By using simple general knowledge about the user, workflow, and component requirements, we can prune the search space, thus simplifying the scheduling task. We take the approach that no dynamic resource information can be considered at this point. Note that this stage is normally performed through the use of lazy evaluation. The techniques we propose for this stage are (listed in order):

- *Authorization.* If a user is not allowed to use a resource or software implementation it can quickly be removed from the potential search space.
- *Hardware/software requirements.* This stage is performed by many brokering and scheduling systems. Resources can be pruned from the tree if they don't match the minimum requirements (e.g., processor type).
- *Problem-specific requirements.* Many components have specific requirements (e.g., long execution time without the ability to checkpoint), thus allowing more resources to be pruned. Daily et al. [103] proposed that "closeness" of resources be taken into account when communication between components is significant.
- *Out-of-bounds selection*: Although a resource may match the minimum requirements for a component implementation, knowledge about the intended use may render it inappropriate. For example a 90 MHz Pentium

processor may be able to run a linear solver; however, if the number of unknowns is large, it can be pruned at this stage.

24.4.4 Workflow-Aware, Performance-Guided Scheduling

ICENI provides a framework for plugable schedulers [488]. Schedulers may be designed to be workflow aware. This has been implemented within ICENI [287]. Thus the scheduling of components depends not only on the performance of a component on a given resource but also on the effect this will have on the other components in the workflow. Schedulers may support the notion of just-in-time evaluation, allowing only those components currently required to be scheduled. Other components in the workflow are left unscheduled until required. Described below are the general steps taken to evaluate a suitable mapping of components onto resources.

As the components that make up the abstract workflow only describe the meaning of what should be carried out (we define this to include the data flow between components), the first task of the scheduler is to match these component meanings with component implementations. The scheduler can speculatively match implementations with resources.

Performance information [288] can be used to obtain estimates on the execution times for these combinations, the duration of the entire application, and the time at which each component is expected to begin execution. Performance data can be used to determine if the application will be able to meet the *quality of service* (QoS). The critical path of the application can also be determined [288]. This will allow greater flexibility for selection of component implementations and resources for those components not on the critical path.

Performance estimates along with resource discovery information allow the scheduler to determine the valid mapping of the components over the resources. The scheduling algorithm can then select a set of equally optimal mappings. The predicted component start times for each realized workflow can then be passed to the reservation system, which responds with a single realized workflow, including any reservations it was able to make.

Many current Distributed Resource Management (DRM) systems operate as a simple queue system. This makes the determination of the time when components will start executing difficult. Previous work has shown that it is possible to use gathered execution times to predict future execution times. This can be used as a wrapper around existing DRM systems to help predict the start time for components on such systems [288]. Hovestadt et al. [199] propose a system where a planning system on a DRM sets a start time for each task, which can be used to provide an accurate start time for each component. The work by Hovestadt may also be used to implement advanced reservations.

Furthermore, as our system monitors the execution of an application as it progresses, it is able to react to potential problems (and breaches of QoS). It may determine that a component won't start execution as planned and may

reselect the resource for deployment, selecting the "best" realized workflow — as defined by some user-defined criteria — of the factors that are important to them. This could be based around quickest execution time, cheapest execution (where resources are priced), or some other metric or combination of metrics. The techniques for combining these metrics and accurately modeling the requirements of users, resource owners, and Grid managers are an area of current research [487].

A number of scheduling algorithms have been developed for use in ICENI. These include random, best of n random, simulated annealing, and game theory schedulers [487]. We are currently developing schedulers based around constraint equations solved by mixed integer linear programming.

24.4.5 Just-in-Time Scheduling/Deployment

In many workflows, it may be beneficial not to map all the components to resources at the outset. This may be due to the fact that it is not possible to determine the execution time of a given component until further information is obtained by running previous components in the application. It may also be desirable to delay mapping until a later time in order to take advantage of resources and/or component implementations that may become available during the lifetime of the workflow.

Full-ahead planning, where all components are assigned to resources at the outset, tends to be more useful for applications that contain only a critical path. This is especially important if time constraints are critical.

Certain deployment environments are capable of handling advanced reservations (see below), in which case the scheduler will allocate a resource for the component to run on but the deployment of the component will not happen until the reservation is active.

The information held about a component implementation indicates whether a component can benefit from just-in-time scheduling and/or advanced reservations. The scheduler may then decide to ignore these components for the purpose of mapping. When the rest of the components are deployed to resources, all components that are not currently mapped to a resource are instantiated in a virtual space — referred to as the "green room". Components in the "green room" are able to communicate with other instantiated components. The only valid operations that may be called on components held in the "green room" are those that add configuration data to the component; any call that requires computation will cause the component to be scheduled.

Scheduling of components contained in the "green room" can be triggered by one of two events. If a component already running tries to communicate with a component in the "green room" with more than a configuration call, then the component will trigger a scheduling operation. Alternatively, the scheduler, which is aware of the time when a component should be required, can preemptively start scheduling so that the component is made real (just)

before it is required. Components that hold advanced reservations will remain in the "green room" until the start of their reservation slot. At this time, the components will be deployed onto the resource that contains the reservation.

As multiple components may exist on the same resource at any given time, it is always possible to select a resource on which the component may be deployed. If only resources that are available through queues are available, or time is required to deploy the component, then any data sent to the component are buffered during this period. However, it should be noted that these two circumstances should arise infrequently, as this time lag to deployment should have been predicted during the planning stage and the component realized before this point.

24.4.6 Advanced Reservations

Due to the uncertainties of resource and network availability in a dynamic system such as the Grid, it is necessary to support advanced reservations to provide QoS guarantees. Reservations may be made on computational resources, storage resources, instruments, or the underlying fabric of the Internet, such as network links. The reservations may be made for exclusive use of the entity or, in some cases, some pre-agreed proportion of it, although currently few deployment systems support advanced reservations.

It should be noted that not all components require reservations, nor do all resources provide reservations. However, component-resource pairings are only selected without reservations if time constraints are not critical for the application or the components without reservations are considered distinct enough from the critical path as not to affect it. These components are monitored during the life cycle of the application in order to ensure that they don't become a problem for the overall application.

Although many DRM systems currently don't support reservations, a number of techniques exist to "simulate" the same effect. These include the launching of fake jobs that are submitted to the queue prior to the reservation time, taking possession of the resource prior to the start of the reservation period. There is a trade-off here between the utilization of the resource and the ability to guarantee the reservation. Taking possession too early will prevent other jobs from running, while releasing a possession a long time before the reservation time may be wrong if it proves to be impossible to obtain another possession before the reservation interval.

24.5 Execution Environment

At the end of the optimization stage, the abstract workflow (or subworkflows) presented by the user is transformed into a concrete workflow where the software selection and resource arrangement decisions have been made. This decision might be revoked when opportunity or failure arises in which some
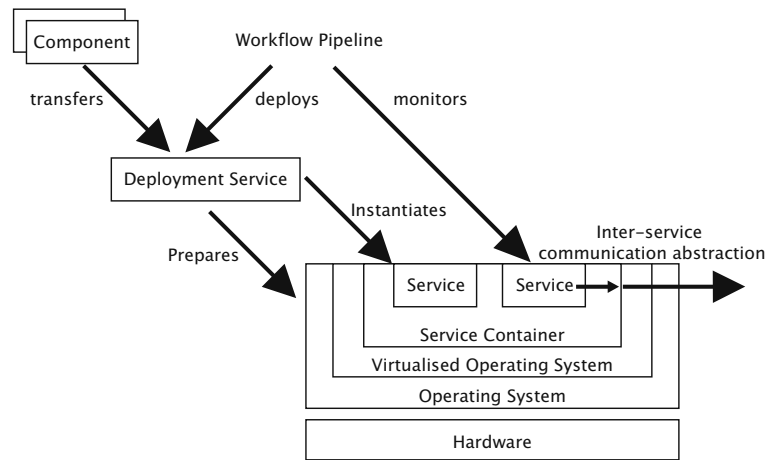
Figure 24.5: Execution environment and multi-level visualization.

parts of the application shall be relocated for efficiency reasons or as a means of recovery. Moreover, parts of the application can be instantiated in a just-in-time fashion according to the temporal analysis performed in the optimization stage. The tentative resource allocation decision of those components can be refined while the application is executing.

The execution environment represents the visualization of the resource that manages the life cycle of the parts of an application (Figure 24.5). The execution environment encapsulates the facilities available to the software component, such as intercomponent communication, logging, monitoring, failure recovery, checkpointing, and migration. These facilities are exposed to the software component through a set of abstract APIs. These abstractions allow the execution environments managing the parts of an application to co-operate and coordinate their runtime capabilities, such as network transport, colocation, and shared file system. Software engineers developing the components are insulated from the implementation choice made by the optimization stage by following the software patterns offered by the APIs. This is analogous to the MPI [179] abstraction for message-passing in parallel applications.

The software component instantiated in the execution environment is referred to as a service. We adopt Web services as one *view* of the running software component. It is an ideal way for services on different physical resources to communicate with each other in an interoperable manner. The elements in the execution environment will be discussed in more detail.

24.5.1 Component Deployment

The ICENI deployment service is the gateway to a computational resource. It is responsible for facilitating the provisioning and instantiation of a component assigned to a particular resource. First, the deployment service prepares the execution environment. This might involve the preparation of a component container in a cluster resource. Recent advances in visualization technologies [38, 433] offer visualization at the operating system level. Within the visualized operating system, a component container provides the higher-level abstraction to the software component on top of the operating system facilities. The compartment model offers attractive features such as security and fault isolation. Multilevel visualization allows runtime facilities to be flexibly configured depending on the deployment requests [388]. Although visualization provides a sandbox environment in which a component can execute seemingly exclusively, the cost in instantiating the container on-demand [235] may be too high for short-running components. Predictive instantiation might alleviate the setup cost by allocating resources in advance.

Once an execution environment is available, the deployment service will facilitate the provision of the software component onto the resource. This might involve the staging of software packages and their dependencies available remotely into the system. In order for this architecture to succeed across the Grid, a standardized interface for deployment and a language for software requirement description are essential. This reduces the need for users and software agents to understand a large number of description languages and deployment mechanisms to exploit a variety of Grid resources.

The Job Submission Description Language (JSDL) [227] is being defined in the Open Grid Forum (OGF) [324]. Although currently focused on describing deployment of a traditional POSIX application, an extension has been proposed for describing software components for Java enterprize compliant containers and others. Configuration Description, Deployment and Lifecycle Management (CDDLM) [86] is another standard effort focusing on the generic description and life cycle control of components.

An ICENI component can utilize the GridSAM job submission service [251,418] exposed through an interface provided by the execution environment for launching legacy software in the form of a POSIX application. We have developed tools to generate ICENI components from a description of the POSIX application expressed in a template JSDL document. These components can therefore expose an application-specific Web service interface that can take part in the Web service workflow orchestration while hiding the legacy details. Previous related work (Furmento et al. [94]) has shown how legacy code can be wrapped up as components using automated tools and then used within a workflow.

24.5.2 Checkpointing and Migration

Checkpointing is a technique for preserving the state of a process in order to reconstruct it at a later date. It is a crucial element for providing fault recovery from a saved state. In scientific applications, checkpointing provides a means for long-running simulations to be restarted at a previously examined parameter space [90]. This is also an important means for migrating the state of a process to another execution environment. An execution environment that provides checkpoint facility allows ICENI to reschedule and migrate a running application when the opportunity arises based on the monitoring information collected at runtime. In addition, migration can be initiated by a user wishing to steer an application according to performance and colocation concerns, typical in a simulation involving collaborative visualization [89, 355].

ICENI acts as a management layer on top of checkpointing and migration systems such as OpenMosix [424], OpenSSI [425], and Kerrighed [422]. It is worth pointing out that not all components can be checkpointed. Checkpointing and migration schemes can be classified into three broad categories. The generality of the approaches increases from one technique to the next. Application-level checkpointing might be initiated by the application itself through a checkpointing and migration API. This provides fine-grain control to the developer to save the states of the application at a critical moment of the execution; however, it requires that existing applications be modified to take advantage of the functionality. Existing executables could also be made checkpointable by linking to checkpoint libraries that capture running stack, heap, and program counter information in order to reconstruct the process remotely. This solution produces a checkpoint image that is rarely portable and complete because network sockets or file handles are inherently difficult to reconstruct. The system-level checkpointing provided by many virtual machine technologies allows the whole visualized environment to be checkpointed. This provides a generic solution for most cases, but its coarse nature means the checkpoint image can be very large.

24.5.3 Resource Charging

The true vision of the type of Grid that ICENI is designed to support is that of a large number of resources owned by many different organizations and available to execute code for anyone with access to the Grid. Work such as that carried out in the UK e-Science project A Market for Computational Services [263] is under way to develop frameworks to support programmatic charging and negotiation for Web services [95]. Once components of an application execute on a resource, the resource owner may wish to charge the job owner a fee for access to the resource. Given that an application may have large numbers of components executing on many different resources, a secure, programmatic means of transferring funds from the job owner to each of the resource owners is required. We are merging the ideas from the Market

for Computational Services project and ICENI II to support transparent programmatic charging for resource usage, enhancing the appeal of cross-organizational usage for resource owners.

24.6 Application Interaction

Once execution of a workflow begins, output may be produced. This output needs to be managed according to the requirements of the application owner. It may be that the executing application simply writes out results to a file and, after execution completes, the file is returned to a location decided by the application owner. With a large Grid of computational and storage resources, the location where the output is stored may be crucial to ensure that the computation runs efficiently and that results can be returned in a reasonable time. If the computation is set to run on a high-performance resource but subsequent transmission of the results takes twice as long as the computation itself due to congested networking, overall throughput may have been increased by carrying out the computation on a slower resource that would allow more efficient return of the results. More complex computations may need to be visualized during execution and possibly steered, raising other complex issues.

24.7 Conclusion

Through the presentation of a complete workflow pipeline encompassing specification, optimization, and execution, we have discussed many issues that arise in the complex process of generating and running a Grid application.

We see that e-Scientists have a requirement for large, complex workflows. Given this complexity, realizing these applications in a heterogeneous Grid environment is an inherently difficult task. To make this process transparent to the end user, we define a series of stages that separate the concerns of using this pipeline. By providing intelligence at each stage, through the manipulation of rich component information, we help reduce the effort required in subsequent pipeline stages.

One of the main aims of the workflow pipeline is to reduce application execution time. This is partly achieved by selecting appropriate component implementations and resources on which to deploy the parts of the workflow. This on its own is insufficient to ensure an optimal execution and thus other information needs to be taken into account such as the workflow, the interdependencies between the components, network performance, and reliability. This all needs to be monitored and acted upon to ensure that the QoS constraints imposed by the user are not violated and to take advantage of other opportunities that may arise. However, there is a trade-off between application execution time and workflow pipeline execution time. It

may be possible to spend more time tweaking out more optimal component placements/selections, though this may end up taking more time than is saved. This trade-off is necessary to ensure that the overall throughput is maximized.

Many of the ideas presented in this chapter have developed from our work on the ICENI architecture. In light of our experiences with ICENI and the emergence of several new use cases, we have developed the full, expanded pipeline described in this chapter. This work is now feeding into our ICENI II implementation.

Expressing Workflow in the Cactus Framework

Tom Goodale

25.1 Introduction

The *Cactus Framework* [15, 73, 167] is an open-source, modular, portable, programming environment for collaborative HPC computing. It was designed and written specifically to enable scientists and engineers to perform the large-scale simulations needed for their science. From the outset, Cactus has followed two fundamental tenets: respecting user needs and embracing new technologies. The framework and its associated components must be driven from the beginning by user requirements. This has been achieved by developing, supporting, and listening to a large user base. Among these needs are ease of use, portability, the ability to support large and geographically diverse collaborations and to handle enormous computing resources, visualization, file IO, and data management. It must also support the inclusion of legacy code, as well as a range of programming languages. It is essential that any living framework be able to incorporate new and developing cutting edge computation technologies and infrastructure, with minimal or no disruption to its user base. Cactus is now associated with many computational science research projects, particularly in visualization, data management, and Grid computing [14].

Cactus has a generic parallel computational toolkit with components providing, e.g., parallel drivers, coordinates, boundary conditions, elliptic solvers, interpolators, reduction operators, and efficient I/O in different data formats. Generic interfaces are used (e.g., an abstract elliptic solver API), making it possible to develop improved components that are immediately available to the user community. Cactus is used by numerous application communities internationally, including numerical relativity e.g. [26], climate modeling [118, 404], astrophysics [32], biological computing [211], computational fluid dynamics (CFD) [239], and chemical engineering [74]. It is a driving framework for many computing projects, particularly in Grid computing (e.g., GrADS [12], GridLab [175], GriKSL [176], ASC [32, 57]).

Also, due to its wide use and modular nature, Cactus is geared to play a central role in general dynamic infrastructures.

Although Cactus is distributed with a unigrid MPI parallel driver, codes developed in it can also already use multiple *adaptive* mesh-refinement drivers *with minimal or no changes to the code*, including Carpet [80, 380], PAGH GrACE [170]), and SAMRAI [282, 375].

25.2 Structure

As with most frameworks, the Cactus code base is structured as a central part, called the "flesh" which provides core routines and components called "thorns" .

The flesh is independent of all thorns and provides the main program, which parses the parameters and activates the appropriate thorns, passing control to thorns as required. It contains utility routines that may be used by thorns to determine information about variables and which thorns are compiled in or active, or perform non-thorn-specific tasks. By itself, the flesh does very little apart from move memory around; to do any computational task the user must compile in thorns and activate them at runtime.

A thorn is the basic working component within Cactus.[1] All user-supplied code goes into thorns, which are, by and large, independent of each other. Thorns communicate with each other via calls to the flesh API plus, more rarely, custom APIs of other thorns. The Cactus component model is based upon tightly coupled subroutines working successively on the same data, although recent changes have broadened this to allow some element of spatial workflow.

The connection from a thorn to the flesh or to other thorns is specified in configuration files that are parsed at compile time and used to generate glue code that encapsulates the external appearance of a thorn. Two thorns with identical public interfaces defined in this way are equivalent.

At runtime, the executable reads a parameter file that details which thorns are to be active, rather than merely compiled in, and specifies values for the control parameters for these thorns. Inactive thorns have no effect on the code execution. The main program flow is shown in Figure 25.1.

25.3 Basic Workflow in Cactus

In most existing workflow systems component composition is achieved by specifying the components and their connections in some workflow language or

---

[1] Thorns are organized into logical units referred to as "arrangements." Once the code is built, these have no further meaning — they are used to group thorns into collections on disk according to function, developer, or source.
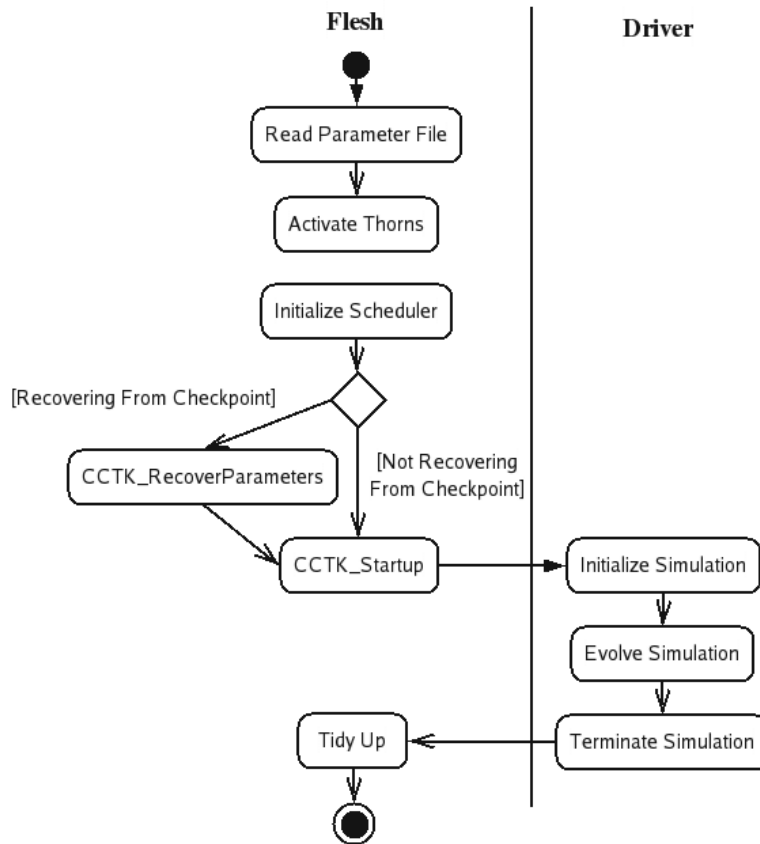
Figure 25.1:   The main flow of control in the Cactus framework. The flesh initializes the code and then hands control to the driver thorn (see Section 25.3.2). The actions in the driver swimlane are detailed in Figures 25.2, 25.3, and 25.4.

through a graphical user interface, a paradigm familiar to most users. Cactus component composition however, is a function of the flesh, guided by rules laid down by developers when they develop their thorns.

Cactus defines a set of coarse scheduling bins, as shown in Figures 25.1–25.4[1]; routines from a thorn are scheduled to run in one of these bins relative to the times when other routines from this thorn or other thorns are run. The thorn author may also schedule groups within which routines may be

_____

[1] Future versions of Cactus will allow thorns to specify additional top-level scheduling bins.

scheduled; these groups are then scheduled themselves at time bins or within schedule groups analogously to routines.
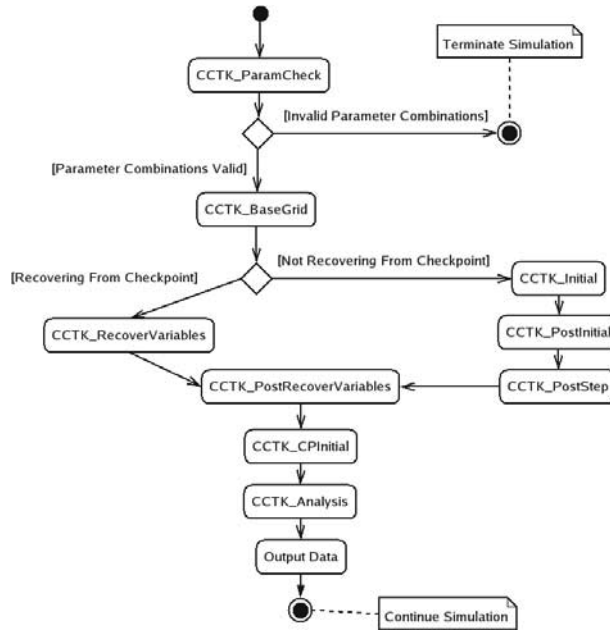
Figure 25.2: Initialization action diagram (corresponding to the *initialize simulation* action in Figure 25.1). All activities prefixed with "CCTK_" are schedule bins (see Section 25.3).

Routines (or schedule groups — for scheduling purposes they are the same) scheduled from thorns may be scheduled *before* or *after* other routines from the same or other thorns and *while* some condition is true. In order to keep the modularity, routines may be given an alias when they are scheduled, thus allowing all thorns providing the same implementation to schedule their own routine with a common name. Routines may also be scheduled with respect to routines that do not exist, thus allowing scheduling against routines from thorns or implementations that may not be active in all simulations. Additionally, the schedule.ccl file may include if statements which only register routines with the scheduler if some condition involving parameters is true.

Once all the routines have been registered with the scheduler, the before and after specifications form a directed acyclic graph, and a topological sort is carried out. Currently this is only done once, after all the thorns for this simulation have been activated and their parameters parsed.
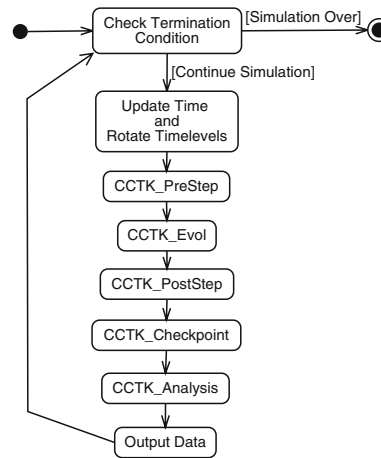
Figure 25.3: Evolution action diagram (corresponding to the *evolve simulation* action in Figure 25.1). All activities prefixed with "CCTK_" are schedule bins (see Section 25.3).
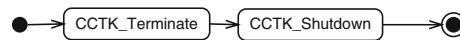
Figure 25.4: Termination action diagram (corresponding to the *terminate simulation* action in Figure 25.1). All activities prefixed with "CCTK_" are schedule bins (see Section 25.3).

This scheduling mechanism is rule-based as opposed to script-based. There are plans to allow scripting as well; see Section 25.4.3 for further discussion of this.

25.3.1 Conditional Scheduling and Looping

The scheduling of routines may currently be made conditional by two methods: at schedule creation time based on parameters having certain values, using `if` statements in the CCL (Cactus Configuration Language) file; and at schedule execution time by use of a `while` statement. More sophisticated flow control will be possible in the future.

An `if` clause in the schedule specification controls whether a routine is registered with the scheduler for running at any point. These `if`s are based upon algebraic combinations of the configuration parameters read in at program startup and thus are only evaluated once.

The `while` specification allows for a degree of dynamic control and looping within the schedule, based upon situations in the simulation, and allows looping. A routine may be scheduled to run while a particular integer grid

scalar is nonzero. On exit from the routine this variable is checked and if still true, the routine is run again. This is particularly useful for multistage time integration methods, such as the method of lines, which may schedule a schedule group in this manner.

25.3.2 Memory Management

The language (CCL) used to define the scheduling also defines the variables passed between the routines invoked by the scheduler. This allows memory and parallelization to be managed by a central component (the *driver*) and also provides support for legacy codes written in languages, such as FORTRAN 77, that lack dynamic memory support.

When scheduling routines, the developer specifies which variables require memory allocated during the course of a particular routine. Memory for variables may be allocated throughout the course of the simulation or just during the execution of a particular scheduled routine or schedule group; specifying memory just for a group has no effect if the memory was already allocated for that variable.

25.3.3 Spatial Workflow

Thorn authors may define specific functions that they provide to other thorns or expect other thorns to provide. This provides an aliasing mechanism whereby many thorns may provide a function that may be called by another thorn with a particular name, with the choice of which one is actually called being deferred until runtime.

A thorn using such a function may state that it requires the presence of the function or that this function is optional. If it is required, the flesh will produce an error and stop code execution after all thorns have been activated if none of the activated thorns provides the function; if it is optional, the thorn using it must make its own check before calling the function.

25.4 Extensions

Cactus defines a basic scheduling mechanism for tightly coupled simulations. It is, however, possible to use Cactus in more loosely coupled situations. This section describes two such applications: large-scale distributed task farming and the use of Cactus within other frameworks.

25.4.1 Task Farming

Many problems are amenable to a *task farming* approach, whereby a large number of independent tasks are started and their results collated;

e.g. Monte Carlo simulations and parameter searches. The Cactus task farming infrastructure allows many independent processes to be started across a heterogeneous set of resources; these processes need not be Cactus applications.

In order to deal efficiently with startup costs on remote resources, such as security and batch queues, the Cactus task farming infrastructure makes use of a three-level approach, in contrast with a classical master–slave two-level approach. The user starts a Task Farm Manager (TFM) on a machine that has good connectivity to the outside world, or at least to the potential resources, and this TFM then finds resources and starts slave TFMs (e.g., by submitting to a batch queue) on the resources; in principle, this can be repeated, so we number the generation of TFMs—TFM0 is the master, TFM1s are the first-generation child TFMs, etc. When a TFM1 starts, it contacts the TFM0 and requests tasks, based upon the resources it has allocated to it. For example we may wish to run 500 two-processor tasks. The TFM finds two resources — a 100 processor queue on one machine (A) and a 400 processor queue on another machine (B) — and queues TFM1s for these machines. When the TFM1 on machine A starts, it requests 50 tasks from the TFM0.

A TFM is just a running instance of Cactus containing two particular components: a core TFM thorn providing the application-independent part of the task farming, such as choosing resources and starting child TFMs or tasks themselves, and an application-specific part, referred to as a `logic manager`, which provides the application-specific information. Figure 25.5 shows the logical relationship of the thorns and shows the interface that a logic manager must expose to the TFM:
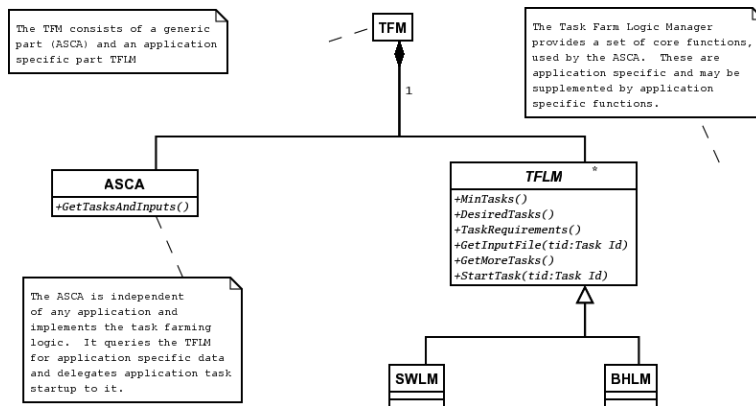
Figure 25.5: The "classes" making up a Task Farm Manager. SWLM and BHLM are application specific logic managers.

- *MinTasks.* Returns the minimum number of tasks that must be run simultaneously. For many applications, this would be 1; however for applications where tasks exchange data between them, such as a distributed Smith–Waterman algorithm, a larger number is necessary.
- *DesiredTasks.* In an ideal world, it would be possible to specify the maximum number of tasks that will be run, search for the required resources, and run them all simultaneously; in practice, however, there are not infinite resources, and, for a guided parameter search, the number may not be known in advance. This function returns the application's (or application programmers') best guess for reasonable resource requirements.
- *TaskRequirements.* This function returns the number of processors and amount of memory required by each task.
- *GetInputFiles.* Given a task ID, this function returns the command-line arguments for starting the task and a list of URLs of files that must be staged to the task's working directory before it starts.
- *GetMoreTasks.* The TFM uses this to retrieve more task IDs from the logic manager whenever tasks finish.
- *StartTask.* Some tasks require to be started up in special ways (e.g., using mpirun); this function allows the logic manager to customize the startup. This is invoked on the TFM1.

The first three functions are used by the TFM0 to determine the characteristics of the tasks before searching for resources; `GetMoreTasks` and `GetInputFiles` are used by the TFM0 whenever a TFM1 requests more tasks; and `StartTask` is used by the TFM1 to start individual tasks. See Figure 25.6 for a diagram showing the interaction sequence of the various thorns and processes.

25.4.2 Connection to Other Frameworks

The Cactus framework is designed around the needs of tightly coupled, high-performance simulations. The majority of the other frameworks included in this book deal with large-scale distributed workflows, and Cactus can easily be integrated as a component within such a workflow.

Triana

Within Cactus, users typically want to view or analyze certain files to monitor the progress of the application or derive scientific results. Cactus can output data in many formats, such as HDF 5 files, JPEGs, and ASCII formats suitable for visualising with common tools such as X-Graph or GNUPlot. A user would typically want the flexibility of being able to choose, at runtime, the files he or she wishes to view or analyze in an interactive fashion. For example, a user may notice from the JPEG images that a simulation of a system consisting
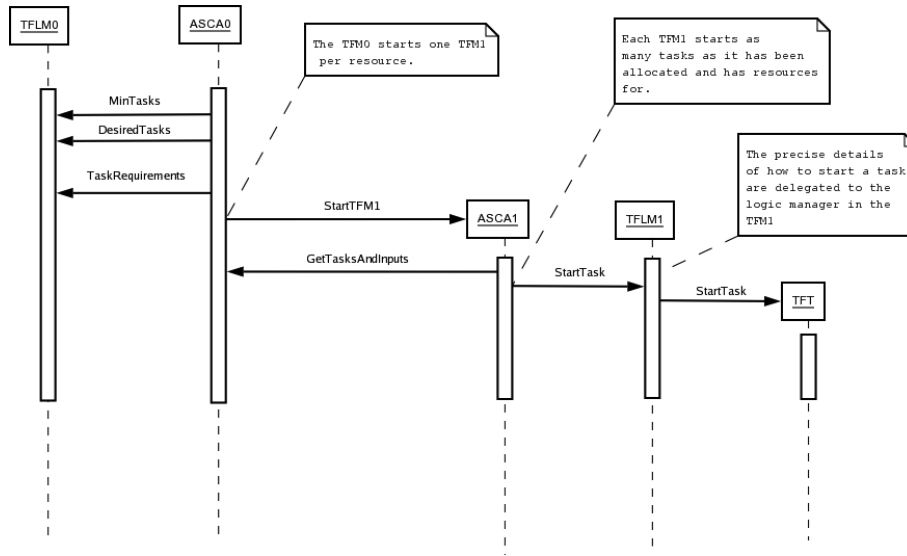
Figure 25.6:   The sequence of actions performed in the startup of an application using the Cactus task farming infrastructure.

of two orbiting sources is showing the sources coalescing; this user may then wish to verify these findings by retrieving the detailed simulation data and passing them to other analysis tools or even converting the output to an audio format and listening to the acoustic waveform directly. Our protocol therefore supports the dynamic notification necessary for such interactions. When a file is created, the Web service deployed within a Triana unit is notified, and at each time step, the Web service is contacted and can choose to receive any of the files that are available. By default, the application only sends differences in text files since the last time the Web service received part of the file, thus reducing bandwidth; binary files are transferred in their entirety. If something interesting happens, the Web service can select and receive a different set of files in the next iteration.

This is aided by the use of the Triana problem-solving environment, which allows components to be dynamically added/removed as the application runs. Within Triana, a unit was created to host the Web service representing the underlying protocol. This is shown in Figure 25.7. The unit upon initialization uses WSPeer [187] to dynamically create and deploy the Web service within the Axis environment and create the necessary WSDL file representing the methods within the protocol. The actual protocol is quite simple. It involves a notification and selection procedure but is carefully designed so that it is completely application (i.e., Cactus) driven. This ensures that we do not run into firewall issues.

In our initial development, Triana and the Cactus application are deployed and instantiated independently, which is a useful model for occasional monitoring of an application's progress, as it allows a user to make a later decision to use Triana to monitor the output. In the full usage scenario we envisage, however, a Triana unit would also be used to deploy Cactus on a remote resource on demand, thus allowing Triana to manage the full life cycle of the workflow, as is done in other workflow management systems.

In this current stage of deployment, we are using one Cactus Triana unit per Cactus instance running on the Grid. This approach is not scalable in the visual sense (e.g., imagine trying to visualise several thousand Triana units) or in the networking sense (e.g. having thousands of local instances of the same Web service would be impractical for hosting environments). To address these issues, we are currently planning on building a scalable Cactus unit that allows many instances to be mapped internally within one Web service instance. We imagine that this would build around the Triana dynamic scripting or looping implementation (to hide the visual complexity) and then such instances would be mapped using proxies to a Cactus Triana unit instance for that script or loop. This would allow the connection of possibly hundreds of instances.

If more instances are needed, then we can use the Triana distributed mechanisms [408] to segregate the workflow and run it across several Triana GAP services across the Grid, allowing potentially many thousands of instances. However, the algorithmic problem of how these results are analyzed would be application-specific. Within one scenario involving Cactus, we imagine that Triana would be monitoring the output of its results to see if something interesting had happened (e.g., the apparent horizon of a black hole simulation). Then Triana would invoke a separate workflow to farm off many independent Cactus simulations to investigate this phenomenon more closely and then analyse the results upon completion. The user would only wish to view when a certain optimization level has been reached.

A prototype of this protocol has been demonstrated in SC2004 and SC2005, where we showed the visualisation of a 3D scalar field produced by two orbiting sources. This was accomplished by using this protocol to connect a Cactus simulation, running on an HPC resource, and Triana, running on a users workstation. Triana received notifications of the files created by Cactus and then selected the ones it wished to visualize. The result was that the user could see real-time JPEG images from the remote application, representing the three dimensions of the scalar field, as the simulation progressed.

25.4.3 Future Directions

Another enhancement would be to allow scripting as an alternative to the current scheduling mechanism. The current mechanism allows thorns to interoperate and for simulations to be performed with the logic of when things happen encapsulated in the schedule CCL file; other frameworks do the same thing by providing a scripting interface, which gives more complete control
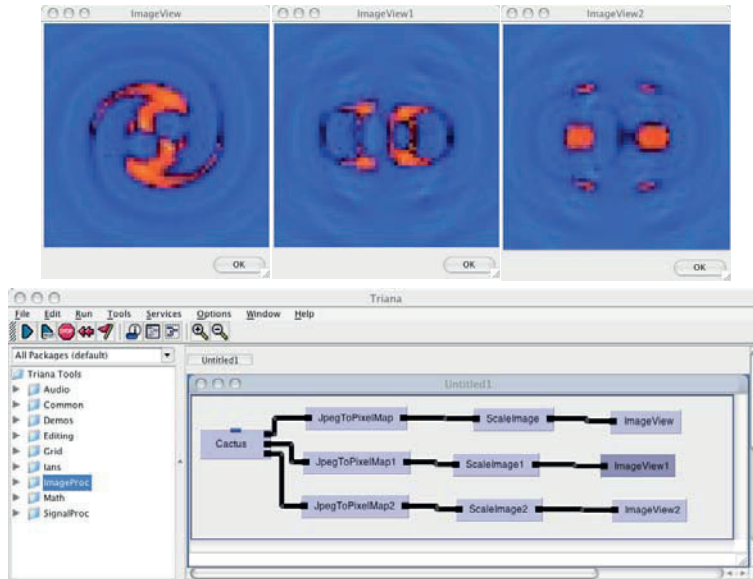
Figure 25.7: The resulting visualization from a Cactus simulation of the evolutions of a 3D scalar field.

of the flow of execution, at the expense of the user needing to know more of the internals. Both schemes have advantages and disadvantages. In the future we would like to allow users to script operations using Perl, Python, or other scripting languages.

Automated Composition of Workflows

While in Cactus composition of workflow currently consists of activating the requisite thorns, as the size and complexity of workflows increase it may becomes difficult or impossible for a human to create the workflow explicitly. Future versions of Cactus will address this problem by providing components with semantic information that can be used to automatically compose workflows and allow automated recomposition to be triggered on demand. We plan to provide the ability to take a set of software components and a task specification and determine the appropriate composition and configuration of these components.

Distributed Component Level Debugging

Debugging large distributed applications is hard. On single systems or clusters, tools such as TotalView [136] are very useful; however, these do not scale well

to the wide-area heterogeneous component-based simulations currently being developed. There are currently plans to develop the component interfaces in Cactus to allow single stepping at the component level and tracing data flow into and out of components, and to add features to allow debugging through familiar mechanisms, such as breakpoints, trace variables, stepping through workflows, or even dynamically reconfiguring workflows.

Sedna: A BPEL-Based Environment for Visual Scientific Workflow Modeling

Bruno Wassermann, Wolfgang Emmerich, Ben Butchart, Nick Cameron, Liang Chen, and Jignesh Patel

26.1 Introduction

Scientific Grid computing environments are increasingly adopting the Open Grid Services Architecture (OGSA), which is a service-oriented architecture for Grids. With the proliferation of OGSA, Grids effectively consist of a collection of Grid services, Web services with certain extensions providing additional support for state and life cycle management. Hence, the need arises for some means of composing these basic services into larger workflows in order to, for example, express a scientific experiment.

There are several approaches for composing Web services into larger workflows, most of which, at least for the composition of scientific workflows, are based on custom-made workflow languages and corresponding enactment environments. Another approach, which we have taken in our work, is to use an industry standard for the orchestration of Web services, such as the Business Process Execution Language (BPEL) [24].

BPEL, which has been introduced by IBM and Microsoft, makes a number of benefits available to scientific Grid computing. The host of commercial providers supporting BPEL means that there are industrial-strength enactment environments and middleware technologies available that exhibit a level of scalability and reliability that a research prototype could not match. The multitude of providers supporting BPEL creates a market, which means that it is a live standard with ongoing efforts to develop new features. Furthermore, BPEL could serve as a standard representation for scientific workflows and hence aid reproducibility. Finally, as a programming language that focuses on high-level state transitions, it could enable computational scientists to compose scientific workflows themselves, relieving them of a dependence on software engineers.

In our work, we have been investigating the applicability of BPEL for the expression of scientific workflows. We have established in a companion paper that freely available BPEL enactment environments satisfy the scalability and

reliability requirements of scientific workflows and that the language itself is sufficiently expressive [132].

There are still a number of questions that need to be answered and certain obstacles that need to be overcome, before being able to make the benefits of BPEL available to computational scientists. First, as BPEL is primarily targeted at business workflows, in which respects are its abstractions lacking expressiveness for scientific workflows, and how can such shortcomings be overcome? Second, considering that our target group cannot be expected to have expert knowledge of distributed systems and software engineering and given that BPEL relies on a number of XML-based standards such as Web Services Description Language (WSDL), Simple Object Access Protocol (SOAP), and XPath, how can we simplify the modeling of workflows?

In order to address these issues, we have developed a visual language and a visual modeling environment. The language provides language abstractions in addition to those found in BPEL in order to simplify the modeling of scientific workflows. We identified the need for these abstractions and subsequently their value during a case study with computational chemistry experiments. Our visual language is made accessible via a visual modeling environment, that, through a number of usability features, hides the complexity of the underlying orchestration language and middleware. The modeling environment furthermore ensures the compliance of workflows to the BPEL specification to enable scientists to execute their experiments on a host of available enactment engines. The combination of additional language abstractions and adequate tool support is what enables us to fully return ownership of workflows to scientists while retaining the benefits BPEL has to offer.

This chapter presents our work on solving these questions. In Section 26.2, we are going to introduce our visual representation of BPEL and a number of additional language abstractions aimed at simplifying the modeling of scientific workflows. Section 26.3 then explains the need for tool support, the requirements such tools need to be able to satisfy in order to hide the complexity inherent in BPEL and the distribution middleware involved, and finally discusses how our modeling environment attempts to provide the features necessary to support computational scientists. In Section 26.4, we then discuss a case study we have carried out in order to demonstrate the use of our environment for modeling and executing a nontrivial scientific workflow. Section 26.5 situates our work within the range of existing tools and approaches. Finally, in Section 26.6, we reflect on our experience with BPEL so far and present a list of future work to improve the usefulness of our modeling environment.

26.2 Modeling Scientific Workflows

There are a number of obstacles to the use of BPEL by computational scientists. The first issue is that BPEL's XML syntax is rather verbose. Furthermore, the parameters that need to be configured for BPEL activities are not always trivial in their semantics. Providing a visual language for BPEL is an obvious choice to improve the productivity of BPEL programmers and has been taken up by a number of commercial products. However, simply providing a one-to-one mapping between BPEL and a visual representation may still cause non-expert users to be overwhelmed. The second issue arises from the fact that BPEL was originally defined for business workflows. These generally are less complex than scientific workflows. Business workflows also do not, in general, exhibit the need for concurrent execution of a large number of processes.

Figure 26.1 presents an overview of our solution to these issues. The lowest level is standard BPEL, which we represent in a visual language that also provides a number of usability features. These features are, in general, split among the visual language itself and the tool support providing access to the language. The level above represents the Scientific Process Execution Language (PEL), which adds general-purpose language abstractions to increase BPEL's expressiveness for scientific workflows. The Domain PEL layer allows domain-specific abstractions, which can be added by users of our environment in order to extend the available vocabulary with abstractions closer to their respective domains. Workflows are then constructed that make use of abstractions from any of these layers.
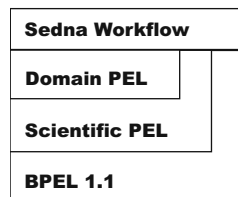
Figure 26.1: Extension of standard BPEL through additional general-purpose abstractions (Scientific PEL layer) and by allowing domain-specific extensions (Domain PEL layer).

26.2.1 Scientific Versus Business Workflows

It is important to be aware of and give due consideration to the differences between business and scientific workflows, as the application domain area influences the abstractions available in BPEL.

The most notable difference is probably one of scale. When compared with scientific workflows, business workflows usually define a relatively small number of BPEL partners with whom they interact. Scientific workflows may involve thousands of service instances that will need to be modeled as partners. Furthermore, scientific workflows will often execute thousands of basic service invocations and, consequently, send tens of thousands of SOAP messages to be exchanged among service partners. Business workflows, in the majority of instances, operate on a smaller scale.

Another difference relates to the modeling requirements of parallel execution in scientific workflows. Scientific workflows apply complex computational models that generate large amounts of data and then analyze these data. Therefore, such workflows contain large numbers of independent sub-workflows that may be executed concurrently; for example, to run models concurrently and to filter and extract data resulting from an experiment. Business workflows do not usually display massively parallel execution of very similar subworkflows on such a scale.

A related consideration is caused by the fact that e-Science [311] applications generate massive amounts of data and then need to analyze these data in successive steps. Consequently, powerful and flexible data-manipulation primitives are of utmost importance. Again, the amounts of data that need to be handled in business workflows will, in general, be smaller.

A noteworthy difference to consider is the nature of what finds expression in a workflow. A scientific workflow represents an experiment that is likely to be run only a limited number of times before new ideas and insights will need to be incorporated. Frequent changes and redeployment need to be supported and made simple. A business workflow captures a set of activities and their relationships in order to describe a business process. The overall aim is to be able to automate this process and execute it repeatedly over possibly long periods of time.

There is a gap between what BPEL aims to provide and what is required by scientific workflows. The next two sections demonstrate how we close this gap through a suitable visual language and additional language abstractions.

26.2.2 Visual Representation of BPEL

Our visual modeling environment provides meaningful interaction with the visual representation while guiding the user with numerous usability features. However, there are a few issues worth considering that are independent from the integration of modeling language and tool support.

The main issue by which our efforts have been driven is how to take account of the large scale of scientific workflows. A useful visual representation of BPEL for the purpose of expressing large-scale scientific experiments needs to provide abstractions that can help make this complexity manageable. Furthermore, we do not want to define a visual language that would require both BPEL novices and BPEL experts to learn the notation. Users with

existing knowledge of BPEL should be able to benefit from this knowledge and only have to learn how to use the additional features of a visual representation. This means that we could not base our representation on a language such as, for example, UML class diagrams, as this would have neither given us the means to address issues of scale and complexity adequately nor would it have preserved previously acquired knowledge of BPEL by a user. Figure 26.2 shows an extract of an example workflow using our notation. Our visual representation is split into three parts: the basic BPEL activities, the complex BPEL activities, and a representation of our additional language abstractions, which will be discussed in the next section.
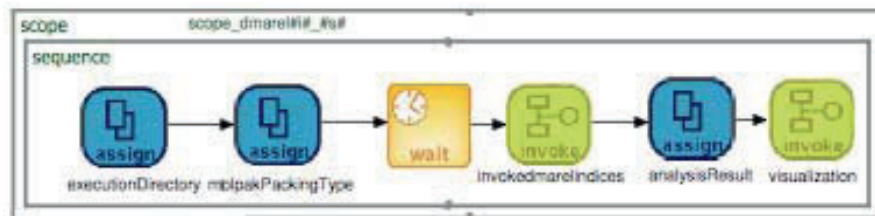
Figure 26.2: Extract of visual representation showing basic and complex activities.

All elements of our visual language have three concepts in common. They all have some form of graphical representation, a name that can be assigned to an element to identify it in a large workflow, and a list of properties that can be modified in order to configure an activity. Furthermore, all language elements have connector points that allow them to be connected to other elements. This allows users to express control flow.

The basic BPEL activities such as, for example, the assign activity (copying data between two variables), the receive activity (receipt of a message), or the invoke activity (invocation of an operation from a service partner), are represented by simple icons. The configuration of their basic properties is deeply rooted within our tool support and will therefore be discussed in Section 26.3.

Complex activities are comprised of several other complex or basic activities. Three examples of complex activities are the while construct, the top-level process construct, and the scope construct. In a one-to-one mapping from BPEL, we would have to represent these constructs by start and end tags as illustrated with two examples in Figure 26.3. Instead, we represent these constructs by what we call containers. Containers have a visible border, which restricts their scope and allows other activities to be inserted into that scope. In this way, containers help to clear up the process diagram and compress the visual representation. The abstraction of containers is also used by several commercial editors, as we will discuss in Section 26.5. However, we are able

to derive a further benefit than is the case in the existing representations. Figure 26.2 displays the scope container, which is similar to a programming block and allows for the definition of local variables. We reuse the graphical representation of the scope construct to allow users to set up, inspect, and modify these variables in the scope container itself. In this way, all elements of relevance to a particular scope are displayed graphically in a scope container and are immediately apparent when inspecting the graphical representation of a workflow.
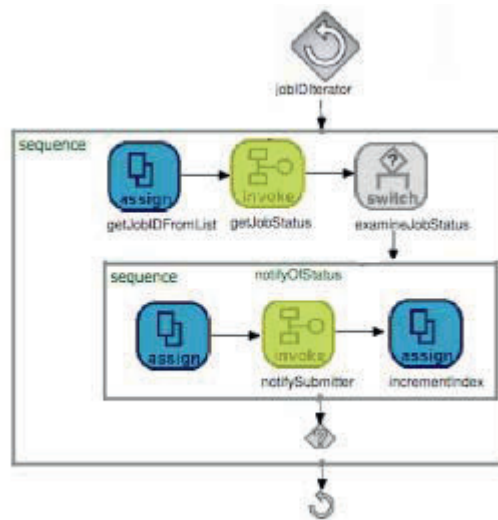
Figure 26.3: While and switch activities represented by start and end tags instead of containers.

These few elements allow us to provide a clearer representation and compress large workflows to make the complexity of scientific workflows more manageable.

26.2.3 Extensions to BPEL

The elements of our visual representation form a first step toward making BPEL more usable for the expression of scientific workflows. However, the vocabulary offered by the visual language still largely corresponds to that provided by BPEL. While we need to maintain compliance with the BPEL specification and manage to do so through our tool support, we have also established that, due to its original focus on business workflows, BPEL lacks adequate abstractions for the design and manipulation of scientific workflows. In this section, we present the first set of additional language constructs we have developed in order to address these issues.

In the scientific process execution language layer (Figure 26.1), we have general-purpose constructs making up the primitives that are demanded by the requirements of scientific workflow modeling, namely scale and concurrent execution of complex sets of activities. To further illustrate the purpose of this layer, we are going to discuss two of its primitives in more detail: the indexed flow construct and the concept of hierarchical composition of workflows. The domain process execution language layer allows for domain-specific extensions that can encapsulate complex sets of activities required in certain domains into a single reusable activity. In this section, we present the concepts of plug-ins and macros. The new language constructs presented below have been developed according to insights gained in a case study (Section 26.4).

Indexed flows. As mentioned before, scientific workflows frequently require the modeling of concurrent execution of sets of activities to apply complex computational models and then analyze the resulting data. BPEL supports concurrent execution with flows. BPEL's flow construct allows the definition of multiple sequences of activities, each of which will be executed in parallel. For scientific workflows, where we often have very similar sequences of activities that can be executed in parallel, this requires the repeated specification of the same information. Clearly, having to model the same set of parallel activities 200 times is tedious and furthermore leads to an explosion of any representation of the workflow, whether textual or graphical. The indexed flow construct is better suited to modeling concurrent execution of sets of activities than BPEL's native flow construct, as it does not require the repetition of similar information over and over again. The indexed flow is a container into which other activities can be placed for execution. It allows a user to specify an index that determines the required number of parallel executions. The index has a start and an end range, and the contained activities will be executed $(endrange - startrange + 1)$ times. An index has a name, which allows us to use its numerical value in queries and conditions to identify a particular flow and manipulate its behavior. The modeling environment we have developed translates an indexed flow into a number of standard BPEL flow constructs, effectively relieving users of the tedious repetition of the same information while maintaining a simple graphical representation. The next version of the BPEL specification is going to introduce a similar construct, called parallel forEach.

Hierarchical composition. Mechanisms are needed to manage the sheer size and complexity of scientific workflows. A complete workflow, as we will present in Section 26.4, can involve a great deal of basic activities and invocations of partner services. Designing such a workflow in a top-down manner can be extremely difficult, and modeling it is likely to be error-prone. It can often be possible, however, to identify common subworkflows in such large workflows. We therefore need a mechanism that enables us to split large workflows into several subworkflows. The hierarchical composition of workflows is not so much an abstraction we have developed but rather a concept we have found to be quite useful. It exploits the fact that each BPEL process is itself

described by a WSDL interface, which enables other workflows to invoke a workflow like any other service. A workflow's initial message-receive activity provides for the input elements of the interface, and the eventual reply activity provides the output. This provides us with a means of conquering some of the complexity introduced by very large workflows, as it enables us to design workflows in a bottom-up manner. We can identify individual subworkflows, which may be reused by other workflows, and can start to model and test them independently. The benefits of hierarchical composition are clearly the reuse of existing workflows and a reduction of the complexity of larger workflows. Consider Figure 26.4. In this example, the main workflow has been broken up into two subworkflows. The job submission subworkflow is responsible for the submission of jobs to a Grid scheduler and for returning the results of these jobs to its caller as they become available. The visualizer subworkflow interacts with various services in order to achieve visualization of data in certain formats (e.g., tabular representation, scatter plot representation). These subworkflows would be reusable among many other workflows. In this case, the main workflow would coordinate among its two subworkflows. It would gather the input data for some computation and submit this to the job submission workflow, which prepares it for submission to the Grid and actually submits the jobs. As soon as the results of a computation are returned from the Grid, the job submission workflow will respond to its caller with the resulting data. Then, the main workflow can in turn invoke the visualizer workflow to amend the current visualization with the results as they become available.
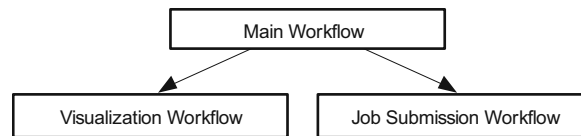
Figure 26.4: Decomposition of a large workflow into several subworkflows.

Plug-ins. It may not always make sense to break up a complex workflow into subworkflows. There may be cases in which we have an extract of a workflow and not necessarily a complete one with an initial receive and eventual reply activity to accept input and return a response, respectively. Such sets of activities may be interacting with several services commonly used in a particular scientific domain and use complex XPath queries in order to, for example, carry out data conversion from and/or to domain- or service-specific data formats. In such cases, we want to reuse these activities as a single atomic unit of operation, and hierarchical composition, which always involves complete workflows, would not be the best option; even though a workflow or part of it is of considerable complexity, we want to keep it as one activity. For example, several sequences of activities for data conversion into a domain-

specific format and invocation of services used in computational chemistry can, conceptually, be considered as a single domain-specific activity, and it would therefore be beneficial if we could inline these sequences of activities into our workflows as one activity. To address these issues, we have developed plug-ins. Plug-ins encapsulate a domain-specific parameterizable sequence of BPEL activities that once defined can be used as a basic BPEL activity. The semantics of plug-ins are defined by providing an operational description in the form of a simple Java class that generates the BPEL code, as well as, an XML descriptor containing information about the plug-in's graphical representation and configurable parameters. Using plug-ins, an otherwise complex workflow can have its representation substantially compressed, and the complexity of the encapsulated BPEL activities is effectively hidden from workflow designers.

Macros. Hierarchical composition, however useful it is, incurs a certain amount of overhead in terms of communication and thread use. The reason for this is simply that workflows that have been composed in a hierarchical manner run in separate threads and communicate by passing SOAP messages. Using plug-ins in order to specify domain-specific activities, which can be further configured, is a powerful alternative to hierarchical communication, which does not incur the same overhead. However, in order to cater to the configurable properties of a plug-in, we need to write custom Java code that knows how to use these properties in the exported BPEL. In order to avoid the overhead of hierarchical composition and in case no further configuration of a reusable activity is required, we also introduce macros. To define a macro, a user models a set of activities in the editor and then, via a menu option, turns these activities into a reusable macro that is available for use in workflows like any other activity. Macros can be added to workflows and are like inlined BPEL activities that get expanded during the editor's export of a workflow into standard BPEL. Users can build up toolboxes of useful macros and make them available to their colleagues.

26.3 Scientific Workflow Editor

These additional higher-level abstractions we have introduced are an essential prerequisite for introducing BPEL into the domain of scientific computing. However, BPEL will not be taken up by application scientists unless we can also provide adequate tools to support them in their work and hide the complexity of the underlying technologies. In this section, we will discuss how we can return ownership of workflows to scientists through the automation and usability features of an adequate visual modeling environment.

26.3.1 The Need for Tool Support

Let us briefly characterize our target group. Computational scientists can, in general, be regarded as highly computer literate, as several branches of science

have employed scientific computing technologies for decades. We can expect to find some programming skills. However, we should by no means assume large-scale software development experience or expert knowledge of distributed systems and middleware. Furthermore, we need to remind ourselves that scientific computing aims to be an enabling technology. To a computational scientist, acquiring skills usually associated with software engineering is a distraction from what is relevant.

It will often be necessary to change a workflow incorporating new insights or ideas gained from previous results, and it is therefore important that ownership of the workflow remain with the scientist. They should be able to carry out any modifications whenever this is needed, as well as deploy and execute these workflows. This ensures that their knowledge can be materialized directly, without requiring communication with and translation of ideas into a computational form by software engineers. In order to achieve this goal of truly returning ownership, we need to hide complexity at several levels apart from developing more suitable language abstractions. In particular, we need to relieve scientists from a detailed understanding of BPEL and the distribution middleware used.

BPEL relies on a complex set of underlying technologies, which include XML, XML schema definitions (XSD), XPath queries, WSDL, and SOAP. In order to master BPEL, it is necessary to understand how all these technologies relate to each other. Due to the effort required in learning BPEL and its associated technologies, we should provide a development environment that abstracts away from the details and automates the generation of valid BPEL as much as possible. Furthermore, given the large number of Web services and XML schema definitions with which scientific workflows need to interact, it is necessary to at least provide for a means of inspecting these WSDL interfaces and schema definitions from within the modeling environment.

Two further sources of complexity arise from the distribution middleware used, which involves a variety of middleware such as Grid job schedulers, BPEL enactment engines, Web service containers, and so on. Scientists need not be concerned about the details of the underlying distribution middleware, such as, for example, what kind of scheduling mechanism is used to schedule jobs arising from workflows on the Grid. Support is also required for the deployment of workflows on BPEL engines. Scientists need to be able to deploy workflows as well as modifications to them without being concerned about the mechanisms and peculiarities (e.g., different formats of deployment descriptors) of individual BPEL engines provided by different vendors. Therefore, an adequate modeling environment needs to provide sufficient integration with various BPEL engines to be able to fully automate the process of deploying workflows.

Finally, there is a requirement for validation, monitoring, and debugging of scientific workflows due to their considerable level of scale and long-running nature. BPEL is statically typed. This supports the detection of a number of errors during modeling. A workflow editor should thus support extensive

predeployment validation of workflows in order to enable users to correct any detected problems before they are deployed and executed. This is important for two reasons. First, if we were to allow the generation of invalid BPEL, then this would interfere with our ability to automatically deploy workflows on BPEL engines and would require user interaction at a potentially detailed level. Second, some errors may only materialize themselves in a running workflow after a considerable amount of time, which could become quite expensive in terms of lost computation. Tools should provide some effective means of debugging workflows, potentially in a manner similar to that offered by Java debuggers, given that our users are non-experts and that the size of scientific workflows may increase the chance of introducing errors. The current monitoring capabilities of our environment are discussed in [132].

26.3.2 Sedna

Our visual modeling environment is called Sedna, in keeping with the tradition of the Eclipse platform.[1] We have developed it using the Eclipse IDE plug-in mechanism [155]. Sedna presents scientists with a graphical process modeling environment and provides a number of features whose aim is to further abstract away from BPEL and simplify the development of workflows.
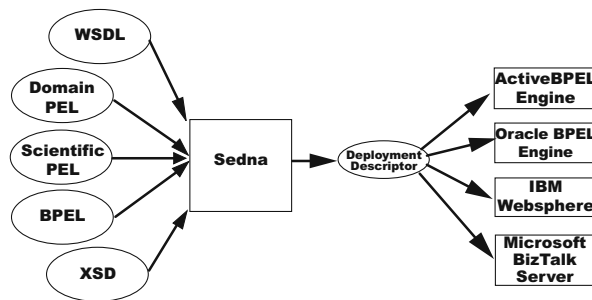
Figure 26.5: Sedna translates and exports the various language elements into standard BPEL and creates deployment descriptors for various BPEL workflow engines.

Figure 26.5 presents a high-level overview of the editor. It provides access to the visual language representation of BPEL as well as the additional language abstractions we have discussed. Furthermore, it gathers information about the services with which a workflow interacts in the form of WSDL

---

[1] Sedna is the most recently discovered trans-Neptunian planetoid of our solar system. Astronomers do not consider it a planet, although it has a perfect shape. We adopted its name because it is a humble object. Like our modeling environment, it is small and lightweight.

interface definitions and the data types used as XML schema definitions. During workflow modeling, it provides numerous usability features and carries out several tasks and settings automatically in the background. The editor works with users on validating the workflow, and once this is complete, it deploys the workflow in a format that can be executed on any compliant BPEL engine. This is achieved by translating all the various language elements into standard BPEL and generating deployment descriptors for a number of engines.

The Structure of Sedna

Sedna has been developed as a plug-in in the Eclipse IDE. Eclipse is a popular and highly extensible open-source IDE that integrates many features of a development environment, such as various source code editors, access to CVS repositories, and a number of task-specific views, and provides many native user interface components as part of its Standard Widget Toolkit [127].

New components can be added by providing them as plug-ins. This modularity and openness has the advantage that any new plug-in can extend and make use of all the other features Eclipse provides. This, for example, enables our editor to provide the built-in facilities for handling projects and provide access to a graphical CVS client. Furthermore, as our editor is an Eclipse plug-in, it can be further extended by third-party plug-in developers to add support for additional features (e.g., deployment on a new BPEL engine or support for collaborative workflow modeling).

Plug-in development in Eclipse incurs a considerable learning curve for Java developers, but overall, given the vast array of existing plug-ins that can be reused to a large extent, it simplifies the provision of development tools. In particular, it aids in creating a consistent user experience through a familiar graphical user interface (i.e., the icons and other widgets used by all Eclipse plug-ins) and interaction mechanisms, such as, for example, a unified mechanism by which new projects and resources of many different types are created. Moreover, a number of features that would otherwise be hard to implement (such as printing) are provided by the Eclipse Graphical Editing Framework. Our editor benefits from a number of other plug-ins, such as IBM's WSDL4J [206] for handling WSDL files and the Graphical Editing Framework (GEF) [126] for implementing the graphical parts of the editor. Our editor also reuses the plug-ins provided as part of the Eclipse Web Tools Platform project [462], which provides, among other features, graphical editors for XML schema and WSDL definitions. These editors are perceived as an integrated feature of our modeling environment.

The editor consists of two parts: the overview page and the process map. The overview page allows users to set up any partners, global variables, and name spaces required for the workflow definition. Partners define services with which the workflow will need to interact, and variables are temporary data containers whose types are either defined in a partner's WSDL definition

or some XML schema. The process map (see Figure 26.6) is the visual programming part of the editor, where activities comprising a workflow are actually composed and configured. Users are free to start on either of the two pages. They can also interleave the setup of partners and variables with modeling the actual workflow.
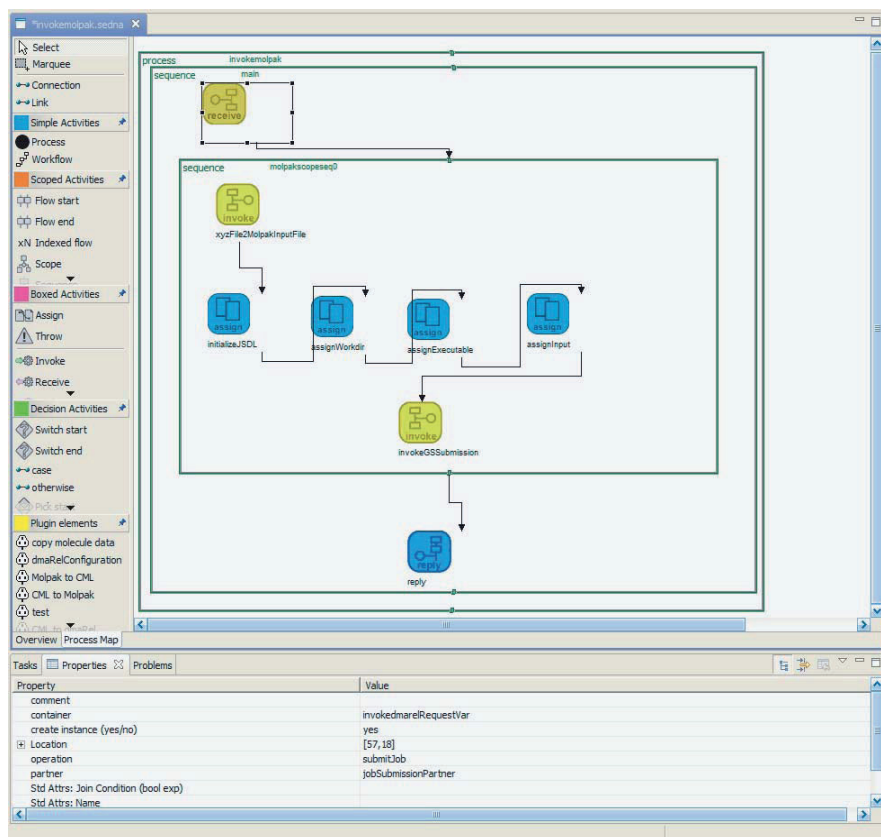
Figure 26.6: The process map displaying a workflow.

The process modeling page, or process map, is where the actual modeling of workflows takes place. The process map contains a palette of activities, the canvas displaying the workflow diagram, and a properties view for configuration of activities. The palette groups activities into several categories, such as "Scoped Activities" and "Decision Activities". It contains all standard BPEL activities, as well as our additional language abstractions from the scientific and domain PEL layers. Activities are represented as icons with descriptive text and can simply be dragged from the palette and dropped

onto the canvas in order to be used in a workflow. The canvas expands these icons into large images.

The file format used internally by our editor to represent workflows is called sedna. Sedna files store the nonstandard BPEL language constructs we have discussed along with some metainformation for storing information about the locations and sizes of the graphical elements of a workflow, as well as additional information about required partners, variables, and name spaces. Existing workflows expressed as BPEL files can be read and can be represented graphically by the editor, although an automatic translation of the standard features of BPEL into our additional abstractions is not currently supported.

Usability and Automation

There are a number of notable usability features that we want to discuss here.

A good example for the reuse of Eclipse features and resulting usability are the project management features of Eclipse, which we use for setting up new workflow projects. The editor is integrated with the "New Project" creation wizard and menu options in Eclipse. That allows users to create a new workflow project in the same way they would create any other kind of project resource in Eclipse. To create a new workflow, a user selects the type of resource to be created (i.e., BPEL workflow) from a list of options. According to this selection, an appropriate wizard appears, which, in our case, allows setting of the filename, storage location, and target name space of the workflow. Upon completion, the wizard will create the set of required files and open an instance of our editor.

The management of projects and files can be carried out using the package explorer, which looks similar to a file browser. The standard package explorer allows the manipulation and organization of all related file resources (e.g., WSDL files, XSD files, input files) and the management of multiple workflow projects.

The use of wizards is an important usability feature in our editor. The wizard mechanism of Eclipse suggests that we detect what the user is doing and validate her actions, so that we can provide instant feedback and context-sensitive guidance at each step. The ability to take corrective action from within a wizard is of particular benefit for the nonskilled user. On the overview page, the process of setting up partners, variables, and name spaces is facilitated by wizards. For example, the partner setup wizard asks the user to specify the URL or file location of the partner's WSDL definition. The wizard can then parse this WSDL in order to validate it and provide feedback to the user about any problems that may be present. The wizard furthermore detects the absence of partner link type definitions in the WSDL and can offer to automatically generate appropriate definitions in the specified WSDL. Another example is a wizard guiding users through the process of setting up a BPEL assign activity, which relies on complex XPath queries to derive its versatility. By offering a wizard, we relieve users from having to learn XPath.

Another source of complexity that the editor hides is the configuration of activities with information such as service partners, operations, variables, etc. The editor displays what is called a properties view, which can be seen at the bottom of Figure 26.6. The properties view is a tabular representation of the named properties of an activity selected in the workflow. The view allows users to enter or select appropriate values. For example, selecting an invoke activity in the process map will prompt the properties view to display fields for configuring the operation that is to be invoked as well as its input and/or output variables. In most instances, the view will display the available options for configuring an activity from drop-down lists (e.g., lists of operations, variables, partner links, etc.), and the editor restricts the list of valid options by deriving some properties of certain activities and carrying out the corresponding settings automatically. For example, the invoke activity in BPEL usually requires the specification of a partner's portType to determine the operations that might be invoked. Instead, our editor uses information about the use of the activity in a particular instance to determine the portType automatically and then offers a suitable set of operations from which to choose once the user has selected the appropriate service partner from a drop-down list. The user only needs to be concerned about choosing the desired service and the operation provided by this service.

The usability benefits offered by the editor would break down if users were required to program the WSDL interfaces of their BPEL workflows themselves. Therefore, the editor generates WSDL interfaces of workflows on the fly. It does this by detecting relevant additions and deletions of activities that have an impact on the WSDL interface of the current workflow and any workflows with which it interacts, as is the case for asynchronous interactions between workflows. In case WSDL generation is enabled, the editor will automatically generate a WSDL interface as the user progresses with modeling the workflow. An example where this feature is particularly useful is when one workflow is a client of an asynchronous workflow and another one acts as the provider. Writing the correct WSDL interfaces for such workflows requires a solid understanding of BPEL as well as Web services.

Given the large number of service partners defined via WSDL interfaces and the different data types defined via XML schema definitions, it is necessary to be able to inspect these definitions to determine how they should be used. The Eclipse plug-ins of the Web Tools Platform project complement the features of our editor with graphical editors for inspecting and editing WSDL as well as XML schema definitions. The WTP editors simplify choosing or even generating data types for an automatically generated WSDL message. The editor seamlessly integrates inspection and modification of WSDL and XSD used in a workflow. It enables users to see the relationship between a particular XML data type, the WSDL message using these data, and the corresponding operations of a service using these messages as input and output.

Another simple but nevertheless important feature of the editor is the ability to label activities and equip them with additional comments. Due to the potentially large size of scientific workflows, the ability to label basic activities as well as containers helps to communicate the workings of a workflow. Comments can be added for further clarity.

Validation

As explained before, a crucial usability feature is the BPEL type system and syntax rules to provide predeployment validation of workflows. The aim of predeployment validation is to ensure, as far as possible, that deployment will be successful and that there are no preventable errors in the workflow. Our ultimate aim is to catch any problem possible before we even deploy the workflow and give the user and editor a chance to resolve identified issues. The editor validates the current workflow whenever it is saved, displays any issues in its problems view, and changes the graphical representation of a problematic activity in the process map. Validation can, for example, detect incompatibilities between the source and target types used in an assign, an incorrect number of activities in a scope, unconnected activities, or incompatible variables assigned as input and/or output to a particular operation. We reuse the problems view to inform users of any problems, which is the same view used by the Java compiler in the Eclipse development tools to communicate compiler warnings and errors. The benefit of reusing this view is that we can provide feedback about any problems. The user can then try to resolve these and gains immediate feedback about the success (or lack thereof) of her efforts.

Deployment

An important aspect of hiding the complexity of the distribution middleware involved is the automation of the deployment process of a workflow onto a BPEL engine. Once any issues identified during validation have been resolved, the workflow can be deployed onto a workflow engine. Sedna currently integrates with the ActiveBPEL engine, as it is an open-source BPEL engine, and our recent investigations have confirmed that it satisifies the scalability and reliability requirements of scientific workflows. Support for further BPEL engines can be added by third-party developers extending Sedna via published interfaces. Details about our examination of the ActiveBPEL engine and the workflow execution aspects of our environment can be found in [132].

Deployment of a workflow consists of two parts: generation of deployment-related files required by a particular engine and transfer of these files to the engine. At this stage, our environment automates the first part. At the click of a button, the editor exports the workflow, with all its nonstandard BPEL features, into standard BPEL. It then generates a deployment descriptor for the selected engine, which contains information about the service interfaces

of the workflow and its partners. Finally, it packages all files required for deployment in an archive ready for deployment onto the engine. For example, the ActiveBPEL engine accepts so-called bpr archives as deployment units. A bpr archive contains the BPEL representation of a workflow, a deployment descriptor used by the engine to keep track of all required resources, and any nonremote WSDL files. The actual transfer of these files onto the engine has not yet been automated, due to insufficient knowledge about users' deployment models. Open questions remain as to whether users tend to develop workflows on the same machine hosting the BPEL engine or whether transfer mechanisms must take account of a potential need for authentication in order to copy files to an engine.

26.4 Case Study: Polymorph Search

In this section, we further illustrate use of our environment and new language abstractions and show how the various middleware technologies we integrated come together. For this purpose, we present a real-world example from computational chemistry. We have used the same case study in [132] to evaluate the suitability of BPEL engines for the enactment of scientific workflows.

The application deals with the computational prediction of organic crystal structures or polymorphs. Each of the organic crystal structures an organic molecule can take has different physical properties. A method for computationally predicting likely polymorphs along with their physical properties would be of considerable benefit for the development of molecular materials [364] and in the pharmaceutical industry. For several years, the computational prediction of polymorphs has been carried out with the help of FORTRAN programs. MOLPAK [195] and DMAREL [471] are two such programs. The computational prediction of polymorphs is an exhaustive search in which MOLPAK can be used to generate possible molecule packings followed by DMAREL to optimize the lattice energy and cell volume to determine how thermodynamically feasible the resulting hypothetical crystal structures are. The calculations of the physical properties for each of those packings with DMAREL are completely independent of each other, which enables this problem to be solved using CPUs in a computational Grid without shared memory and with low-bandwidth connections.

Figure 26.7 shows an abstract overview of a polymorph search workflow. Scientists need to set up the search and prepare the molecule description. They then need to choose which packing types they might wish to explore. Each of the 38 possible packing types can be analyzed in parallel. Scientists then determine the degree of precision with which the exploration of each packing type occurs, and this determines how many different subsequent DMAREL executions are required for the packing type. For the highest precision, this may result in 200 concurrent executions of DMAREL per packing type. The

rectangles in Figure 26.7 represent Grid services, and arrows show control flow. Black bars show spawning and joining of concurrent subprocesses. Submission of MOLPAK and DMAREL computation jobs relies on the GridSAM job submission service that is available form the OMII. GridSAM implements the Job Submission Description Language (JSDL) defined by the GGF [252]. The figure does not show any data flow, which is mainly in a peer-to-peer manner by auxiliary staging Grid services.

It is worthwhile to consider the scalability requirements of this workflow. The workflow might involve up to $(38 \times 200) = 7600$ concurrent invocations of MOLPAK and DMAREL. MOLPAK and DMAREL jobs may take any time between two minutes and several hours to complete. We have used the UCL Condor pool to execute jobs arising from our workflows. The polymorph search application is reasonably rich in that it not only involves massively parallel computations but also needs to handle the amount of data that is produced during the search. The total volume of data produced during an exhaustive search of a molecule is in the region of 6 GB, and scientists might wish to complete up to 40 studies during a month, producing a 0.25 TB of data per month. Processing these data during workflows involves conversion between the output of MOLPAK and the input format for DMAREL, transformation of results to the standardized Chemical Markup Language (CML) and enriching results with metadata about the computation prior to upload of selected search results to a data portal. This combination of parallel computation with data handling makes it a fairly representative scientific Grid application. More
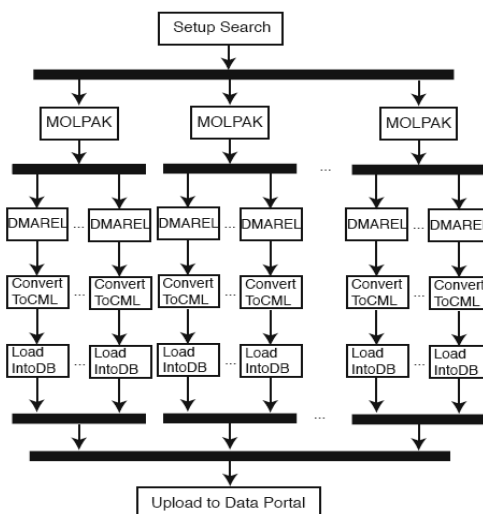
Figure 26.7: Overview of polymorph search workflow.

detailed information about the scalability and performance characteristics of the polymorph search workflow can be found in [132].

We will now briefly look at how such a workflow can be expressed as a BPEL workflow in our editor. The main workflow is indexedMolpak, which relies on a number of subworkflows. The main workflow starts by gathering some input data, such as the list of packing types, and then invokes the invokeMolpak workflow via its WSDL interface supplying these data. From the prepared input data, the invokeMolpak workflow generates the JSDL required to actually execute MOLPAK. It then uses an invocation to a further subworkflow (gssubmit). This subworkflow invokes the GridSAM job submission Web service and passes the JSDL it has received to GridSAM. GridSAM will then translate the received JSDL into a script for the underlying Grid scheduler (i.e., a Condor script), which then takes care of executing the actual jobs on the Grid. The gssubmit workflow continually checks the job status by repeatedly invoking the GridSAM job monitoring Web service and eventually replies to its caller, depending on the status of the jobs (i.e., completed or failed). The main workflow then uses the resulting data to prepare the input data for use by the invokeDmarel workflow and invokes this workflow. InvokeDmarel then needs to carry out some data manipulation on the input data, prepare the JSDL needed to execute DMAREL, and invoke gssubmit in the same way as invokeMolpak has done. The DMAREL invocations issued from within invokeDmarel operate on the data resulting from the MOLPAK runs. The main workflow eventually receives the results from all computations and stores them in an XML file that contains a set of Chemical Markup Language (CML) [307] crystal structures. As each set of results from invokeDmarel becomes available, we invoke a visualizer workflow, which prepares the results in various formats, such as a tabular format and a scatter plot. This visualization can be updated in real time as further results become available.

The indexedMolpak workflow presents some interesting features. First of all, we note the use of two indexed flows, one for the invocations of MOLPAK and one for the invocation of DMAREL. This allows the concurrent execution of, say, 200 DMAREL invocations with little effort required from a modeling perspective. In order to change the precision (number of DMARELs to run), the end range of the relevant indexed flow needs to be changed in the properties view. This compares very favorably with the native flow construct BPEL has to offer, which would require us to repeat the same information 200 times! Another feature whose importance can be illustrated by looking at a large, real-world workflow like this one is the importance of finding ways of compressing the graphical representation. This is, for example, achieved by representing scopes as containers, which contain activities and also display all their variables and means for adding or modifying these variables in one place. Furthermore, this workflow is a good example of how hierarchical composition of workflows can help to significantly reduce the modeling complexity and achieve reusability of common workflows (i.e.,

gssubmit, workflows for visualization of output data). Transparency of the underlying Grid scheduler being used to execute jobs on a Grid is achieved by using GridSAM. Scientists only need to define the JSDL for their jobs once and not worry about which scheduler is used now or at any point in the future.

The polymorph search workflow provides us with reassurance that, given an appropriate set of language constructs for large-scale workflows and given that adequate tool support and middleware integration can be established, the use of BPEL by application scientists can become a reality.

26.5 Related Work

A number of industrial modeling tools have become available for BPEL recently. All tools of which we are aware provide some means of visual modeling. They usually provide a one-to-one mapping from their visual constructs to the ones in BPEL and are primarily targeted at software engineers who possess knowledge of BPEL, WSDL, XSD, and other related technologies. Therefore, they usually lack higher-level abstractions and sufficient support for non-expert users. IBM Alphaworks offers the BPWS4J editor for free download. This editor relies on a tree-based one-to-one representation of BPEL and hence is not capable of dealing with the requirements of scientific workflows. Oracle's BPEL Designer is a free Eclipse plug-in. Again, it provides a one-to-one mapping to BPEL, but, in addition to that, offers macros, which can be used to arrange sets of activities into reusable components. The Oracle tool also offers a flowN construct, which is similar to our indexed flow activity. However, this construct can only be interpreted by Oracle's BPEL engine. ActiveWebflow is another Eclipse-based editor offered by ActiveEndpoint. At the time of writing, ActiveWebflow is a commercial editor and we therefore only have limited experience with it. A notable feature of this editor is a debugger that enables step-by-step debugging of a BPEL process on a local machine. Again, this editor is also tied to a specific BPEL engine, in this case the ActiveBPEL engine. The main differences of our editor, especially in future incarnations, will be found in its focus on additional language abstractions, a number of usability features, whose aim is to hide BPEL as far as possible, and the support of several BPEL engines.

Taverna (see Chapter 19 for more information) is a workflow modeling and enactment environment primarily used by applications in bioinformatics and developed as part of the myGrid project. Taverna does support Web services, but it does not rely on an industry standard for the orchestration of Web services such as BPEL. In Taverna, due to the heterogeneity of service in bioinformatics, data are always of type string, which provides a great deal of flexibility at the expense of complicating validation of data compatibility. In our work, we primarily focus on an industry standard as our workflow language and attempt to make it accessible to scientists by integrating the tools and technologies that have been developed for this standard. When comparing

Sedna with Taverna, one of the benefits of relying on BPEL becomes apparent: We can make use of BPEL's type system to provide the kind of validation mentioned above.

Triana (see Chapter 20)provides a GUI that allows users to drag services onto a canvas and to connect these services to each other. Triana supports a subset of BPEL and can export its workflows into BPEL. Again, our approach differs in that we aim to make the power of BPEL directly available to users by hiding its complexity as far as possible. We believe that the focus on a single workflow language enables extensive and targeted support to users.

GridFlow [76] is a workflow management system for Grid computing and as such focuses on resource allocation, as do Condor and the Globus GRAM. The GridFlow Portal is a simple GUI used for the definition and monitoring of workflows. The support users receive in Sedna is more sophisticated than that required for GridFlow. Furthermore, by using GridSAM, we separate the definition of a workflow from the issues involving resource allocation.

GridAnt [22] allows users to make use of the Ant batch language for the definition and monitoring of Grid workflows. GridAnt offers extensions to the Ant language and requires users to engage in textual programming in Ant's XML format. Disconnection of the client submitting a workflow cannot be achieved effortlessly (some form of proxy mechanism is required), even though this is an important feature given the long-running nature of scientific workflows on the Grid. In contrast to GridAnt, we have chosen to use a full-fledged workflow language.

The Grid Services Flow Language (GSFL) [246] represents an attempt to provide a workflow language with additional support for Grid service life cycle management and P2P service invocation without relying on standards such as WS-Notification. Our work exploits the fact that BPEL is an industry standard for which sufficiently scalable and robust enactment environments are available. We are not aware of an available enactment environment or any kind of tool support for GSFL, but we believe that comparing a similar environment based on GSFL would be interesting.

Efforts led by John Grundy [180] [181] focus on providing visual languages and tools targeted at particular application domains such as software process modeling, flexible CASE tools, and complex data mappings. The software engineering tools developed in his work allow visual representation of domain concepts and their translation into code. We have, so far, not focused on a particular application domain but aim to make BPEL usable for the expression of scientific workflows in general.

26.6 Lessons Learned and Future Work

There is a need for composition of Grid services into workflows in scientific Grid environments, and the use of BPEL for this purpose promises many benefits but at the same time presents a number of issues that need to be

addressed. We have seen how the verbosity of BPEL and its original target domain make its abstractions to a certain extent insufficient for use in scientific workflows and how the complexity of both its underlying technologies and the distribution middleware present an unacceptable burden to application scientists.

In order to introduce the potential benefits of BPEL to the scientific community, we have developed a first set of additional domain-independent language abstractions, such as the indexed flow, while still allowing for domain-specific extensions. We attempt to hide the complexity of BPEL and the underlying middleware technologies by providing extensive tool support. Our visual modeling environment is integrated within Eclipse and provides a transparent link to the enactment of workflows on BPEL engines and a Grid computing infrastructure. The value of our approach has been confirmed using a case study that has demonstrated how a large-scale scientific workflow is developed using our framework and its abstractions.

There is of course a long list of future work that we need to complete in order to increase the usefulness of our approach. One element of our future work will be to develop further additional language abstractions on top of BPEL to arrive at a comprehensive set of domain-independent constructs to facilitate the creation of scientific workflows. We will also work on better support for asynchronous interaction patterns in the form of workflow templates and additional wizardry. Ultimately, this wizardry will allow our users to exploit the capabilities of BPEL without the burden of having to become thoroughly acquainted with it. The conciseness of the graphical representation can be enhanced by collapsable containers. We will investigate a scalable and informative mechanism for real-time monitoring of processes reusing the existing graphical representation of a workflow in the editor and also work on a tool that provides graphical debugging facilities similar to that offered by Eclipse to Java developers. Two other noteworthy features are the automatic translation of BPEL files into our nonstandard BPEL constructs and a WSDL browser enabling selection of services by using semantic markup information.

Each of the features above will further simplify the modeling of scientific workflows in BPEL by non-expert users. Our experience to date indicates that with an appropriate set of abstractions and adequate tool support that successfully hides the complexity of the underlying technologies, BPEL is a promising language for scientific workflows.

26.7 Acknowledgments

ASKALON: A Development and Grid Computing Environment for Scientific Workflows

Thomas Fahringer, Radu Prodan, Rubing Duan, Jürgen Hofer,
Farrukh Nadeem, Francesco Nerieri, Stefan Podlipnig,
Jun Qin, Mumtaz Siddiqui, Hong-Linh Truong, Alex Villazon, and
Marek Wieczorek

27.1 Introduction

Most existing Grid application development environments provide the application developer with a nontransparent Grid. Commonly, application developers are explicitly involved in tedious tasks such as selecting software components deployed on specific sites, mapping applications onto the Grid, or selecting appropriate computers for their applications. Moreover, many programming interfaces are either implementation-technology-specific (e.g., based on Web services [24]) or force the application developer to program at a low-level middleware abstraction (e.g., start task, transfer data [22, 153]). While a variety of graphical workflow composition tools are currently being proposed, none of them is based on standard modeling techniques such as Unified Modeling Language (UML).

In this chapter, we describe the ASKALON Grid application development and computing environment (see Figure 27.1) [137], whose ultimate goal is to provide an invisible Grid to the application developers. In ASKALON, the user composes Grid workflow applications graphically using a UML-based workflow composition and modeling service. Additionally, the user can programmatically describe workflows using the XML-based Abstract Grid Workflow Language (AGWL), designed at a high level of abstraction that does not comprise any Grid technology details. The AGWL representation of a workflow is then given to the ASKALON WSRF-based middleware services (runtime system) for scheduling and reliable execution on Grid infrastructures.

The *Resource Manager* service is responsible for negotiation, reservation, allocation of resources, and automatic deployment of services required to execute Grid applications. In combination with the AGWL, the Resource Manager shields the user from the low-level Grid middleware technology.

The *Scheduler* is a service that determines effective mappings of single or multiple workflows onto the Grid using graph-based heuristics and
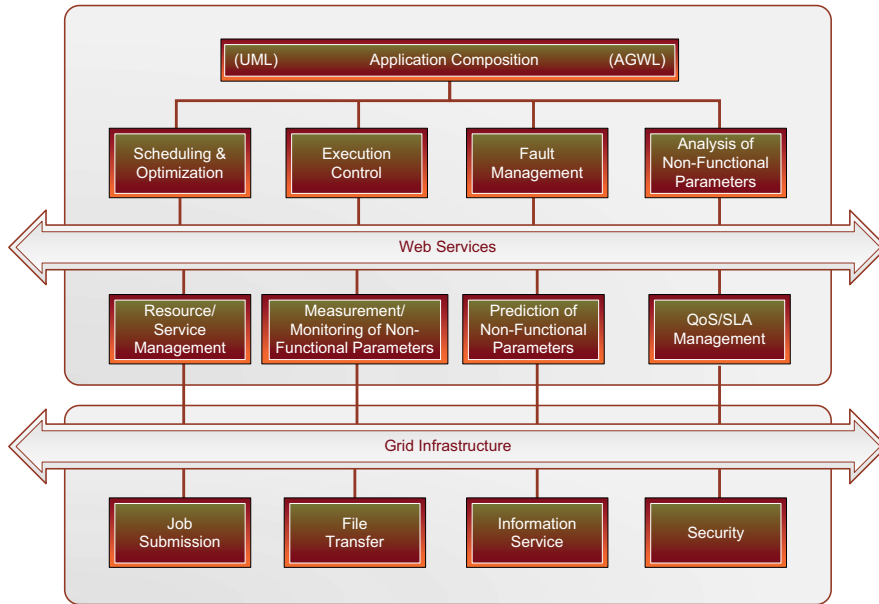
Figure 27.1: The ASKALON architecture.

optimization algorithms that benefit from Performance Prediction and Resource Manager services. Additionally, the Scheduler provides Quality of Service (QoS) by dynamically adjusting the optimized static schedules to meet the dynamic nature of Grid infrastructures through execution contract monitoring [365].

The *Execution Engine* service targets reliable and fault-tolerant execution of workflows through techniques such as checkpointing, migration, restart, retry, and replication.

Performance Analysis supports automatic instrumentation and bottleneck detection (e.g., excessive synchronization, communication, load imbalance, inefficiency, or nonscalability) within Grid workflow executions. We are currently extending our analysis to comprise service-level negotiation and agreement on a variety of nonfunctional parameters.

A *Performance Prediction* service estimates execution times of workflow activities through a training phase and statistical methods using the Performance Analysis service.

27.2 Workflow Case Study and Grid Infrastructure

We present the functionality of the ASKALON environment for modeling, specification, scheduling, and performance-oriented execution of scientific

workflows in the context of a real-world material science application deployed on the Austrian Grid infrastructure.

WIEN2k [52] is a program package for performing electronic structure calculations of solids using density functional theory, based on the full-potential (linearized) augmented plane-wave ((L)APW) and the local orbital (lo) method. We have ported WIEN2k as a Grid application by splitting the monolithic code into several coarse-grained activities coordinated in a workflow, as illustrated in Figure 27.2. The LAPW1 and LAPW2 activities can be solved in parallel by a fixed number of so-called *k-points*. A final activity, *converged*, applied on several output files, tests whether the problem convergence criterion is fulfilled. The number of recursive loops is statically unknown.

For the experiments that we will present throughout this chapter, we have solved a problem case with 252 parallel k-points (i.e., size of the two parallel sections – LAPW1 and LAPW2). We have chosen a problem size of 8.5, which represents the number of plane-waves used and is equal to the size of the eigenvalue problem (i.e., the size of the matrix to be diagonalized).

ASKALON serves as the main application development and computing environment for the Austrian Grid [33] infrastructure, which aggregates over 300 processors geographically distributed across several different sites in
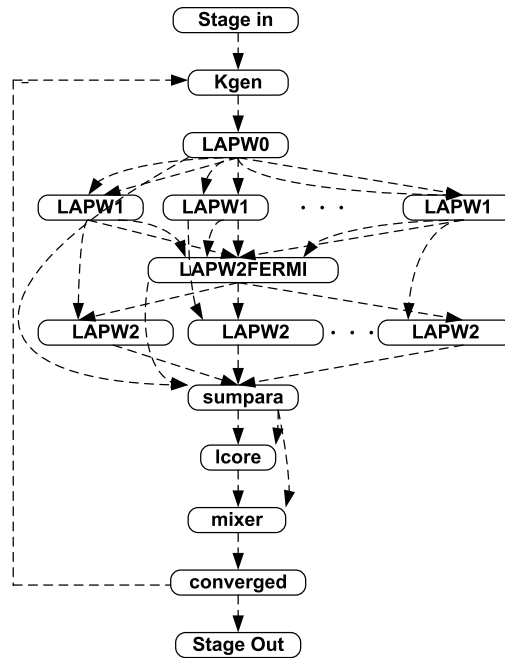
Figure 27.2: The WIEN2k workflow.

Site	Architecture	#	CPU	GHz	RAM	Mgr.	Location
Altix1.jku	NUMA, SGI Altix 3000	14	Itanium 2	1.6	61408	Fork	Linz
Altix1.uibk	NUMA, SGI Altix 350	14	Itanium 2	1.6	15026	Fork	Innsbruck
Schafberg	NUMA, SGI Altix 350	14	Itanium 2	1.6	15026	Fork	Salzburg
Gescher	COW, Gigabit Ethernet	16	Pentium 4	3	1024	PBS	Vienna
Agrid1	NOW, Ethernet	20	Pentium 4	1.8	512	PBS	Innsbruck
Arch19	NOW, Ethernet	20	Pentium 4	1.8	500	PBS	Innsbruck
Arch20	NOW, Ethernet	20	Pentium 4	1.8	500	PBS	Innsbruck
Arch21	NOW, Ethernet	20	Pentium 4	1.8	500	PBS	Innsbruck

Table 27.1: The Austrian Grid infrastructure.

Austria: Innsbruck, Linz, Salzburg, and Vienna. The Austrian Grid currently uses Globus Toolkit version 2 [144] as the main platform for security, job submission, file transfer, and resource information. Table 27.1 summarizes a subset of the Austrian Grid that we have used for the results presented in this chapter. The SGI Altix 3000 parallel computer in Linz has a total of 64 processors, while the Altix 350 computers in Innsbruck and Salzburg both comprise 16 parallel processors. However, the local system administrators only provide 14 concurrent processors to a regular Austrian Grid user, which is also the machine size that we used in our experiments. The Grid site in Vienna (Gescher) is a Beowulf cluster, while the four workstation network sites in Innsbruck are labs intensively used by students during the day but automatically rebooted in Grid mode during the night, weekends, or holidays.

27.3 Workflow Generation

ASKALON offers the users two interfaces for generating large-scale scientific workflows in a compact and intuitive form: graphical modeling using the UML standard (see Section 27.3.1) and a programmatic XML-based language (see Section 27.3.2).

27.3.1 UML Modeling-Based Workflow Composition

ASKALON offers to the end user the privilege of composing workflows through a graphical modeling service based on the UML standard that combines Activity Diagram modeling elements in a hierarchical fashion. We have implemented this graphical service as a platform-independent workflow editor in Java based on the Model-View-Controller paradigm comprising three main components: graphical user interface (GUI), model traverser, and model checker. The GUI consists of the following components: menu, toolbar, drawing space, model tree, and element properties. The drawing space consists of a tabbed panel that can contain several diagrams. The model traverser provides the possibility to walk through the model, visit each
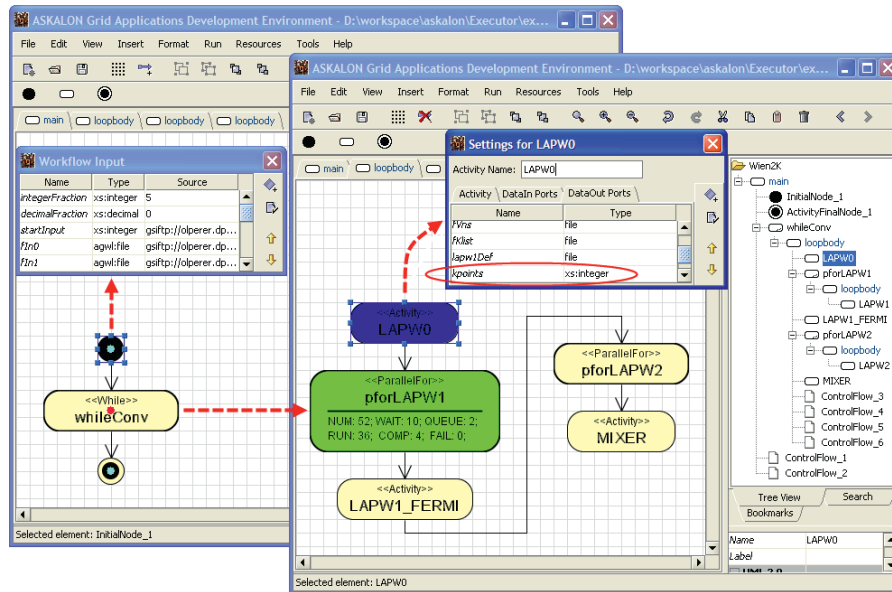
Figure 27.3: The WIEN2k UML representation.

modeling element, and access its properties (for instance, element name). We use the model traverser for the generation of various model representations; for instance, an XML representation serves as input for the ASKALON Grid environment. The model checker is responsible for the correctness of the model.

Figure 27.3 shows the UML representation of the WIEN2k workflow, which consists of several diagrams. The hierarchical representation allows the user to focus on the basic constructs of the full workflow and easily understand them. The left panel shows the main sequential outermost `while` loop, called `whileConv`, which embraces the entire workflow. The workflow inputs and outputs are specified by opening additional dialog boxes that define the input and the output ports of this activity. To display the loop body that implements one iteration of the `while` loop, the user selects the `Edit/Loop Body` menu item in the pop-up menu or selects the second `loop body` node. As a consequence, the loop body of the while loop is displayed, as shown in the right window, with arrows representing the control flow dependencies. For each activity, the user specifies typed data input and output ports through a special dialog box, as shown in the right window of Figure 27.3.

27.3.2 Abstract Grid Workflow Language

We have designed an XML-based workflow language that enables the description of workflow applications at a high level of abstraction that shields the user from the middleware complexity and dynamic nature of the Grid.

The *Abstract Grid Workflow Language* (AGWL) [138] enables the composition of workflow applications from atomic units of work called *activities* interconnected through control-flow and data-flow dependencies. Activities are represented at two abstract levels: *activity types* and *activity deployments*. An *activity type* is a simplified abstract description of functions or semantics of an activity, whereas an *activity deployment* (not seen at the level of AGWL but resolved by the underlying Resource Manager) refers to an executable or deployed Web service and describes how they can be accessed and executed on the Grid.

In contrast to most existing work, AGWL is not bound to any implementation technology such as Web services. The control-flow constructs include `sequences`, directed acyclic graphs (`dag`), `for`, `forEach`, `while` and `do-while` loops, and `if` and `switch` constructs, as well as more advanced constructs such as `parallel` activities, `parallelFor` and `parallelForEach` loops, and `collection` iterators. In order to modularize and reuse workflows, so-called subworkflows can be defined and invoked. Basic data flow is specified by connecting input and output ports between activities, while more advanced data-flow constructs include collections and access to abstract data repositories.

Optionally, the user can specify `properties` and `constraints` for activities and data-flow dependencies that provide functional and nonfunctional information to the runtime system for optimization and steering of the Grid workflow execution. Properties define additional information about activities or data links, such as computational or communication complexity, or semantic description of workflow activities. Constraints define additional requirements or contracts to be fulfilled by the runtime system that executes the workflow application, such as the minimum memory necessary for an activity execution or the minimum bandwidth required on a data-flow link.

The AGWL representation of a workflow can either be automatically generated from the UML representation or manually written by the end user. In both cases, AGWL serves as input to the ASKALON runtime middleware services (see Figure 27.1).

Figure 27.4 illustrates a representative excerpt of the WIEN2k AGWL representation, which can be automatically generated from the UML representation or manually written by the end user. The highest level of the WIEN2k workflow consists of a while loop `whileConv`. In this while loop, the activities `LAPW0`, `pforLAPW1` (parallel for loop), `LAPW2_FERMI`, `pforLAPW2` (parallel for loop), and `MIXER` are invoked sequentially. The activities `pforLAPW1` and `pforLAPW2` are parallel for loops that execute a large number (i.e., 252 for the case study considered) of `LAPW1` and `LAPW2`

```
<cgwd name="Wien2K">
  <cgwdInput>
    <dataIn name="startInput" type="agwl:file"
        source="gsiftp://…//…/WIEN2K/atype/STARTINPUT.txt"/> ...
  </cgwdInput>
  <cgwdBody>
    <while name="whileConv">
      <dataLoops>
        <dataLoop name="overflag" type="xs:boolean"
            initSource="Wien2K/overflag" loopSource="MIXER/overflag"/>
      </dataLoops>
      <condition> whileConv/overflag </condition>
      <loopBody>
        <activity name="LAPW0" type="wien:LAPW0">
          <dataIns>
            <dataIn name="startInput" type="…" source="Wien2K/startInput"/> ...
          </dataIns>
          <dataOuts>
            <dataOut name="kpoints" type="xs:integer"/> ...
          </dataOuts>
        </activity>
        <parallelFor name="pforLAPW1">
          <loopCounter name="lapw1Index" type="…" from="1" to="LAPW0/kpoints"/>
          <loopBody>
            <activity name="LAPW1" type="wien:LAPW1" …/>
          </loopBody>
          <dataOuts …/>
        </parallelFor>
        <activity name="LAPW1_FERMI" type="wien:LAPW1_FERMI" …/>
        <parallelFor name="pforLAPW2" …/>
        <activity name="MIXER" type="wien:MIXER" …/>
      </loopBody>
      <dataOuts>
        <dataOut name="overflag" type="xs:boolean" source="MIXER/overflag"/>
      </dataOuts>
    </while>
  </cgwdBody>
  <cgwdOutput>
    <dataOut name="overflag" type="xs:boolean" source="whileConv/overflag"
        saveto="gsiftp://…//…/WIEN2K/result/…"/>
  </cgwdOutput>
</cgwd>
```

Figure 27.4: WIEN2k AGWL excerpt.

activity invocations in parallel. It is important to notice at this stage that one runtime output port of the activity `LAPW0` called `kpoints` (see also Figure 27.3) represents the number of parallel loop iterations that will be executed by the following parallel loops (i.e., `pforLAPW1` and `pforLAPW2`), which is statically unknown. Therefore, the workflow can dynamically change its shape at runtime, depending on the runtime value of this output port. The condition to exit the outermost while loop refers to the data loop port `overflag`, which can be changed after each iteration by the data output port of the activity `MIXER` referred by the `loopSource` attribute. Finally, the output port `overflag` returns the final result of the workflow.

27.4 Resource Manager

ASKALON's Resource Manager, called GridARM, renders the boundaries of Grid resource management and brokerage and provides resource discovery,

advanced reservation, and virtual organization-wide authorization along with GLARE, a dynamic registration framework for activity types and activity deployments [383]. GridARM covers physical resources, including processors, storage devices, and network interconnections, as well as logical resources comprising Grid/Web services and executables.

Based on Scheduler requests, the GridARM discovers resources or software components, performs user authorization to verify resource accessibility, optionally makes a reservation, and returns the result. The result could be a list of resources along with their specifications, a list of software components, or a reservation ticket, depending on the request type. In case of a failure, a Resource Manager can interact with other GridARM instances distributed in the Grid to recursively discover and allocate the required resources. Moreover, the GridARM monitors the allocated resources and propagates exceptional situations to the client. It also works as coallocation manager.

Grid resource discovery and matching are performed based on the constraints provided by the Scheduler in the form of a resource request (see Section 27.5). The GridARM can be configured with one or more Monitoring and Discovery services [101] (of Globus versions 2 and 4) and the Network Weather Service [472].

Advanced reservation of the Grid resources (including computers and software components) based on the constraints provided by the requester is a distinguishing feature of the Resource Manager. The Scheduler can negotiate for a reservation based on time, cost, and QoS models. The essential attributes of a reservation include resource contact information, time frame, and resource requester and provider constraints. The acquisition of reserved resources by the Execution Engine is only possible by providing a valid user credential based on which the reservation was made or a valid reservation ticket.

GLARE, as part of GridARM, is a distributed framework for dynamic registration, automatic deployment, and on-demand provision of workflow activities. The framework provides an effective mapping between high-level application descriptions (called activity types) and actual installations (called activity deployments) on specific Grid sites. Activity types are described in a hierarchy of abstract and concrete types. Concrete types may have activity deployments that are shielded from the Grid application developer. On-demand deployment of activities is triggered automatically when they are requested by the client. GLARE is built based on a superpeer distributed framework.

Figure 27.5 illustrates a real-world example of a concrete activity type of WIEN2k [52] called `wien:lapw0`, which inherits generic `wien2k` and `wien` types. The activity type `wien:lapw0` can have two activity deployments: a legacy executable, `lapw0`, and a WSRF-compliant service called `WS-WienLAPW0`, both visible to the GLARE framework only internally. GLARE performs on-demand installation of these activity deployments and maps them automatically to the activity types, thus shielding the Grid from the application developers.

Activity Types

Wien

```
<ActivityType name="wien" type="">
 <domain name="SolidStatePhysics"/>
 <InputArgument name="input" type="file"/>
</ActivityType>
```

wien2k

```
<ActivityType name="wiek2k" type="wien">
 <InputArgument name="StartInput" type="File"/>
 <InputArgument name="IntegerFraction" type="file"/>
 <InputArgument name="decimalInput" type="xsd:integer"/>
 <OutputArgument name="out" type="file" saveTo="/tmp"/>
</ActivityType>
```

wien:LAPW0

```
<ActivityType name="wien:LAPW0" type="wien2k"/>
 <Installation>
  <Constraints />
  <BuildFileURL url="http://dps.uibk.ac.at/~mumtaz/glare/wien2k.build"/>
 </Installation>
 <Deployment EndpointReference="..lapw0..."/>
 <Deployment EndpointReference="...WS-WienLAPW0..."/>
</ActivityType>
```

Visible to Clients/Application Developers

Invisible to Clients/Application Developers   Visible to GLARE

lapw0

```
<ActivityDeployment name="lapw0" type="wien:LAPW0">
 <nodeName="schareck.askalon.org" path="/opt/wien2k/bin"
         home="/home/mumtaz/wien2k"/>
</ActivityDeployment>
```
Executable

Activity Deployments

WS-WienLAPW0

```
<ActivityDeployment name="WS-WienLAPW0"
         type="wien:LAPW0"> <service EndpointReference=" .. .. .."/>
</ActivityDeployment>
```
Web/Grid Service
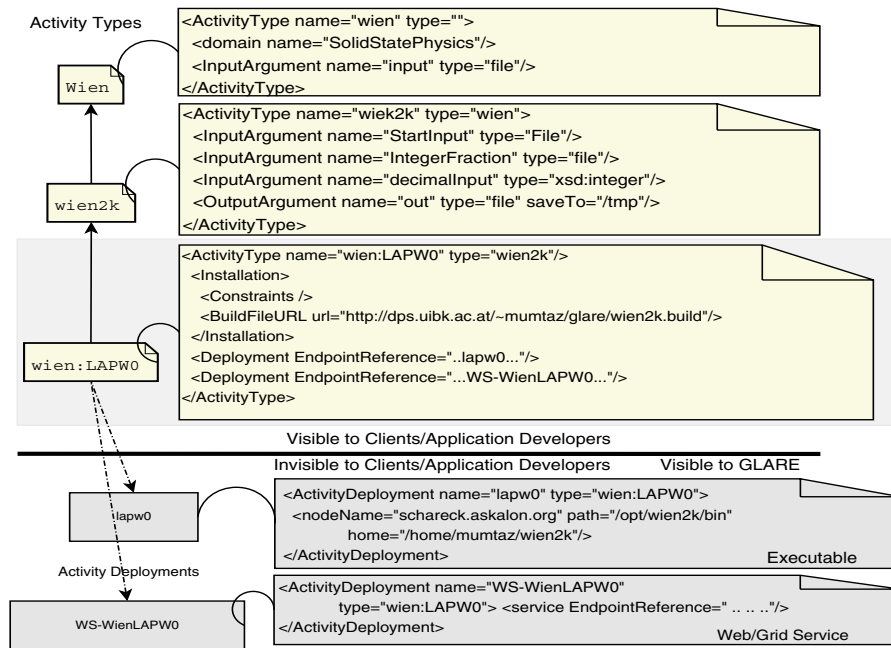
Figure 27.5: WIEN2k activity type to deployment mapping.

We have developed a graphical console to GridARM and GLARE that can be used to browse current Grid sites and the activity types available (see left panel of Figure 27.6). For each Grid site or activity type, the concrete deployments can be retrieved. Additionally, the user can use this tool to add/remove, register/unregister, and update activity types and deployment entries and automatically deploy/undeploy actual software components (services and executables).

GLARE provides a special activity type called `askalon.service` that stores configuration and deployment information of the ASKALON middleware services (i.e., Scheduler, Execution Engine, GridARM). This activity type is the entry point to the ASKALON middleware services by providing the required discovery and invocation functionality. Together with AGWL, GLARE has the responsibility of shielding the user from low-level Grid middleware and resource details.

27.5 Scheduler

The Scheduler service prepares a workflow application for execution on the Grid. It processes the workflow specification described in AGWL, converts it to an executable form, and maps it onto the available Grid resources.
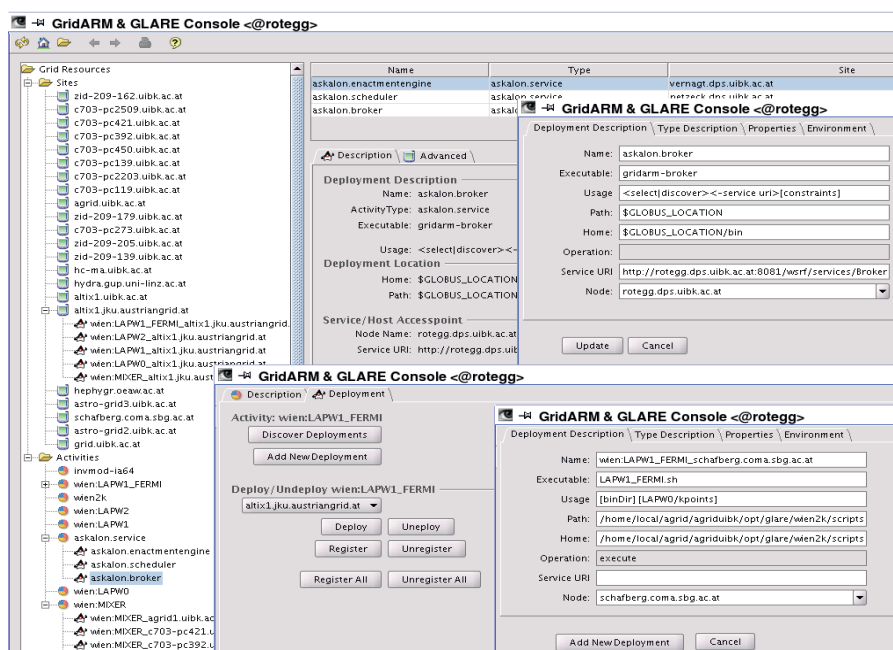
Figure 27.6: GridARM and GLARE console snapshots.

The scheduling process starts when the Execution Engine sends a scheduling request with a workflow description. The workflow consists of nodes representing activity types connected through control- and data-flow dependencies, as well as overall workflow input and output data. The Scheduler uses the Resource Manager to retrieve the current status of the Grid resources and to determine available activity deployments that correspond to the workflow activity types. In addition, the queries submitted by the Scheduler to the Resource Manager can contain constraints that must be honored, such as processor type, minimum clock rate, or operating system. The Performance Prediction service supplies predicted activity execution times and data transfer times required by the performance-driven scheduling algorithms.

The scheduling process consists of three main phases: (1) *refinement*, performed by the workflow converter component of the Scheduler; (2) *mapping*, performed by the scheduling engine component; and (3) *rescheduling* upon important events triggered by the event generator component (see Figure 27.7).

27.5.1 Workflow Refinement

The *workflow converter* resolves all the ambiguities and refines sophisticated workflow graphs into simple directed acyclic graphs (DAGs) on which existing graph-scheduling algorithms can be applied. Initially, several assumptions are made for various workflow parameters such as conditionals (e.g., `while`, `if`, `switch`) or loop iteration bounds (e.g., number of parallel loop iterations) that cannot be evaluated statically before the execution begins. Afterward, a set of refinements are applied to refine the original complex but compact workflow specifications into a pure DAG-based representation. Typical transformations include branch prediction, parallel loop unrolling, and sequential loop elimination. Transformations based on correct assumptions can imply substantial performance benefits, particularly if a strong imbalance in the workflow is predicted. Incorrect assumptions require appropriate runtime adjustments such as undoing existing optimizations and rescheduling based on the new Grid information available.

27.5.2 Workflow Scheduling

The *scheduling engine* is responsible for the actual mapping of a converted workflow onto the Grid. It is based on a modular architecture, where different DAG-based scheduling heuristics can be used interchangeably. The algorithms with varying accuracy and complexity are based on different metrics as optimization goals. We have currently incorporated three scheduling algorithms: Heterogeneous Earliest Finish Time (HEFT) [493], a genetic algorithm [365], and a myopic just-in-time algorithm acting like a resource broker, similar to the Condor matchmaking mechanism used by DAGMan (see Chapter 22). All algorithms receive as input two matrices, representing the predicted execution time of every activity instance on each computation architecture and the predicted transfer time of each data dependency link on every Grid site interconnection network, and deliver a Grid schedule.
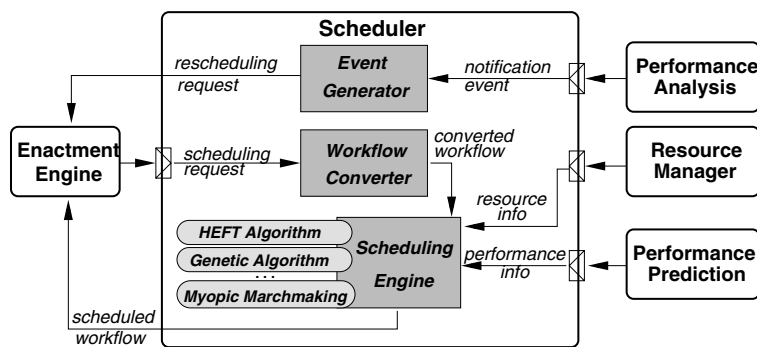
Figure 27.7: The Scheduler architecture.

As explained in Section 27.3.2, a peculiarity of the WIEN2k workflow is that the number of parallel activities is unknown until the first activity finishes its execution and instantiates its output port `kpoints`. As a consequence, the workflow converter initially assumes a value of one for this output port, which produces a schedule that serializes all workflow activities onto the fastest Grid processor available. In order to graphically display the Gantt chart produced by the Scheduler (see Figure 27.8(a)), we have customized and integrated the Jumpshot tool [491] originally developed for postmortem visualization of MPI(CH) programs.

27.5.3 Workflow Rescheduling

After the initial scheduling, the workflow execution is started based on the current mapping until the execution finishes or any interrupting event occurs. The *event generator* module uses the Monitoring Service to watch the workflow execution and detect whether any of the initial assumptions, also called *execution contracts*, have been violated. The execution contracts that we currently monitor include structural assumptions made by the workflow converter, external load on processors, processors no longer available, congested interconnection networks, or new Grid sites available. In case of a contract violation, the Scheduler sends a rescheduling event to the Execution Engine, which generates and returns to the Scheduler a new workflow based on the current execution status (by excluding the completed activities and including the ones that need to be reexecuted). We have formally presented this approach in detail in [365].

In the case of WIEN2k workflow, the number of parallel activities `kpoints` is determined after the first activity completes, which triggers a rescheduling event because of a workflow structural change. Figure 27.8(b) illustrates a sample Gantt chart upon a rescheduling event for three Grid sites and 100 parallel activities (`kpoints`). One can clearly see the two parallel activities LAPW1 and LAPW2, whose inner activities are distributed across all processors available. The middle sequential activity LAPW2_FERMI synchronizes the parallel activities of LAPW1 and scatters them once again for the next parallel activity, LAPW2. One can also notice that at least two parallel activities are serialized on each processor, which we will interpret as a serialization overhead in Section 27.7.

Figure 27.8(c) illustrates the outcome of applying our scheduling algorithms at the rescheduling events generated by the completion of the first activity. The results show that optimization algorithms such as HEFT and genetic search produce substantially better schedules than the myopic matchmaking. HEFT is also superior to the genetic algorithm since it is a workflow-specific heuristic highly suitable for heterogeneous environments such as the Grid. The full-graph scheduling approach produces better results than the workflow partitioning strategy [116], especially in the case of strongly unbalanced workflows when one parallel iteration contains significantly more
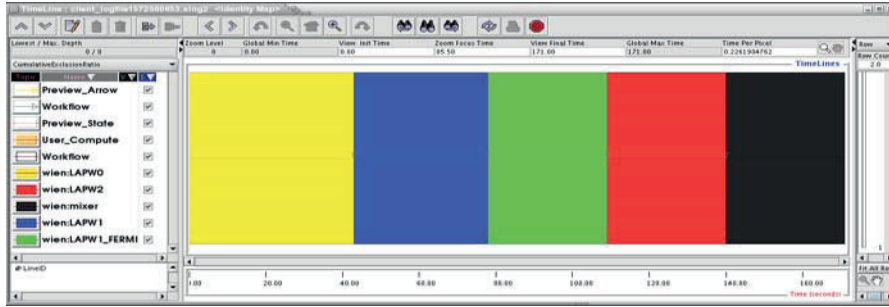
(a) Initial Gantt chart.
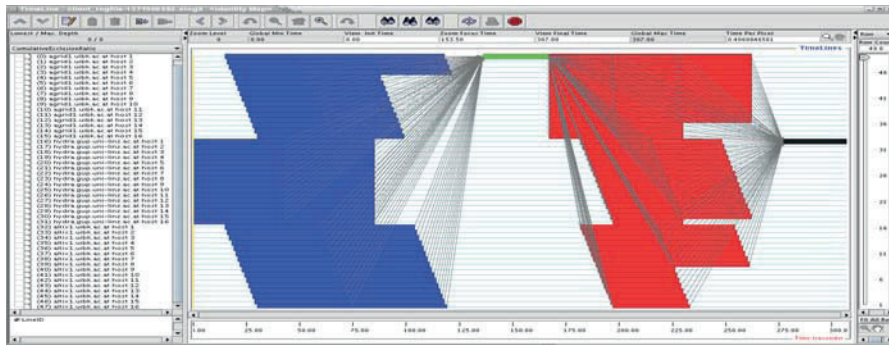
(b) Gantt chart after rescheduling.
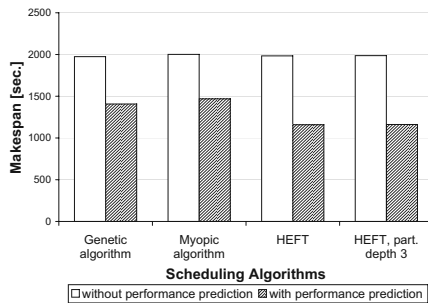
(c) WIEN2k execution time.
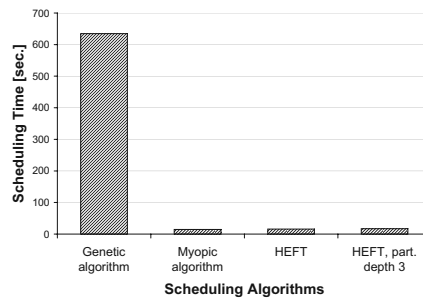
(d) WIEN2k scheduling time.

Figure 27.8: Scheduling experimental results.

work than the others. We can also notice that the genetic algorithm needs two orders of magnitude longer than the other algorithms for achieving results of the same quality; however, its ratio to the overall workflow execution time is still negligible (see Figure 27.8(d)).

27.6 Execution Engine

The Execution Engine is the central service of the ASKALON middleware responsible for controlling the execution of a workflow application based on the Grid mapping decided by the Scheduler. The main tasks performed by the Execution Engine are to coordinate the workflow execution according to the control-flow constructs (i.e., `sequence`, `if`, `switch`, `while`, `for`, `dag`, `parallel`, `parallelFor`) and to effectively resolve the data-flow dependencies (e.g., activity arguments, I/O file staging, high-bandwidth third-party transfers, access to databases) specified by the application developer in AGWL.

The Execution Engine provides flexible management of large collections of intermediate data generated by hundreds of parallel activities that are typical of scientific workflows. Additionally, it provides a mechanism to automatically track data dependencies between activities and performs static and runtime workflow optimizations, including archiving and compressing of multiple files to be transferred between two Grid sites or merging multiple activities to reduce the job submission and polling for termination overheads.

The Execution Engine provides fault tolerance at three levels of abstraction: (1) *activity level*, through retry and replication; (2) *control-flow level*, using lightweight workflow checkpointing and migration (described later in this section); and (3) *workflow level*, based on alternative task workflow-level redundancy and workflow-level checkpointing.

Checkpointing and recovery are fundamental techniques for saving the application state during normal execution and restoring the saved state after a failure to reduce the amount of lost work. The Execution Engine provides two types of checkpointing mechanisms, described below.

Lightweight workflow checkpointing saves the workflow state and URL references to intermediate data (together with additional semantics that characterize the physical URLs) at customizable execution time intervals. The lightweight checkpoint is very fast because it does not backup the intermediate data. The disadvantage is that the intermediate data remain stored on possibly unsecured and volatile file systems. Lightweight workflow checkpointing is typically used for immediate recovery during one workflow execution.

Workflow-level checkpointing saves the workflow state and the intermediate data at the point when the checkpoint is taken. The advantage of the workflow-level checkpointing is that it saves backup copies of the intermediate data into a checkpoint database such that the execution can be restored and resumed at any time and from any Grid location. The disadvantage is that the checkpointing overhead grows significantly for large intermediate data.

The Execution Engine provides a comprehensive monitoring interface through which the user can observe online various metrics that characterize the progress of the overall workflow execution. First of all, the user is provided with a dialogbox displayed in Figure 27.9(a), that enables one to customize personal monitoring metrics and charts, including histograms, line charts, or
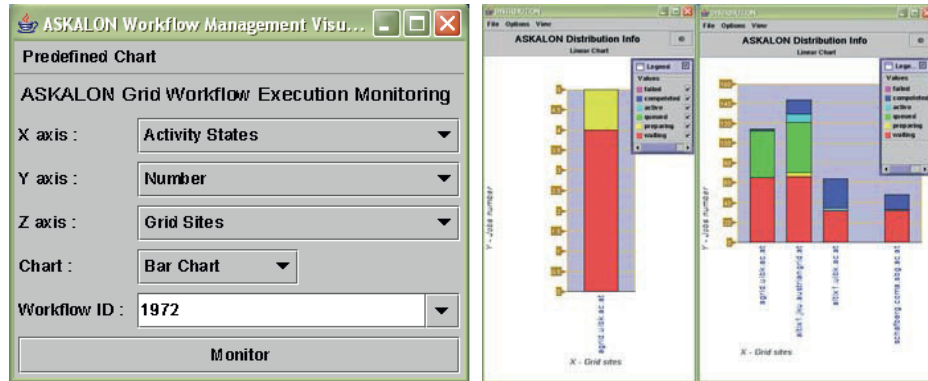
(a) Monitoring dialog box.

(b) Activity state distribution bar charts.
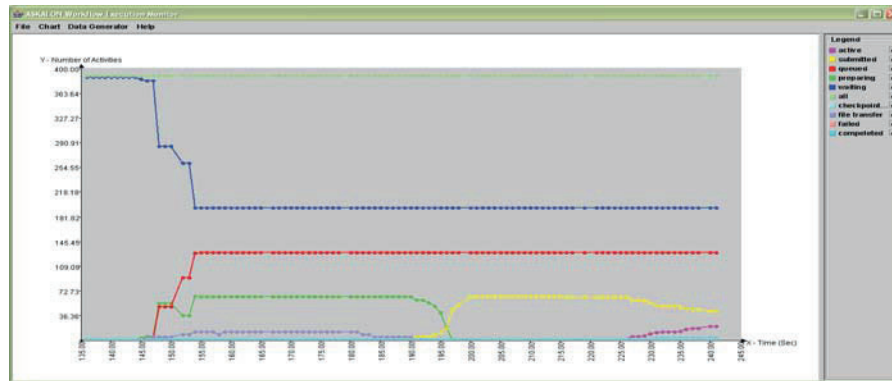
(c) Activity state execution histogram.

Figure 27.9: Online workflow-monitoring snapshots.

online bar charts. Figure 27.9(b) displays two online bar charts that show the workflow activity states on each Grid site before and after the rescheduling phase. The left chart shows that the activity LAPW0 is in a preparation phase, while the following ones are waiting to be executed on the same site due to control-flow dependencies and assuming one k-point per LAPW1 and LAPW2 parallel section (see Section 27.5). The right chart of Figure 27.9(b) displays an execution snapshot after rescheduling the workflow on four Grid sites, where each bar corresponds to one Grid site and displays the number of activities in each state scheduled on that site. The histogram in Figure 27.9(c) traces at regular customizable time instances the number of workflow activities in each possible state (i.e., waiting, preparing, submitted, active, checkpointing, completed, failed, data transfer).

27.7 Overhead Analysis

One of the main concerns when executing scientific workflows on the Grid is achieving faster completion times proportional to the performance or quality of the Grid resources available. The distributed execution of workflow applications on the Grid, however, is prone to large overhead that must be understood in order to improve the overall speedup and efficiency.

As part of a service that analyzes nonfunctional parameters for Grid applications, we have developed a formal overhead analysis model that defines the execution time of a Grid workflow application as the sum between a theoretical *ideal time* T_{ideal} and a set of *temporal overheads* that originate from various sources:

$$T = T_{ideal} + Total\_Overhead.$$

Describing the sources of the temporal overheads, classifying them in a fine-grained hierarchy, and measuring them in a systematic manner is the scope of our overhead analysis effort.

We model a *workflow application* as a directed graph (*Nodes*, *Edges*), where $Nodes = \{N_1, \ldots, N_n\}$ is the set of workflow nodes and $Edges = \bigcup_{i=1}^{n-1} (N_i, N_{i+1}) \bigcup \{(N_j, N_k) \mid j > k\}$ is the set of *control-flow dependencies*. The edges in the latter union set model backward dependencies that implement recursive loops. A node N can have any of the following types: (1) *computational activity* (or remote job submission), *CA*; (2) *data transfer activity* (or file transfer between Grid sites), *DT*; (3) *parallel section*, denoted as $N_{Par} = (N_{p1}, \ldots N_{pm})$; and (4) *subworkflow*, denoted as (*Nodes_i*, *Edges_i*), recursively defined according to this definition.

For the moment, we ignore in our analysis the arbitrary DAG-based workflow structures, which are nevertheless supported by AGWL.

The *ideal execution time of a sequential computational activity, CA*, denoted as T_{CA}^{ideal}, is the minimum of the wall-clock execution times on all idle processor types available on the Grid. The *ideal execution time of a data transfer activity* is zero since we consider it an overhead: $T_{DT}^{ideal} = 0$. The *ideal execution time* of a parallel section N_{Par} is the fastest ideal execution time of a single computational activity: $T_{N_{Par}}^{ideal} = \min_{\forall CA \in N_{Par}} \{T_{CA}^{ideal}\}$. The *ideal execution time* of a workflow (*Nodes*, *Edges*) is the sum of the ideal execution times of all (computational and parallel section) activities $N \in Nodes$: $T_{ideal} = \sum_{\forall N \in Nodes} T_N^{ideal}$.

We propose a new hierarchical classification of performance overhead for Grid workflow applications, which we currently base on four main categories: *middleware, loss of parallelism, data transfer*, and *activity* overheads (see Figure 27.10). Additionally, we consider the difference between the total overhead and the sum of the measured identified (nonoverlapping) overheads as *unidentified overhead*, which has to be minimized by an effective performance analysis tool. A high unidentified overhead value indicates that
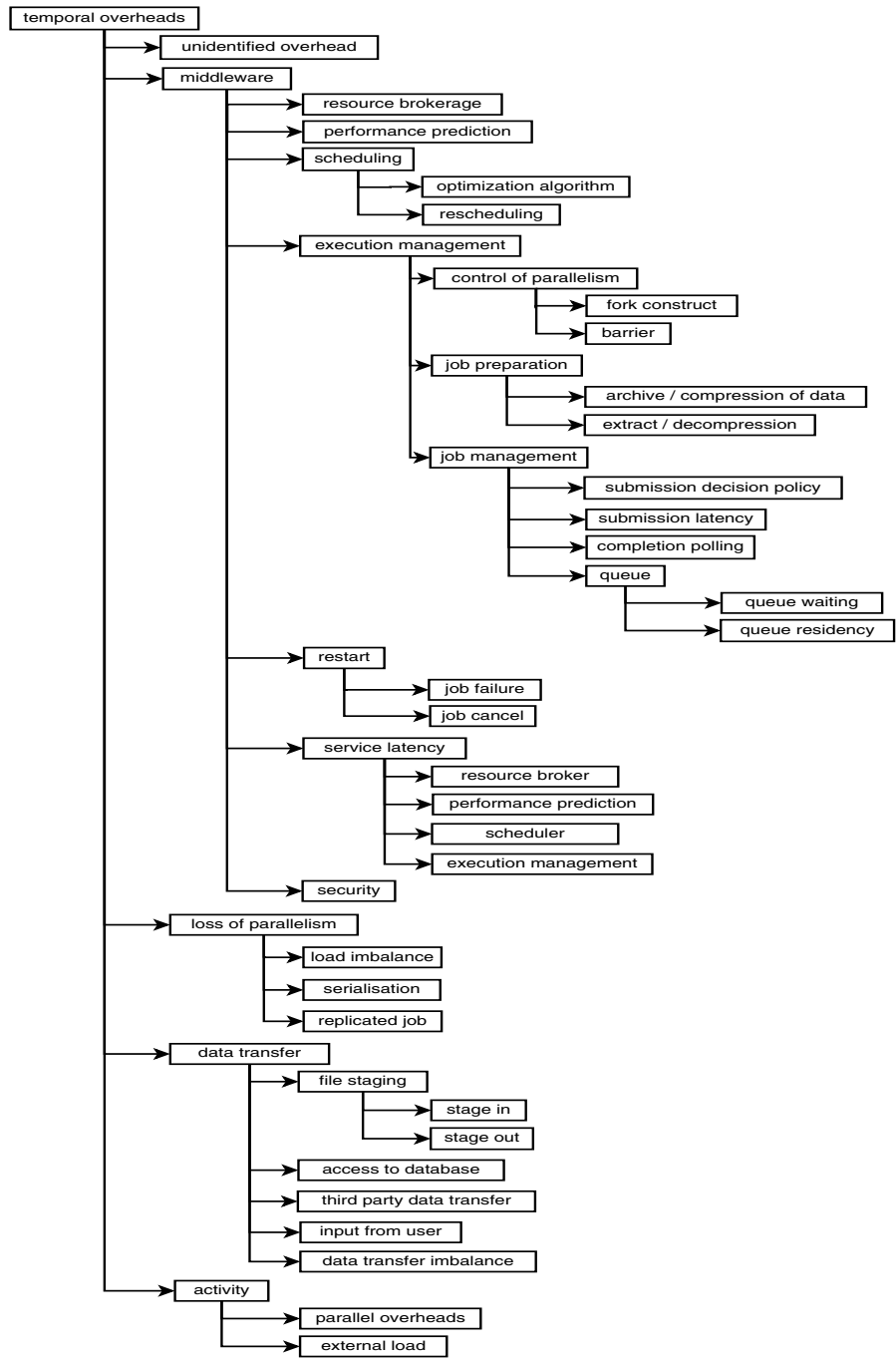
Figure 27.10: Workflow overhead classification.

the analysis is unsatisfactory and further effort is required to spot new sources of overhead in the workflow execution.

Normalized metrics are valuable means for understanding the importance of the temporal overheads with respect to the entire workflow execution. We define the value of an overhead T_o normalized against the workflow execution as the *overhead severity*, which quantifies the importance of the performance overhead for the specific workflow execution: $SV = \frac{T_o}{T}$.

In addition, we define the workflow *speedup* as the ratio between the fastest single-site execution time (of the entire workflow) T_{seq}^M and the actual execution time of the workflow on the Grid T:

$$S = \frac{\min\limits_{\forall M \in Grid} \{T_{seq}^M\}}{T}.$$

Furthermore, we define the workflow *efficiency* as the speedup normalized against the number of the Grid sites used, where each Grid site M is weighted with the speedup of the corresponding single site execution time:

$$E = \frac{S}{\sum_{M \in Grid} S_M}, \text{ where: } S_M = \frac{\min\limits_{\forall M' \in Grid} \left\{T_{seq}^{M'}\right\}}{T_{seq}^M}.$$

The efficiency formula therefore becomes:

$$E = \frac{T^{-1}}{\sum_{\forall M \in Grid} (T_{seq}^M)^{-1}}.$$

The fastest Grid site has a weight of one, whereas the slowest Grid site has the smallest weight (i.e., closest to zero).

The rationale behind the speedup formula is that only by normalizing against the fastest parallel computer available can the scientists get a true indication of what they can gain by extending their focus to Grid computing.

Beyond a detailed overhead analysis, our experiments try to answer the following question: Assume that we execute and measure the execution time of a workflow on the fastest Grid site available and thereafter incrementally add the next fastest site to the Grid execution environment. Does the execution time of a specific application decrease considerably compared with the single-site execution? If we can demonstrate reasonable speedups and understand the nature of the most relevant overheads, the application groups are likely to become interested in Grid computing and give us additional support in porting their applications as Grid workflows.

We proceeded by executing the workflow application on every individual Grid site and ranked the sites according to the execution times obtained (see Table 27.1). In our experiments, the SGI Altix machines delivered the fastest execution times, followed by the 3 GHz Pentium 4 compute cluster from Vienna (Gescher) and the Pentium 4 workstation networks in Innsbruck
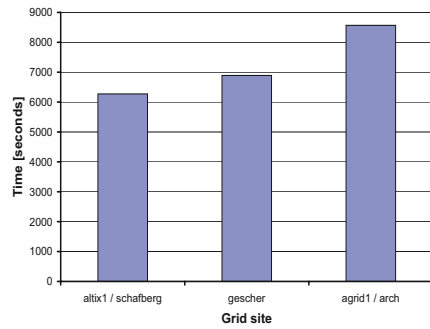
(a) Single site comparison.
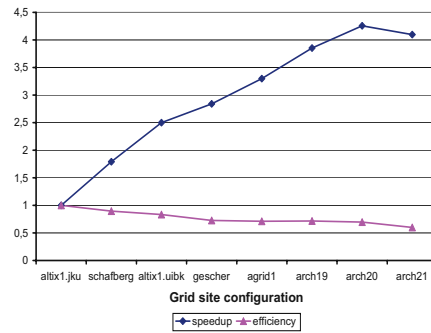
(b) Speedup and efficiency.
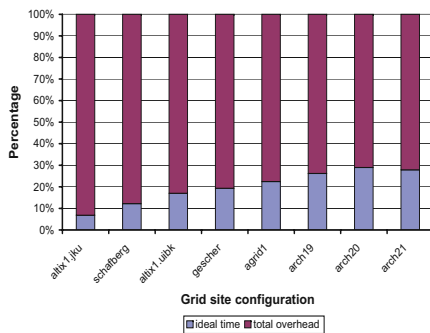
(c) Total overhead severity.
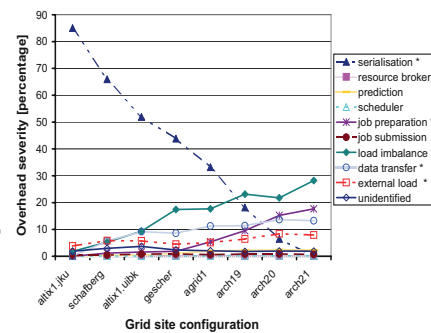
(d) Overhead severity.
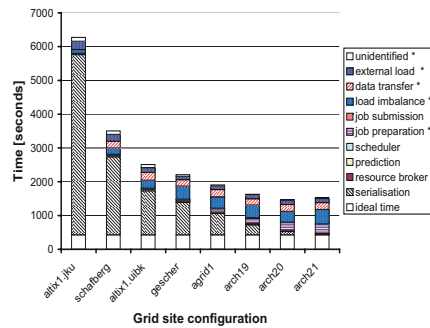
(e) 252 k-point executions.
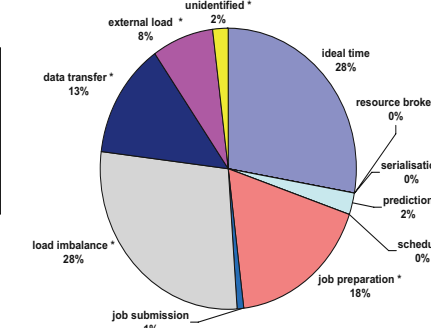
(f) Overhead breakdown for eight sites.

Figure 27.11: WIEN2k overhead analysis results.

(see Figure 27.11(a)). The ideal execution time of the SGI Altix machines and Gescher are almost equal, but Gescher has a significantly higher PBS queuing time (over one minute), which makes it only fourth in the overall Grid site

ranking. After establishing a ranking of the Grid sites, we proceeded with the repeated execution of the workflow on multiple Grid sites by incrementally adding the next fastest site to the Grid environment.

The speedup curve displayed in Figure 27.11(b) shows that the Grid execution improves up to seven sites. The improvement comes from the parallel execution of the computationally expensive k-points on multiple Grid sites, which significantly lowers the execution time of the parallel sections. The efficiency curve gently decreases in both cases and remains above 0.5, which we find promising considering the modest problem sizes executed and the rather high overhead experienced.

Figure 27.11(e) displays the contribution of the ideal execution time and the most significant overheads to the real execution time. We have marked with asterisks the most severe overheads, which the reader can follow in a top–bottom order on the individual bars.

The severity of the total overhead constantly decreases with the Grid size from over 80% on one site to 45% on eight Grid sites (see Figure 27.11(c)). Figure 27.11(d) summarizes in one graph the overhead severities in every Grid site configuration, which indicates the importance of each overhead and guides the application and middleware developers to the parts on which to concentrate the next tuning efforts.

The most important overhead is the serialization overhead due to the limited Grid size, which cannot accommodate the entire number of parallel activities that have to be serialized on some of the processors (typically through a job queuing system such as PBS). This overhead accounts for over 90% of the total overhead on a single site, but decreases to zero on eight sites. This overhead indicates the performance that could be gained by acquiring or adding new processors to the Grid environment. If extending the Grid size is irrelevant to the user, one could consider this overhead as part of the ideal execution time.

The second severe overhead is the loss of parallelism due to load imbalance, which fluctuates depending on the number of k-points, processors available, and the size of the serialized block of each processor. In our experiments, it steadily grows with the Grid size because of the slower processors added to the Grid environment (see Figure 27.11(f)).

The next important overhead is the job preparation overhead for compression/decompression of a large number of files into/from an archive with the purpose of reducing the data transfer overhead. The WIEN2k activities have a large number of data dependencies (i.e., files), which increase proportionally with the number of parallel k-points (about three times). Moreover, the size of the data that need to be transferred between activities increases with the number of k-points and Grid sites (about 500 MB for 100 k-points). Therefore, it becomes crucial to archive and compress the output files before transferring them over the network to the next activity. This overhead remains relatively constant for the first four Grid site configurations. The Pentium 4 workstation networks (which are part of a large, intensively

used student workstation network), however, exhibit unexpectedly large access latencies to the shared AFS file system upon decompressing file archives of about 50 MB. This overhead grows linearly with the number of archives used (i.e., $n - 1$ tar archives for n Grid sites), which significantly slows down the execution.

We managed to keep the data transfer overhead relatively constant (about 140 seconds) by using parallel streams over the GridFTP protocol to transfer the archives between sites. Additionally, we exhibit a constant imbalance on parallel data transfers of between 50 and 60 seconds per workflow execution.

The external load overhead is significant for one and two Grid site executions and decreases with the number of Grid sites. Its severity slightly increases with the Grid size; however, its value decreases and is proportional with the number of activities that execute concurrently on the SGI Altix parallel machines (which is obviously decreasing with the Grid size). The overhead is not simply due to external factors but is also caused by the nature of the NUMA SMP architecture concurrently executing 14 parallel activities. This overhead consists of remote memory accesses and contention on the shared bus, operating system parallel process management, and cache coherency protocols. For compute clusters and workstation networks, the external load is (almost) zero due to the dedicated access to the single processors via the PBS queuing system.

The overheads of the ASKALON middleware services, comprising the Resource Broker, Scheduler, and Performance Prediction, are constant and count for less than 1% each of the entire execution time.

27.8 Conclusions

In contrast to many existing systems, ASKALON supports workflow composition and modeling using the UML standard and provides an XML-based programming interface that shields the application developer from low-level middleware technologies. A separate Resource Manager, which covers both physical resources and workflow activities, renders the boundaries of resource brokerage, virtual organization-wide authorization, and advanced reservation, and provides mechanisms for Grid resource discovery, selection, and allocation along with resource requester and provider interaction. Our Scheduler supports HEFT and genetic search as optimization algorithms which perform significantly better than a pure resource broker, in particular in the case of unbalanced workflows. The Scheduler benefits significantly from a Performance Prediction service that provides expected execution times based on a training phase and statistical methods. The Execution Engine efficiently handles large collections of data dependencies produced by hundreds of parallel activities specific to scientific workflows. We have demonstrated significant performance gains through two checkpointing methods for saving

and restoring the execution of Grid workflows upon engine and application failures.

We have demonstrated the integrated use of the ASKALON services for real-world scientific workflows executed in the Austrian Grid infrastructure. Our future work will focus on further optimization of workflow executions to increase their scalability on the Grid, scheduling based on QoS parameters to be negotiated between the Scheduler and the Resource Manager, and automatic bottleneck detection and steering based on online performance analysis.

27.9 Acknowledgments

Part IV

Future Requirements

Looking into the Future of Workflows: The Challenges Ahead

Ewa Deelman

Contributors: Bruce Berriman, Thomas Fahringer, Dennis Gannon, Carole Goble, Andrew Jones, Miron Livny, Philip Maechling, Steven McGough, Deana Pennington, Matthew Shields, and Ian Taylor

In this chapter, we take a step back from the individual applications and software systems and attempt to categorize the types of issues that we are facing today and the challenges we see ahead. This is by no means a complete picture of the challenges but rather a set of observations about the various aspects of workflow management. In a broad sense, we are organizing our thoughts in terms of the different workflow systems discussed in this book, from the user interface down to the execution environment.

1 User Experience

It is often difficult to provide users with a satisfying experience in building and managing applications, mainly because user expectations with respect to transparency and control vary greatly. Some users may want to describe their problems in a high-level application-specific manner, some may want to view intermediate data, while others may make very detailed plans, including specifying particular resources to use and possibly interacting with the live analysis by suspending and restarting particular portions of the analysis. Thus, workflow requirements are varied, often being subject to user- or domain-specific issues that cannot be satisfied by one system. However, a workflow system needs to be smart enough to handle the low-level technical details behind the scenes, hiding that complexity from scientists while at the same time exposing interfaces to workflow management aspects.

Most of the workflow systems today support a "one-shot" user interaction where, having started the workflow execution, it must continue to completion or error state or be aborted. However, it is often the case that users are not decided on the exact steps in the analysis to be conducted. They may want to use the workflows in an explorative manner, exploring different ideas and avenues of investigation. In order to enable this explorative and

interactive mode, the user must be very much part of the workflow. The systems must provide meaningful information to the user, at an appropriate level of abstraction, and provide adequate user interface responsiveness and system performance to enable the user to interact with the system on a realistic time scale.

Scientific users are often comfortable with their existing methodologies and techniques for conducting their analysis and may resist spending large amounts of time learning new tools and technologies. It would be useful to create an environment where new users can view how other applications have benefited from the technologies. Another benefit of such an approach would be for novice users to be able to view and use the knowledge of domain experts, captured in workflows who have solved the same or similar problems. This type of expert "knowledge capture" is extremely valuable to commercial research institutions where staff may move on and a new employee is expected to take over. Collecting workflows and their components into libraries that can be easily explored, shared between scientists and organizations, and reused will become increasingly important. In some cases new workflows can be generated by finding a workflow that is "close" to the desired analysis and then modifying it to suit the particular needs by substituting different components or data sources. Additionally, demonstrating the usefulness of the workflow technologies in a variety of applications and scenarios would enable other scientists to leverage existing experiences.

Result validation and verification is always uppermost in a scientist's mind, often the journey to the result is as important, if not more important, than the result itself. Reproducibility is vital for the scientific process. To be able to validate a given set of results, we must be able to take the original workflow and start data and rerun the execution to give the same results. Thus, it is important to provide detailed provenance about every step of the workflow process, even to the level of the execution environment. Each of the components or steps in the workflow must also be validated to ensure that each individual result for each component is accurate. Finally, to truly be able to verify and reproduce experiments accurately since all aspects of the system are software-based, we must have version information to ensure that when an experiment is rerun everything is as it was. Are the start data the exact version that the original experiment used? Do we have the same versions of all of the components and the execution environment, or have they been modified? Even if modifications do not affect the results, we must have information about the system versions and be able to prove that this is the case. Some aspects of the extremely complicated systems that make up modern workflow environments are very difficult to version accurately. For example, if we rely on external services such as Web services as components, what information do we have access to about the version of the service we are using from one instance to the next? Standard Web Services Description Language (WSDL) has no capability for representing version information, and

even if it did, if the service is controlled by a third party to what level do we trust any information we may get about the version of the service?

There must also be an infrastructure that can catalog the provenance information in a scalable way and provide means of efficiently searching the large volumes of information. Provenance also needs to be structured in a way that would enable a scientist to easily evaluate the validity of the results. For example, it may not be necessary to provide detailed execution records when a scientist wants to find out about the types of analysis used in the workflow, but it would be if the same scientist wanted to reproduce an experiment from a workflow. While some scientific workflow systems already provide detailed provenance information, the problem of providing a standard representation is not solved. Solving this is necessary if a provenance generated by one workflow system is to be replayed on another. The other vital area in provenance that is understood but not necessarily implemented everywhere, and certainly not in Web services, is all the aspects of workflow system versioning.

Today, various aspects of the user experience are being partially implemented in a variety of workflow systems. However, there is no single system that provides all the necessary ingredients for comprehensive, flexible, and scientifically rigorous experimentation.

2 Workflow Languages and Representations

An aspect not addressed in the user experience is the language used to encode scientific workflows. In some cases, it is graphical, and in others it is script-based. In all cases, the language needs to provide the users with easy ways of specifying the required steps in analysis tasks and a means of connecting them either with a flow of control or data. As mentioned above, given the differences between the types of users, developing a standard workflow language is very challenging. The issue remains whether the cognitive overhead involved in creating workflows may distract scientists from creative exploration.

Although a plethora of tools, GUIs, and paradigms are currently used, in practice many suffer from the drawback that they are too low-level and do not shield the programmer from underlying systems. In other cases, expressiveness is too limited to describe all the needed control and data flow. For example, very few graphical systems support exception handling or other forms of dynamic, adaptive behavior. As with the world of programming languages, there can be no standard form of expression: Different users will always need different ways of describing computations. It is possible that a common intermediate form may exist. Based on such a common intermediary, the wide variety of workflow-related tools could have a chance of becoming interoperable and some of the existing duplication of effort could be eliminated.

In terms of "visual editing" of workflows, much work still needs to be done. Current workflows range from those that have a few tasks executed by a few

services to those that are composed of thousands of tasks distributed over thousands of processors. Many of the editors existing today can be awkward to apply in a distributed setting. Thus, developing compact and meaningful visualizations is an important challenge.

Workflow representations need not only provide a way to describe a workflow but also support the transitions between the different levels of abstraction from high-level user descriptions down to low-level execution details. One example of the information that needs to be captured by a workflow representation is the performance requirements necessary to map a workflow to an executable form.

One also has to be careful not to take workflow languages to the extreme and turn them into full-featured programming or scripting languages since they already exist in abundance and are inadequate for scientists to use on a daily basis. Workflow languages need to capture the salient features of a scientific analysis without providing so much flexibility as to make the workflow composition process too complex.

One possible solution to this problem would be to develop a series of languages that can be mapped from one to the other, where we have different languages that are appropriate in different contexts—different levels of abstraction. Users could then enter the system at their appropriate level.

Using a common intermediate representation would be one approach. This could be augmented with a common runtime and standard workflow enactment engine. In a manner analogous to the Microsoft Common Language Runtime and Infrastructure (CLR), one could integrate small scripts as executing components within a larger workflow. Within the Web services community, especially for business interactions, BPEL is already becoming the de facto standard for service orchestration and workflow. It is one possible candidate for a common intermediate representation for e-Science workflows as well, although there are issues with this approach.

3 Workflow Compilers

Workflow compilers can be used as a mapping tool between workflow languages at different levels of abstraction. They allow scientists to express their analysis at any level of abstraction and then compile it to the target execution system, which can range from a single host or service to a distributed, heterogeneous set of resources and services.

Compiling a workflow down to an executable form requires knowledge about the requirements and performance characteristics of the workflow tasks and knowledge of the availability and the characteristics of the resources. Currently, this knowledge is rather limited and often encoded in an ad hoc manner. A challenge for the future would be to capture the application-level and the execution-level knowledge using semantic representations and employ reasoners to find suitable mappings.

The compilation process involves many decisions, for example, why particular resources were selected over others. It may be beneficial to encode some of the decision process as the workflow is being mapped. In fact, the dynamic nature of resource availability may make this late binding necessary. This would enable more efficient compilation and possibly a more rich interaction between the workflow compiler and the workflow executors.

Considering that e-Science workflows are often mapped to a set of heterogeneous, distributed resources, failures in execution are commonplace. This failure-prone environment poses a significant challenge to the workflow compilers. Ultimately, the compilers should anticipate failures and plan accordingly, possibly producing "plan B," or backtracking. They should also work closely with the workflow engines to react to problems as they occur.

Compilers also need to support the mapping of information about the execution of the workflow components back to the high-level descriptions, for example, in order to provide user-level monitoring and failure information.

As we mentioned before, the management of metadata and provenance at every step of the workflow is crucial. Compilers can be very beneficial in this aspect as they can augment the executable workflows with metadata and provenance management tasks, for example, adding tasks for collecting execution statistics and tasks for storing them in relevant databases. However, the compiler cannot manage the metadata and provenance alone. It needs appropriate workflow representations to support annotations of the workflow products with relevant metadata.

4 Workflow Enactors or Executors

The main job of a workflow executor, or workflow engine, is to faithfully and robustly execute the workflows. However, current enactment engines are not as fault tolerant as we would like, and many application and system faults still occurr. Many lightweight systems embed the enactor directly into the workflow composer: If the user turns off his laptop, the workflow will stop. Others, such as those based around the BPEL specification, are designed to allow the entire workflow state to be made persistent in a database. Consequently, a workflow enactment can survive a reboot of the engine.

Today there are many workflow engines, as there are many workflow compilers and user environments. It would potentially be beneficial to have a common engine, or at least a limited set of engines, for execution in distributed environments such as the Grid. Again, workflow language standardization, at various levels of abstraction, would be of great benefit in developing common engines.

An important challenge for workflow engines is to detect when a failure in the environment is a mask, which needs to be passed to the compiler and perhaps in turn to the user. Clearly, the workflow executor needs to provide

enough information at an appropriate level of abstraction to enable this type of failure handling.

Another issue not addressed fully by workflow executors today is the management of dynamic workflows, where new portions of a workflow can be added at any time while some other portions are cancelled. A related problem stems from the amounts of data involved in the e-Science workflows. Modest-sized workflows can create gigabytes of data on the execution sites. These data, once they are successfully transferred to permanent storage, should be removed, unless of course they are needed by subsequent analysis.

5 Debugging

As we mentioned before, errors often occur and need to be dealt with either by the workflow engine, the workflow compiler, or the user. Today, a user often has to examine logs provided by the workflow management system, which are mostly too low-level to be comprehensible by an average user. Much of the complexity stems from the cryptic error messages generated by the underlying distributed execution environment. However, some progress at the workflow level could be made as well. For example, it would be beneficial to provide the capability of replaying arbitrary portions of the workflow while modifying the data sources, the execution systems, and workflow components. This may provide the users with some insight into the nature of the failures.

In the distributed case, the most common approach to this is to provide a global event notification system. Such a system can also be tied to the provenance tracking, and the event history can be of great value in the replay process. However, a larger challenge is managing all the intermediate data products of the workflow. These are needed if a workflow is to be interrupted and restarted without redoing all previous work. Again, in the distributed case, this requires a distributed virtual data management system. Every intermediate data product needs to have a unique identifier that can be used to access that object if it is needed again.

6 Execution Environments

Much work needs to be done in terms of the distributed execution environment. Reliability is of paramount importance, as is providing detailed yet meaningful information about failures when they occur. As more scientists depend on large-scale distributed systems to do their work, these systems need to provide production-level availability and reliability.

Much work also needs to be done in characterizing the execution system so that workflow services can make meaningful decisions. This includes not only characterizing computational resources but also storage. Currently, we

don't distinguish between different types of storage such as fast I/O storage, long-term storage, quotas associated with specific resources, etc.

Usage policies of the resources often are not exposed in a way that can be easily examined by workflow management software. As a result, computations may be sent to resources with little chance of successful execution. Consequently, there is a need for dynamic resource-level authorization and policy negotiation. One approach is to associate the identity of the owner of a workflow enactment with the instance of that enactment. The workflow engine can negotiate with resources at runtime to decide on the best resources that the user is authorized to use.

Monitoring tools are critical in a distributed environment. They must be scalable and include meaningful, up-to-date information. Many efforts have gone into coming up with common schemas for representing sets of resources. However, because many execution environments today are managed by different organizations and projects, there needs to be a way to monitor across the organizational boundaries. Possibly, semantic technologies may help match seemingly disparate information.

Finally, it is important for the workflow software to be easy to deploy and manage, ultimately supporting on-the-fly deployment so as to make full use of the dynamic execution environment.

7 The Big Question

Is the workflow metaphor too restrictive for exploratory e-Science? We think not. As we have seen, there are a number of approaches within the workflow arena, but clearly the authors of this book believe that workflow is the correct approach for e-Science applications. Are they right? Well, only time and experience will give us the answer to that question.

References

1. The 2MASS Project. `http://www.ipac.caltech.edu/2mass`.
2. B. Abbott et al. Analysis of LIGO Data for Gravitational Waves from Binary Neutron Stars. *Physical Review D*, 69:122001, 2004.
3. B. Abbott et al. Search for Gravitational Waves from Galactic and Extragalactic Binary Neutron Stars. *Physical Review D*, 72:082001, 2005.
4. B. Abbott et al. Search for Gravitational Waves from Primordial Black Hole Binary Coalescences in the Galactic Halo. *Physical Review D*, 72:082002, 2005.
5. B. Abbott et al. Search for Gravitational Waves from Binary Black Hole Inspirals in LIGO Data. *Physical Review D*, 73:062001, 2006.
6. A. Abramovici, W. E. Althouse, R. W. P. Drever, Y. Gursel, S. Kawamura, F. J. Raab, D. Shoemaker, L. Sievers, R. E. Spero, and K. S. Thorne. LIGO — The Laser Interferometer Gravitational-Wave Observatory. *Science*, 256:325–333, Apr. 1992.
7. M. Addis, J. Ferris, M. Greenwood, D. Marvin, P. Li, T. Oinn, and A. Wipat. Experiences with e-Science Workflow Specification and Enactment in Bioinformatics. In *Proceedings of UK e-Science All Hands Meeting*, pages 459–467, 2003.
8. A. Ali, O. Rana, and I. Taylor. Web Services Composition for Distributed Data Mining. In *ICPP 2005 Workshops, International Conference Workshops on Parallel Processing*, pages 11–18. IEEE, New York, 2005.
9. W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke. Data Management and Transfer in High-Performance Computational Grid Environments. *Parallel Computing*, 28(5):749–771, 2002.
10. B. Allen. A chi^2 Time-Frequency Discriminator for Gravitational Wave Detection. *Physical Review D*, 71:062001, 2005.
11. B. Allen, W. G. Anderson, P. R. Brady, D. A. Brown, and J. D. E. Creighton. FINDCHIRP: An Algorithm for Detection of Gravitational Waves from Inspiraling Compact Binaries. *Submitted to Physical Review D*, 2005.
12. G. Allen, D. Angulo, I. Foster, G. Lanfermann, C. Liu, T. Radke, E. Seidel, and J. Shalf. The Cactus Worm: Experiments with Dynamic Resource Discovery and Allocation in a Grid Environment. *International Journal of High Performance Computing Applications*, 15(4):345–358, 2001. `http://www.cactuscode.org/Papers/IJSA_2001.pdf`.

13. G. Allen, D. Angulo, T. Goodale, T. Kielmann, A. Merzky, J. Nabrzysky, J. Pukacki, M. Russell, T. Radke, E. Seidel, J. Shalf, and I. Taylor. GridLab: Enabling Applications on the Grid. In M. Parashar, editor, *GRID 2002, 3rd International Workshop on Grid Computing*, volume 2536 of *Lecture Notes in Computer Science*, pages 39–45. Springer-Verlag, New York, 2002.

14. G. Allen, W. Benger, T. Dramlitsch, T. Goodale, H. Hege, G. Lanfermann, A. Merzky, T. Radke, and E. Seidel. Cactus Grid Computing: Review of Current Development. In R. Sakellariou, J. Keane, J. Gurd, and L. Freeman, editors, *Europar 2001: Parallel Processing, Proceedings of 7th International Conference, Manchester, UK*. Springer, New York, August 2001. `http://www.cactuscode.org/Papers/Europar01.ps.gz`.

15. G. Allen, T. Goodale, G. Lanfermann, T. Radke, D. Rideout, and J. Thornburg. *Cactus Users Guide*, 2004. `http://www.cactuscode.org/Guides/Stable/UsersGuide/UsersGuideStable.pdf`.

16. M. Alpdemir, A. Mukherjee, A. Gounaris, N. Paton, A. Fernandes, R. Sakellariou, P. Watson, and P. Li. Using OGSA-DQP to Support Scientific Applications for the Grid. In *Proceedings of the First International Workshop on Scientific Applications in Grid Computing (SAG'04)*, volume 3458 of *Lecture Notes in Computer Science*, pages 13–24. Springer-Verlag, Berlin, 2004. Invited paper.

17. M. Alt, H. Bischof, and S. Gorlatch. Algorithm Design and Performance Prediction in a Java-Based Grid System with Skeletons. In B. Monien and R. Feldmann, editors, *Euro-Par 2002*, volume 2400 of *Lecture Notes in Computer Science*, pages 899–906. Springer-Verlag, Berlin, 2002.

18. M. Alt, A. Hoheisel, H.-W. Pohl, and S. Gorlatch. A Grid Workflow Language Using High-Level Petri Nets. In R. Wyrzykowski, J. Dongarra, N. Meyer, and J. Wasniewski, editors, *Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics PPAM'2005*, volume 3911 of *Lecture Notes in Computer Science*, pages 715–722. Springer, New York, 2006.

19. I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludäscher, and S. Mock. Kepler: An Extensible System for Design and Execution of Scientific Workflows. In *16th International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 423–424. IEEE Computer Society, New York, 2004.

20. I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludäscher, and S. Mock. Kepler: Towards a Grid-Enabled System for Scientific Workflows. In *Proceedings of the Workflow in Grid Systems Workshop at the Global Grid Forum (GGF10)*. Global Grid Forum, 2004.

21. AMD. `http://www.amd.com`.

22. K. Amin, G. von Laszewski, M. Hategan, N. J. Zaluzec, S. Hampton, and A. Rossi. GridAnt: A Client-Controllable Grid Workflow System. In *HICSS '04: Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) — Track 7*, page 70210.3. IEEE Computer Society, Washington, 2004.

23. R. P. Anderson, D. Lew, and A. T. Peterson. Evaluating Predictive Models of Species' Distributions: Criteria for Selecting Optimal Models. *Ecological Modelling*, 162:211–232, 2003.

24. T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services Version 1.1.

25. Apache Derby. `http://db.apache.org/derby/`.
26. Apples with Apples: Numerical Relativity Comparisons and Tests. See Web site at `http://www.ApplesWithApples.org`.
27. M. B. Araujo, M. Cabeza, W. Thuiller, L. Hannah, and P. H. Williams. Would Climate Change Drive Species out of Reserves? An Assessment of Existing Reserve-Selection Methods. *Global Change Biology*, 10(9):1618–1626, 2004.
28. M. B. Araujo, R. G. Pearson, W. Thuiller, and M. Erhard. Validation of Species-Climate Impact Models under Climate Change. *Global Change Biology*, 11(9):1504–1513, 2005.
29. A. Arbree, P. Avery, D. Bourilkov, R. Cavanaugh, S. Katageri, J. Rodriguez, G. Graham, J. Vöckler, and M. Wilde. Virtual Data in CMS Productions. In *Proceedings of Computing in High Energy and Nuclear Physics*, volume eConf C0303241. Electronic Conference Proceedings Archive, 2003.
30. R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a Common Component Architecture for High-Performance Scientific Computing. In *Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC-8 '99)*, page 13. IEEE Computer Society, New York, 1999.
31. D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Seymour, K. Sagi, Z. Shi, and S. Vadhiyar. Users' Guide to NetSolve V1.4.1. Technical Report ICL-UT-02-05, University of Tennessee, Knoxville, 2002.
32. Astrophysics Simulation Collaboratory (ASC) home page. `http://www.ascportal.org`.
33. The Austrian Grid Consortium. `http://www.austriangrid.at`.
34. The Avalon Project. `http://avalon.apache.org`.
35. K. Ballinger, D. Ehnebuske, M. Gudgin, M. Nottingham, and P. Yendluri. WS-I Basic Profile Version 1.0. `http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html`, April 2004.
36. M. Bardeen, E. Gilbert, T. Jordan, P. Nepywoda, E. Quigg, M. Wilde, and Y. Zhao. The QuarkNet/Grid Collaborative Learning e-Lab. In *Workshop on Collaborative and Learning Applications of Grid Technology and Grid Education, CCGrid 2005*, volume 1, pages 27–34. IEEE Computer Society, New York, 2005.
37. B. C. Barish and R. Weiss. LIGO and the Detection of Gravitational Waves. *Physics Today*, 52(10):44–50, October 1999.
38. P. Barman, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *SOSP '03: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 164–177. ACM Press, New York, September 2003.
39. J. E. Barnes and P. Hut. A Hierarchical $O(N \log N)$ Force-Calculation Algorithm. *Nature*, 324(4):446–449, 1986.
40. C. Baru, A. Gupta, B. Ludäscher, R. Marciano, Y. Papakonstantinou, P. Velikhov, and V. Chu. XML-Based Information Mediation with MIX. In *SIGMOD '99: Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 597–599. ACM Press, New York, 1999.
41. C. Baru, R. Moore, A. Rajasekar, and M. Wan. The SDSC Storage Resource Broker. In *CASCON '98: Proceedings of the 1998 Conference of the Centre for Advanced Studies on Collaborative Research*, page 5. IBM Press, New York, 1998.

42. M. Beckerle and M. Westhead. GGF DFDL Primer. Technical report, Global Grid Forum, 2004.

43. D. J. Beerling, B. Huntley, and J. P. Bailey. Climate and the Distribution of Fallopia Japonica: Use of an Introduced Species to Test the Predictive Capacity of Response Surfaces. *Journal of Vegetation Science*, 6:269–282, 1995.

44. R. A. Benjamin, E. Churchwell, B. L. Babler, R. Indebetouw, M. R. Meade, B. A. Whitney, C. Watson, M. G. Wolfire, M. J. Wolff, R. Ignace, T. M. Bania, S. Bracker, D. P. Clemens, L. Chomiuk, M. Cohen, J. M. Dickey, J. M. Jackson, H. A. Kobulnicky, E. P. Mercer, J. S. Mathis, S. R. Stolovy, and B. Uzpen. First GLIMPSE Results on the Stellar Structure of the Galaxy. *The Astrophysical Journal*, 600:L149–L152, 2005.

45. J. Bent, V. Venkataramani, N. LeRoy, A. Roy, J. Stanley, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and M. Livny. NeST — A Grid Enabled Storage Appliance. In J. Weglarz, J. Nabrzyski, J. Schopf, and M. Stroinkski, editors, *Grid Resource Management*, volume 64 of *International Series in Operations Research & Management Science*. Kluwer Academic Publishers, Dordrecht, 2003.

46. D. Berkley, S. Bowers, M. Jones, B. Ludäscher, M. Schildhauer, and J. Tao. Incorporating Semantics in Scientific Workflow Authoring. In *SSDBM'2005: Proceedings of the 17th International Conference on Scientific and Statistical Database Management*, pages 75–78. Lawrence Berkeley Laboratory, Berkeley, 2005.

47. L. Bernardinello and F. de Cindio. A Survey of Basic Net Models and Modular Net Classes. In *Advances in Petri Nets 1992, The DEMON Project*, volume 609 of *Lecture Notes in Computer Science*, pages 304–351. Springer-Verlag, New York, 1992.

48. G. Berriman, D. Curkendall, J. Good, J. Jacob, D. Katz, M. Kong, S. Monkewitz, R. Moore, T. Prince, and R. Williams. An Architecture for Access to a Compute Intensive Image Mosaic Service in the NVO. In A. S. Szalay, editor, *Virtual Observatories, Proceedings of The International Society for Optical Engineering*, volume 4686, pages 91–102. SPIE Press, Bellingham WA, 2002.

49. M. Bevers, J. Hof, D. W. Uresk, and G. L. Schenbeck. Spatial Optimization of Prairie Dog Colonies for Black-Footed Ferret Recovery. *Operations Research*, 45:495–507, 1997.

50. V. Bhat and M. Parashar. Discover Middleware Substrate for Integrating Services on the Grid. In *High Performance Computing - HiPC 2003*, volume 2913 of *Lecture Notes in Computer Science*, pages 373–382. Springer, Berlin, 2003.

51. D. Bhatia, V. Burzevski, M. Camuseva, W. F. G. Fox, and G. Premchandra. WebFlow: A Visual Programming Paradigm for Web/Java Based Coarse Grain Distributed Computing. *Concurrency and Computation: Practice and Experience*, 9(6):555–577, 1997.

52. P. Blaha, K. Schwarz, G. Madsen, D. Kvasnicka, and J. Luitz. *WIEN2k: An Augmented Plane Wave plus Local Orbitals Program for Calculating Crystal Properties*. Institute of Physical and Theoretical Chemistry, Technische Universität, Wien, 2001.

53. Basic Local Alignment Search Tool (BLAST). `http://www.ncbi.nlm.nih.gov/blast/`, 2006.

54. 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 *IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2005)*, volume 2, pages 759–767. IEEE Computer Society, New York, 2005.

55. Biological Magnetic Resonance Data Bank. `http://www.bmrb.wisc.edu/`, 2006.

56. L. Bocchi, C. Laneve, and G. Zavattaro. A Calculus for Long Running Transactions. In E. Najm, U. Nestmann, and P. Stevens, editors, *FMOODS 2003: Formal Methods for Open Object-Based Distributed Systems*, volume 2884 of *Lecture Notes in Computer Science*, pages 124–138. Springer-Verlag, Berlin, 2003.

57. R. Bondarescu, G. Allen, G. Daues, I. Kelley, M. Russell, E. Seidel, J. Shalf, and M. Tobias. The Astrophysics Simulation Collaboratory Portal: a Framework for Effective Distributed Research. *Future Generation Computer Systems*, 21(2):259–270, 2005.

58. D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard. Web Services Architecture, W3C Working Group Note. `http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/`, November 2004.

59. S. Bowers and B. Ludäscher. An Ontology-Driven Framework for Data Transformation in Scientific Workflows. In *International Workshop on Data Integration in the Life Sciences (DILS'04)*, volume 2994 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, Berlin, 2004.

60. S. Bowers, D. Thau, R. Williams, and B. Ludäscher. Data Procurement for Enabling Scientific Workflows: On Exploring Inter-Ant Parasitism. In C. Bussler, V. Tannen, and I. Fundulaki, editors, *Semantic Web and Databases, Second International Workshop, SWDB 2004*, volume 3372 of *Lecture Notes in Computer Science*, pages 57–63. Springer-Verlag, Berlin, 2005.

61. D. Box. *Essential COM*. Addison-Wesley, Reading, MA, 1997.

62. D. Box, L. F. Cabrera, C. Critchley, F. Curbera, D. Ferguson, A. Geller, S. Graham, D. Hull, G. Kakivaya, A. Lewis, B. Lovering, M. Mihic, P. Niblett, D. Orchard, J. Saiyed, S. Samdarshi, J. Schlimmer, I. Sedukhin, J. Shewchuk, B. Smith, S. Weerawarana, and D. Wortendyke. Web Services Eventing (WS-Eventing). Technical report, W3C, August 2004.

63. F. Breg, S.Diwan, J. Villacis, J. Balasubramanian, E. Akman, and D. Gannon. Java RMI Performance and Object Model Interoperability: Experiments with Java/HPC++ Distributed Components. *Concurrency Practice and Experience, Special Issue from the Fourth Java for Scientific Computing Workshop*, 10(11–13):941–955, 1998.

64. D. Brookshier, D. Govoni, N. Krishnan, and J. C. Soto. *JXTA: Java P2P Programming*. Sams Publishing, Indianapolis, 2002.

65. D. A. Brown. Using the INSPIRAL Program to Search for Gravitational Waves from Low-Mass Binary Inspiral. *Classical and Quantum Gravity*, 22:S1097–S1108, 2005.

66. M. Brune, G. Fagg, and M. Resch. Message Passing Environments for Metacomputing. *Future Generation Computer Systems*, 15(5–6):699–712, 1999.

67. D. Bunting, M. Chapman, O. Hurley, M. Little (editor), J. Mischkinsky, E. Newcomer (editor), J. Webber, and K. Swenson. Web Services Context (WS-Context) Ver1.0. `http://www.arjuna.com/library/specs/ws_caf_1-0/WS-CTX.pdf`, 2003.

68. A. Buonanno, Y. Chen, and M. Vallisneri. Detecting Gravitational Waves from Precessing Binaries of Spinning Compact Objects: Adiabatic Limit. *Physical Review D*, 67:104025, 2003.

69. A. Buonanno, Y. Chen, and M. Vallisneri. Detection Template Families for Gravitational Waves from the Final Stages of Binary-Black-Hole Inspirals: Nonspinning Case. *Physical Review D*, 67:024016, 2003.

70. C. E. Burns, K. M. Johnston, and O. J. Schmitz. Global Climate Change and Mammalian Species Diversity in US National Parks. *Proceedings of the National Academy of Sciences of the United States of America*, 100(20):11474–11477, 2003.

71. M. Butler, R. Pennington, and J. A. Terstriep. Mass Storage at NCSA: SGI DMF and HP UniTree. In *Proceedings of 40th Cray User Group Conference*. CD Rom Proceedings, 1998.

72. R. Butler, D. Engert, I. Foster, C. Kesselman, S. Tuecke, J. Volmer, and V. Welch. A National-Scale Authentication Infrastructure. *IEEE Computer*, 33(12):60–66, 2000.

73. The Cactus Framework. See Web site at `http://www.cactuscode.org`.

74. K. Camarda, Y. He, and K. A. Bishop. A Parallel Chemical Reactor Simulation using Cactus. In *Proceedings of Linux Clusters: The HPC Revolution, NCSA*. Linux Clusters Institute, 2001.

75. F. Camilo, D. Lorimer, P. Freire, A. Lyne, and R. Manchester. Observations of 20 Millisecond Pulsars in 47 Tucanae at 20 cm. *The Astrophysical Journal*, 535:975, 2000.

76. J. Cao, S. A. Jarvis, S. Saini, and G. R. Nudd. GridFlow: Workflow Management for Grid Computing. In *3rd International Symposium on Cluster Computing and the Grid*, page 198. IEEE Computer Society Press, New York, 2003.

77. E. Caron, B. Del-Fabbro, F. Desprez, E. Jeannot, and J.-M. Nicod. Managing Data Persistence in Network Enabled Servers. *Scientific Programming Journal*, 13(4):333–354, 2005.

78. E. Caron, F. Desprez, F. Lombard, J.-M. Nicod, M. Quinson, and F. Suter. A Scalable Approach to Network Enabled Servers. In *8th International EuroPar Conference*, volume 2400 of *Lecture Notes in Computer Science*, pages 907–910. Springer-Verlag, Berlin, 2002.

79. G. Carpenter, A. N. Gillison, and J. Winter. DOMAIN: A Flexible Modeling Procedure for Mapping Potential Distributions of Animals and Plants. *Biodiversity and Conservation*, 2:667–680, 1993.

80. Fixed Mesh Refinement with Carpet. `http://www.carpetcode.org/`.

81. H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for Scheduling Parameter Sweep Applications in Grid Environments. In *HCW '00: Proceedings of the 9th Heterogeneous Computing Workshop*, pages 349–363. IEEE Computer Society, New York, 2000.

82. CERN Advanced Storage Manager. `http://castor.web.cern.ch/castor/`, 2005.

83. Convention on Biodiversity Article 2 (Rio Earth Summit). `http://www.biodiv.org/convention/default.shtml`, 1992.

84. CCA Forum. The Common Component Architecture Technical Specification — version 0.5. Technical report, Common Component Architecture Forum, 2001.

85. Center for Computation and Technology at Louisiana State University. See `http://www.cct.lsu.edu/`.

86. CDDLM Working Group, GGF. `https://forge.gridforum.org/projects/cddlm-wg`.

87. L. Chen, S. J. Cox, F. Tao, N. R. Shadbolt, C. Puleston, and C. Goble. Empower Resource Providers to Build the Semantic Grid. In *IEEE/WIC/ACM International Conference on Web Intelligence (WI'04)*, pages 271–277. IEEE Computer Society, New York, 2004.

88. 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 *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–17. IEEE Computer Society Press, New York, 2002.

89. J. Chin, P. V. Coveney, and J. Harting. The TeraGyroid Project: Collaborative Steering and Visualisation in an HPC Grid for Modelling Complex Fluids. *UK All-hands e-Science Conference*, 2004.

90. J. Chin, J. Harting, S. Jha, P. Coveney, A. R. Porter, and S. M. Pickles. Steering in Computational Science: Mesoscale Modelling and Simulation. *Contemporary Physics*, 44:417–434, 2003.

91. D. Churches, G. Gombas, A. Harrison, J. Maassen, C. Robinson, M. Shields, I. Taylor, and I. Wang. Programming Scientific and Distributed Workflow with Triana Services. *Concurrency and Computation: Practice and Experience (Special Issue: Workflow in Grid Systems)*, 18(10):1021–1037, 2006.

92. E. Churchwell, B. A. Whitney, B. L. Babler, R. Indebetouw, M. R. Meade, C. Watson, M. J. Wolff, M. G. Wolfire, T. M. Bania, R. A. Benjamin, D. P. Clemens, M. Cohen, K. E. Devine, J. M. Dickey, F. Heitsch, J. M. Jackson, H. A. Kobulnicky, A. P. Marston, J. S. Mathis, E. P. Mercer, J. R. Stauffer, and S. R. Stolovy. RCW 49 at Mid-Infrared Wavelengths: A GLIMPSE from the Spitzer Space Telescope. *The Astrophysical Journal Supplement Series*, 154:322–327, 2004.

93. T. Clark, S. Martin, and T. Liefeld. Globally Distributed Object Identification for Biological Knowledgebases. *Briefings in Bioinformatics*, 5(1):59–70, 2004.

94. J. Cohen, N. Furmento, G. Kong, A. Mayer, S. Newhouse, and J. Darlington. RealityGrid: An Integrated Approach to Middleware through ICENI. *Royal Society of London Philosophical Transactions Series A*, 363(1833):1817–1827, 2005.

95. J. Cohen, W. Lee, A. Mayer, and S. Newhouse. Making the Grid Pay — Economic Web Services. In *Building Service Based Grids Workshop, GGF11*. Global Grid Forum, June 2004.

96. Condor Glidein. `http://www.cs.wisc.edu/condor/glidein`.

97. Condor Team. DAGMan: A Directed Acyclic Graph Manager, July 2005. `http://www.cs.wisc.edu/condor/dagman/`.

98. The CORBA Component Model. `http://www.omg.org/technology/documents/formal/components.htm`.

99. J. Costa, A. T. Peterson, and C. B. Beard. Ecological Niche Modeling and Differentiation of Populations of *Triatoma Brasiliensis* Neiva, 1911, the Most Important Chagas' Disease Vector in Northeastern Brazil (Hemiptera, Reduviidae, Triatominae). *American Journal of Tropical Medicine & Hygiene*, 67(5):516–520, 2002.

100. K. Czajkowski, D. F. Ferguson, I. Foster, J. Frey, S. Graham, I. Sedukhin, D. Snelling, S. Tuecke, and W. Vambenepe. The WS-Resource Framework. Technical report, The Globus Alliance, 2004.

101. K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid Information Services for Distributed Resource Sharing. In *HPDC '01: Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10'01)*, pages 181–184. IEEE Computer Society, Washington, 2001.

102. K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A Resource Management Architecture for Metacomputing Systems. In D. G. Feitelson and L. Rudolph, editors, *IPPS/SPDP '98: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pages 62–82. Springer Verlag, London, 1998.

103. H. Daily, H. Casanovay, and F. Berman. A Decoupled Scheduling Approach for the GrADS Program Development Environment. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–14. IEEE Computer Society Press, Los Alamitos, 2002.

104. J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, and Q. Wu. Parallel Programming Using Skeleton Functions. In *PARLE '93: Proceedings of the 5th International PARLE Conference on Parallel Architectures and Languages Europe*, volume 694 of *Lecture Notes in Computer Science*, pages 146–160. Springer Verlag, London, 1993.

105. Distributed Audio Rendering and Retrieval using Triana (DARRT). `http://www.lcat.lsu.edu/projects.php#DARRT`.

106. S. Das, A. McGough, J. Cohen, and J. Darlington. Lightweight Solution for Protein Annotation. In S. J. Cox and D. W. Walker, editors, *UK e-Science All Hands Meeting, 2005*, pages 396–402, Nottingham, UK, 2005. CD Rom Proceedings.

107. Data Mining Tools and Services for Grid Computing Environments (DataMiningGrid). `http://www.datamininggrid.org/`.

108. E. R. Davis and J. Caron. THREDDS: A Geophysical Data/Metadata Framework. In *Proceedings of the 18th International Conference on Interactive Information Processing Systems (IIPS) for Meteorology, Oceanography, and Hydrology*, pages 52–53. American Meteorological Society, Boston, January 2002.

109. dCache.org. `http://www.dcache.org/`, 2006.

110. E. Deelman, J. Blythe, Y. Gil, and C. Kesselman. Workflow Management in GriPhyN. In J. Nabrzyski, J. Schopf, and J. Weglarz, editors, *Grid Resource Management*, volume 64 of *International Series in Operations Research & Management Science*. Kluwer Academic Publishers, Dordrecht, 2003.

111. E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, and M. Livny. Pegasus: Mapping Scientific Workflows onto the Grid. In *AxGrids 2004: 2nd European Across Grids Conference*, volume 3165 of *Lecture Notes in Computer Science*, pages 11–20. Springer, Berlin, 2004.

112. E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, K. Blackburn, A. Lazzarini, A. Arbree, R. Cavanaugh, and S. Koranda. Mapping Abstract Complex Workflows onto Grid Environments. *Journal of Grid Computing*, 1(1):25–39, 2003.

113. E. Deelman, C. Kesselman, G. Mehta, L. Meshkat, L. Pearlman, K. Blackburn, P. Ehrens, A. Lazzarini, R. Williams, and S. Koranda. GriPhyN and LIGO, Building a Virtual Data Grid for Gravitational Wave Scientists. In *HPDC '02: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing HPDC-11 20002 (HPDC'02)*, pages 225–236. IEEE Computer Society, Washington, 2002.

114. E. Deelman, G. Mehta, and C. Kesselman. Transformation Catalog Design for GriPhyN. Technical Report GriPhyN-2001-17, University of Southern California, Information Sciences Institute (ISI), 2001.

115. E. Deelman, R. Plante, C. Kesselman, G. Singh, M.-H. Su, G. Greene, R. Hanisch, N. Gaffney, A. Volpicelli, J. Annis, V. Sekhri, T. Budavari, M. A. Nieto-Santisteban, W. O'Mullane, D. Bohlender, T. McGlynn, A. H. Rots, and O. Pevunova. Grid-Based Galaxy Morphology Analysis for the National Virtual Observatory. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 47. IEEE Computer Society, Washington, 2003.

116. E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. Katz. Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems. *Scientific Programming Journal*, 13(3):219–237, 2005.

117. T. Delaitre, T. Kiss, A. Goyeneche, G. Terstyanszk, S. Winter, and P. Kacsuk. GEMLCA: Running Legacy Code Applications as Grid Services. *Journal of Grid Computing*, 3(1–2):75–90, 2005.

118. F. Dijkstra and A. van der Steen. Integration of Two Ocean Models within Cactus. *Concurrency and Computation: Practice and Experience (Special Issue: Computational Frameworks)*, 18(2):193–202, 2005.

119. Environment for Industrial Design Optimisation (DIPSO). `http://www.wesc.ac.uk/projects/dipso/index.html`.

120. DOE Grids Certificate Authority. See `http://www.doegrids.org/`.

121. K. Droegemeier. The Ability of CASA Doppler Radars to Observe Tornadoes: An Assessment Using Tornado Damage Path Width Climatology. In *9th Symposium on Integrated Observing and Assimilation Systems for the Atmosphere, Oceans, and Land Surface*. American Meteorological Society, Boston, 2005.

122. K. K. Droegemeier, V. Chandrasekar, R. Clark, D. Gannon, S. Graves, E. Joseph, M. Ramamurthy, R. Wilhelmson, K. Brewster, B. Domenico, T. Leyton, V. Morris, D. Murray, B. Plale, R. Ramachandran, D. Reed, J. Rushing, D. Weber, A. Wilson, M. Xue, and S. Yalda. Linked Environments for Atmospheric Discovery (LEAD): Architecture, Technology Roadmap and Deployment Strategy. In *21st International Conference on Interactive Information Processing Systems (IIPS) for Meteorology, Oceanography, and Hydrology*. American Meteorological Society, Boston, 2005.

123. K. K. Droegemeier, D. Gannon, D. Reed, B. Plale, J. Alameda, T. Baltzer, K. Brewster, R. Clark, B. Domenico, S. Graves, E. Joseph, D. Murray, R. Ramachandran, M. Ramamurthy, L. Ramakrishnan, J. A. Rushing, D. Weber, R. Wilhelmson, A. Wilson, M. Xue, and S. Yalda. Service-Oriented Environments for Dynamically Interacting with Mesoscale Weather. *Computing in Science and Engineering*, 7(6):12–29, 2005.

124. M. J. Duftler, N. K. Mukhi, A. Slominski, and S. Weerawarana. Web Services Invocation Framework (WSIF). In *OOPSLA 2001 Workshop on Object-Oriented Web Services*, 2001.

125. L. Dutka, B. Kryza, K. Krawczyk, M. Majewska, R. Slota, L. Hluchy, and J. Kitowski. Component-Expert Architecture for Supporting Grid Workflow Construction Based on Knowledge. In P. Cunningham and M. Cunningham, editors, *Innovation and the Knowledge Economy: Issues, Applications, Case Studies*, volume 2, pages 239–246. IOS Press, Amsterdam, 2005.

126. Eclipse. Graphical editing framework. See Web site at `http://www.eclipse.org/gef/`.

127. Eclipse. Standard widget toolkit. See Web site at `http://www.eclipse.org/swt/`.

128. EGEE: Enabling Grids for E-science in Europe. See Web site at `http://public.eu-egee.org/`.

129. Einstein@Home Project. See Web site at `http://www.physics2005.org/events/einsteinathome/`.

130. J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming Heterogeneity — the Ptolemy Approach. *Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software*, 91(1):127–144, 2003.

131. J. Elith and M. Burgman. Predictions and Their Validation: Rare Plants in the Central Highlands, Victoria. In J. Scott, P. J. Heglund, and M. L. Morrison, editors, *Predicting Species Occurrences: Issues of Scale and Accuracy*. Island Press, Washington, DC, 2002.

132. W. Emmerich, B. Butchart, L. Chen, B. Wassermann, and S. L. Price. Grid Service Orchestration using the Business Process Execution Language (BPEL). *Journal of Grid Computing*, 3(3–4):283–304, 2005.

133. D. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A Worldwide Flock of Condors: Load Sharing Among Workstation Clusters. *Future Generation Computer Systems (Special Issue: Resource Management in Distributed Systems)*, 12(1):53–65, 1996.

134. M. H. Eres, G. E. Pound, Z. Jiao, J. L. Wason, F. Xu, A. J. Keane, and S. J. Cox. Implementation of a Grid-Enabled Problem Solving Environment in Matlab. In G. Goos, J. Hartmanis, and J. van Leeuwen, editors, *ICCS 2003: International Conference on Computational Science*, volume 2660 of *Lecture Notes in Computer Science*, pages 420–429. Springer, Berlin, 2003.

135. T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall, Englewood Cliffs, NJ, 2005.

136. Etnus. Totalview. `http://www.etnus.com/`.

137. T. Fahringer, R. Prodan, R. Duan, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, A. Villazon, and M. Wieczorek. ASKALON: A Grid Application Development and Computing Environment. In *6th International Workshop on Grid Computing*, pages 122–131. IEEE Computer Society Press, New York, 2005.

138. T. Fahringer, J. Qin, and S. Hainzer. Specification of Grid Workflow Applications with AGWL: An Abstract Grid Workflow Language. In *International Symposium on Cluster Computing and the Grid (CCGRID 2005)*, volume 2, pages 676–685. IEEE Computer Society Press, New York, 2005.

139. A. Faulkener, I. Stairs, M. Kramer, A. Lyne, G. Hobbs, A. Possenti, D. Lorimer, R. Manchester, M. McLaughlin, N. DÁmico, F. Camilo, and M. Burgay. The Parkes Multibeam Pulsar Survey: V. Finding binary and millisecond pulsars. *Monthly Notices of the Royal Astronomical Society*, 355(1):147–159, 2004.

140. E. Field, T. Jordan, and C. Cornell. OpenSHA: A Community-Modeling Environment for Seismic Hazard Research. *Seismological Research Letters*, 74(4):406–419, 2003.

141. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. Internet RFC 2616, W3C, 1999.

142. Flexible Image Transport System. `http://fits.gsfc.nasa.gov/`.

143. Formal Systems (Europe) Ltd. Failure Divergence Refinement: FDR2 User Manual. `http://www.fsel.com/documentation/fdr2/`.

144. I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputing Applications*, 11(2):115–128, 1997.

145. I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computer Infrastructure.* Morgan-Kaufmann, San Francisco, 1999.

146. I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Technical report, Open Grid Service Infrastructure WG, Global Grid Forum, 2002.

147. I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organization. *The International Journal of High Performance Computing Applications*, 15(3):200–222, 2001.

148. I. Foster, J. Voeckler, M. Wilde, and Y. Zhao. Chimera: A Virtual Data System for Representing, Querying, and Automating Data Derivation. In *14th International Conference on Scientific and Statistical Database Management (SSDBM'02)*, pages 37–46. IEEE Computer Society Press, New York, 2002.

149. I. Foster, J. Voeckler, M. Wilde, and Y. Zhao. The Virtual Data Grid: A New Model and Architecture for Data-Intensive Collaboration. In *15th International Conference on Scientific and Statistical Database Management, 2003*, page 11. IEEE Computer Society Press, New York, 2003.

150. Fraunhofer-Gesellschaft. The Fraunhofer Resource Grid. `http://www.fhrg.fraunhofer.de/`, 2006.

151. J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. *Cluster Computing*, 5:237–246, 2002.

152. J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPCD-'01)*. IEEE Computer Society, New York, 2001.

153. N. Furmento, W. Lee, A. Mayer, S. Newhouse, and J. Darlington. ICENI: an Open Grid Service Architecture Implemented with Jini. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–10. IEEE Computer Society Press, New York, 2002.

154. M. Galperin. The Molecular Biology Database Collection: 2006 update. *Nucleic Acids Research*, 34(Database issue):3–5, 2006.

155. E. Gamma and K. Beck. *Contributing to eclipse: Principles, Patterns, and Plug-Ins.* Addison-Wesley, Reading, MA, 2004.

156. D. Gannon, B. Plale, M. Christie, L. Fang, Y. Huang, S. Jensen, G. Kandaswamy, S. Marru, S. L. Pallickara, S. Shirasuna, Y. Simmhan, A. Slominski, and Y. Sun. Service Oriented Architectures for Science Gateways on Grid Systems. In *International Conference on Service Oriented Computing*

2005, volume 3826 of *Lecture Notes in Computer Science*, pages 21–32. Springer-Verlag, Berlin, 2005.

157. Grid Enabled web eNvironment for site Independent User job Submission (GENIUS) Portal. `https://genius.ct.infn.it/`.

158. GEO 600 aims at the direct detection of gravitational waves. `http://www.geo600.uni-hannover.de/`.

159. Global Grid Forum (GGF). `http://www.ogf.org/`. Now the Open Grid Forum (OGF).

160. Y. Gil, E. Deelman, J. Blythe, C. Kessleman, and H. Tangmunarunkit. Artificial Intelligence and Grids: Workflow Planning and Beyond. *IEEE Intelligent Systems Special Issue on e-Science*, 19(1):26–33, 2004.

161. T. Glatard, J. Montagnat, and X. Pennec. An Optimized Workflow Enactor for Data-Intensive Grid Applications. Technical Report 05.32, I3S Laboratory, Sophia Antipolis, France, 2005.

162. T. Glatard, J. Montagnat, and X. Pennec. Grid-Enabled Workflows for Data Intensive Applications. In *CBMS '05: Proceedings of the 18th IEEE Symposium on Computer-Based Medical Systems (CBMS'05)*. IEEE Computer Society, Washington, 2005.

163. The Galactic Legacy Infrared Mid-Plane Survey Extraordinaire (GLIMPSE). `http://www.astro.wisc.edu/sirtf/`.

164. GLIMPSE Validation Images. `http://www.astro.wisc.edu/sirtf/2massimages/2massimages.html`.

165. gLite middleware. `http://www.gLite.org`.

166. The Globus Alliance. See Web site at `http://www.globus.org`.

167. T. Goodale, G. Allen, G. Lanfermann, J. Masso, T. Radke, E. Seidel, and J. Shalf. The Cactus Framework and Toolkit: Design and Applications. In *Vector and Parallel Processing VECPAR 2002, 5th International Conference*, volume 2565 of *Lecture Notes in Computer Science*, pages 197–227. Springer, Berlin, 2003.

168. T. Goodale, I. Taylor, and I. Wang. Integrating Cactus Simulations within Triana Workflows. In *Proceedings of 13th Annual Mardi Gras Conference - Frontiers of Grid Applications and Technologies*, pages 47–53. Louisiana State University, 2005.

169. J. Grethe, C. Baru, A. Gupta, M. James, B. Ludäscher, M. Martone, P. Papadopoulos, P. ST., A. Rajasekar, S. Santini, I. Zaslavsky, and M. Ellisman. Biomedical Informatics Research Network: Building a National Collaboratory to Hasten the Derivation of New Understanding and Treatment of Disease. *Studies in Health Technologies Informatics*, 112:100–109, 2005.

170. Grid Adaptive Computational Engine (GrACE). `http://www.caip.rutgers.edu/~parashar/TASSL/Projects/GrACE/`.

171. Grid ENabled Integrated Earth system model project. `http://www.genie.ac.uk/`.

172. Grid3+. See `http://www.ivdgl.org/grid2003`.

173. Grid5000 French National Grid Initiative. `http://www.grid5000.org`.

174. GridCat: OSG CE Catalog. `http://osg-cat.grid.iu.edu`.

175. GridLab: A Grid Application Toolkit and Testbed Project home page. `http://www.gridlab.org`.

176. DFN-Verein Project "Development of Grid Based Simulation and Visualization Techniques" (GRIKSL) Home Page. `http://www.griksl.org`.

177. J. Grinnell. Field Tests of Theories Concerning Distributional Control. *American Naturalist*, 51:115–128, 1917.

178. J. Grinnell. Geography and evolution. *Ecology*, 5:225–229, 1924.

179. W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

180. J. Grundy, M. Apperley, J. Hosking, and W. Mugridge. A Decentralized Architecture for Software Process Modeling and Enactment. *IEEE Internet Computing*, 2(5):53–62, 1998.

181. J. Grundy, J. Hosking, R. Amor, W. Mugrdige, and M. Li. Domain-Specific Visual Languages for Specifying and Generating Data Mapping System. *Journal of Visual Languages and Computing*, 15(3–4):243–263, 2004.

182. GSI-Enabled OpenSSH. See `http://grid.ncsa.uiuc.edu/ssh/`.

183. T. Gubala, M. Bubak, M. Malawski, and K. Rycerz. Semantic-Based Grid Workflow Composition. In R. Wyrzykowski, J. Dongarra, N. Meyer, and J. Wasniewski, editors, *Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics PPAM'2005*, volume 3911 of *Lecture Notes in Computer Science*, pages 651–658. Springer, New York, 2005.

184. M. Gudgin, M. Hadley, and Tony Rogers (eds). Web Services Addressing 1.0 - Core (WS-Addressing). Technical report, W3C, 2006.

185. M. Y. Gulamali, A. S. McGough, R. J. Marsh, N. R. Edwards, T. M. Lenton, P. J. Valdes, S. J. Cox, S. J. Newhouse, and J. Darlington. Performance Guided Scheduling in GENIE through ICENI. In S. J. Cox, editor, *Proceedings of the UK e-Science All Hands Meeting 2004*, pages 259–266. CD Rom Proceedings, 2004.

186. W. W. Hargrove, R. H. Gardner, M. G. Turner, W. H. Romme, and D. G. Despain. Simulating Fire Patterns in Heterogeneous Landscapes. *Ecological Modelling*, 135:243–263, 2000.

187. A. Harrison and I. Taylor. WSPeer – An Interface to Web Service Hosting and Invocation. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 4*, page 175a. IEEE Computer Society, New York, 2005.

188. J. Hau, W. Lee, and J. Darlington. A Semantic Similarity Measure for Semantic Web Services. In *Web Service Semantics: A Workshop at The 14th International World Wide Web Conference (WWW2005)*. CD Rom Proceedings, 2005.

189. R. L. Henderson. Job Scheduling Under the Portable Batch System. In D. G. Feitelson and L. Rudolph, editors, *IPPS '95: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 279–294. Springer Verlag, London, 1995.

190. S. I. Higgins, D. M. Richardson, R. M. Cowling, and T. H. Trinder-Smith. Predicting the Landscape-Scale Distribution of Alien Plants and Their Threat to Plant Diversity. *Conservation Biology*, 13:303–313, 1999.

191. S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to Petri Nets. In W. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, editors, *Proceedings of the Third International Conference on Business Process Management (BPM 2005)*, volume 3649 of *Lecture Notes in Computer Science*, pages 220–235. Springer-Verlag, Berlin, 2005.

192. C. Hoare. *Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs, NJ, 1985.

193. A. Hoheisel and U. Der. An XML-Based Framework for Loosely Coupled Applications on Grid Environments. In P. M. A. Sloot, D. Abramson, A. V. Bogdanov, J. J. Dongarra, A. Y. Zomaya, and Y. E. Gorbachev, editors, *Computational Science — ICCS 2003*, volume 2657 of *Lecture Notes in Computer Science*, pages 245–254. Springer-Verlag, Berlin, 2003.

194. A. Hoheisel and U. Der. Dynamic Workflows for Grid Applications. In *Proceedings of the Cracow Grid Workshop '03*. Academic Computer Centre CYFRONET AGH, Cracow, Poland, 2003.

195. J. R. Holden and H. Ammon. Prediction of Possible Crystal Structures for C-, H-, N-, O-, and F-Containing Organic Compounds. *Journal of Computational Chemistry*, 14(4):422–437, 1993.

196. R. D. Holt. Adaptive Evolution in Source-Sink Environments: Direct and Indirect Effects of Density-Dependence on Niche Evolution. *Oikos*, 75:182–192, 1996.

197. R. D. Holt and M. S. Gaines. Analysis of Adaptation in Heterogeneous Landscapes: Implications for the Evolution of Fundamental Niches. *Evolutionary Ecology*, 6:433–447, 1992.

198. G. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison Wesley, Reading MA, 2003.

199. M. Hovestadt, O. Kao, A. Keller, and A. Streit. Scheduling in HPC Resource Management Systems: Queuing vs. Planning. In D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *9th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2862 of *Lecture Notes in Computer Science*, pages 1–20. Springer, Berlin, 2003.

200. Y. Huang, A. Slominski, C. Herath, and D. Gannon. WS-Messenger: A Web Services based Messaging System for Service-Oriented Grid Computing. In *CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, pages 166–173. IEEE Computer Society, Washington, 2006.

201. M. Hucka, A. Finney, H. Sauro, H. Bolouri, J. Doyle, H. Kitano, A. Arkin, B. Bornstein, D. Bray, A. Cornish-Bowden, A. Cuellar, S. Dronov, E. Gilles, M. Ginkel, V. Gor, I. Goryanin, W. Hedley, T. Hodgman, J. Hofmeyr, P. Hunter, N. Juty, J. Kasberger, A. Kremling, U. Kummer, N. Le Novère, L. Loew, D. Lucio, P. Mendes, E. Minch, E. Mjolsness, Y. Nakayama, M. Nelson, P. Nielsen, T. Sakurada, J. Schaff, B. Shapiro, T. Shimizu, H. Spence, J. Stelling, K. Takahashi, M. Tomita, J. Wagner, and J. Wang. The Systems Biology Markup Language (SBML): A Medium for Representation and Exchange of Biochemical Network Models. *Bioinformatics*, 19(4):524–531, 2003.

202. D. Hull, R. Stevens, P. Lord, C. Wroe, and C. Goble. Treating shimantic web syndrome with ontologies. In J. Domingue, L. Cabral, and E. Motta, editors, *First Advanced Knowledge Technologies workshop on Semantic Web Services (AKT-SWS04) KMi, The Open University, Milton Keynes, UK. 2004-12-08*, volume 122, page 1. CEUR Workshop Proceedings (CEUR-WS.org), 2004.

203. L. M. Hunter and L. Rinner. The Association Between Environmental Perspective and Knowledge and Concern with Species Diversity. *Society & Natural Resources*, 17(6):517–532, 2004.

204. B. Huntley, P. J. Bartlein, and I. C. Prentice. Climatic Control of the Distribution and Abundance of Beech (*Fagus* L.) in Europe and North America. *Journal of Biogeography*, 16:551–560, 1989.

205. B. Huntley, P. M. Berry, W. Cramer, and A. P. McDonald. Modelling Present and Potential Future Ranges of Some European Higher Plants Using Climate Response Surfaces. *Journal of Biogeography*, 22:967–1001, 1995.
206. IBM. WSDL4J. See Web site at `http://sourceforge.net/projects/wsdl4j`.
207. IBM and BEA. BPELJ: BPEL for Java. `http://www.ibm.com/developerworks/webservices/library/ws-bpelj/`.
208. IBM Alphaworks. Virtual XML Garden. `http://www.alphaworks.ibm.com/tech/virtualxml`.
209. IBM DB2. See `http://www-306.ibm.com/software/data/db2`.
210. IBM Websphere. See Web site at `http://www-306.ibm.com/software/websphere/`.
211. Illinois BioGrid. `http://www.illinoisbiogrid.org/`.
212. Immunology Grid. Immunology Grid Project. `http://www.immunologygrid.org`.
213. Instant-Grid — A Grid Demonstration Toolkit. `http://instant-grid.de/`.
214. Intel. `http://www.intel.com`.
215. International Virtual Data Grid Laboratory. See Project Web site at `http://www.ivdgl.org`.
216. IPHAS Image Gallery. `http://astro.ic.ac.uk/Research/Halpha/North/gallery.shtml`.
217. IPHAS: The INT H-Alpha Emission Survey. `http://iapetus.phy.umist.ac.uk/IPHAS/iphas.html`.
218. R. Irani and S. J. Bashna. *AXIS: Next Generation Java SOAP*. Wrox Press, Hoboken, NJ, 2002.
219. ISO/IEC 10026-1. Information Technology — Open Systems Interconnection — Distributed Transaction Processing — Part 1: OSI TP Model, 1998.
220. ISO/IEC 15909-1. High-Level Petri Nets — Part 1: Concepts, Definitions and Graphical Notation, 2004.
221. ISO/IEC 15909-2. High-level Petri Nets — Part 2: Transfer Format, 2005. Working Draft.
222. K. Jensen. An Introduction to the Theoretical Aspects of Coloured Petri Nets. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency, Reflections and Perspectives, REX School/Symposium*, volume 803 of *Lecture Notes in Computer Science*, pages 230–272. Springer-Verlag, London, 1994.
223. A. C. Jones, R. J. White, W. A. Gray, F. A. Bisby, N. Caithness, N. Pittas, X. Xu, T. Sutton, N. J. Fiddian, A. Culham, M. Scoble, P. Williams, O. Bromley, P. Brewer, C. Yesson, and S. Bhagwat. Building a Biodiversity GRID. In A. Konagaya and K. Satou, editors, *Grid Computing in Life Science: First International Workshop on Life Science Grid*, volume 3370 of *Lecture Notes in Computer Science*, pages 140–151. Springer, Berlin, 2005.
224. A. C. Jones, X. Xu, N. Pittas, W. A. Gray, N. J. Fiddian, R. J. White, J. S. Robinson, F. A. Bisby, and S. M. Brandt. SPICE: A Flexible Architecture for Integrating Autonomous Databases to Comprise a Distributed Catalogue of Life. In *Database and Expert Systems Applications: 11th International Conference, DEXA 2000*, volume 1873 of *Lecture Notes in Computer Science*, pages 981–992. Springer, Berlin, 2000.
225. M. Jones. SEEK EcoGrid: Integrating Data and Computational Resources for Ecology. *DataBits: An Electronic Newsletter for Information Managers, Long Term Ecological Research Program*, Spring:1, 2003.

226. T. Jordan and P. Maechling. The SCEC Community Modeling Environment
     — An Information Infrastructure for System-Level Earthquake Science.
     Seismological Research Letters, 74(3):324–328, 2003.

227. Job Submission Description Language Working Group (JSDL-WG). See Web
     site at `https://forge.gridforum.org/projects/jsdl-wg/`.

228. M. Jüngel, E. Kindler, and M. Weber. The Petri Net Markup Language. In
     S. Philippi, editor, *7. Workshop Algorithmen und Werkzeuge für Petrinetze*,
     pages 47–52. Universität Koblenz-Landau, 2000.

229. P. Kacsuk, A. Goyeneche, T. Delaitre, T. Kiss, Z. Farkas, and T. Boczko.
     High-Level Grid Application Environment to Use Legacy Codes as OGSA Grid
     Services. In *GRID '04: Proceedings of the Fifth IEEE/ACM International
     Workshop on Grid Computing (GRID'04)*, pages 428–435. IEEE Computer
     Society, Washington, 2004.

230. P. Kacsuk and G. Sipos. Multi-Grid, Multi-User Workflows in the P-GRADE
     Grid Portal. *Journal of Grid Computing*, 3(3–4), 2005.

231. V. Kalogera, C. Kim, D. R. Lorimer, M. Burgay, N. D'Amico, A. Possenti,
     R. N. Manchester, A. G. Lyne, B. C. Joshi, M. A. McLaughlin, M. Kramer,
     J. M. Sarkissian, and F. Camilo. The Cosmic Coalescence Rates for Double
     Neutron Star Binaries. *The Astrophysical Journal*, 601:L179–L182, 2004.

232. G. Kandaswamy, L. Fang, Y. Huang, S. Shirasuna, S. Marru, and D. Gannon.
     Building Web services for Scientific Grid Applications. *IBM Journal of
     Research and Development*, 50(2/3):249–260, March/May 2006.

233. K. Karasavvas, M. Antonioletti, M. Atkinson, N. C. Hong, T. Sugden,
     A. Hume, M. Jackson, A. Krause, and C. Palansuriya. Introduction to
     OGSA-DAI Services. In G. Goos, J. Hartmanis, and J. van Leeuwen, editors,
     *First International Workshop on Scientific Applications of Grid Computing
     (SAG 2004)*, volume 3458 of *Lecture Notes in Computer Science*, pages 1–12.
     Springer, Berlin, 2005.

234. D. S. Katz, J. C. Jacob, G. B. Berriman, J. Good, A. C. Laity, E. Deelman,
     C. Kesselman, G. Singh, M.-H. Su, and T. A. Prince. A Comparision of
     Two Methods for Building Astronomical Image Mosaics on a Grid. In
     *34th International Conference on Parallel Processing Workshops ICPP 2005
     Workshops*, pages 85–94. IEEE Computer Society, New York, 2005.

235. K. Keahey, K. Doering, and I. Foster. From Sandbox to Playground: Dynamic
     Virtual Environments in the Grid. In *GRID '04: Proceedings of the Fifth
     IEEE/ACM International Workshop on Grid Computing (GRID'04)*, pages
     34–42. IEEE Computer Society, Washington, 2004.

236. J. Kim, Y. Gil, and M. Spraragen. A Knowledge-Based Approach to Interactive
     Workflow Composition. In S. Zilberstein, J. Koehler, and S. Koenig, editors,
     *14th International Conference on Automated Planning and Scheduling (ICAPS
     04)*. AAAI Press, Menlo Park, CA, 2004.

237. J. Kim, M. Spraragen, and Y. Gil. An Intelligent Assistant for Interactive
     Workflow Composition. In *IUI '04: Proceedings of the 9th international
     conference on Intelligent user interface*, pages 125–131. ACM Press, New York,
     January 2004.

238. K. Knight and D. Marcu. Machine Translation in the Year 2004. In *Proceedings
     of the 2005 IEEE International Conference on Acoustics, Speech, and Signal
     Processing (ICASSP)*, volume 5, pages 965–968. IEEE Computer Society, New
     York, 2005.

239. S.-H. Ko, K. W. Cho, Y. D. Song, and Y. G. Kim. Development of Cactus Driver for CFD Analyses in the Grid Computing Environment. In G. Goos, J. Hartmanis, and J. van Leeuwen, editors, *Advances in Grid Computing — EGC 2005*, volume 3470 of *Lecture Notes in Computer Science*, pages 771–777. Springer, Berlin, 2005.

240. G. Kola, T. Kosar, and M. Livny. A Client-centric Grid Knowledgebase. In *Proceedings of 2004 IEEE International Conference on Cluster Computing*, pages 431–438. IEEE Computer Society, New York, 2004.

241. G. Kola, T. Kosar, and M. Livny. Run-time Adaptation of Grid Data Placement Jobs. *Scalable Computing: Practice and Experience*, 6(3):33–43, 2005.

242. G. Kola and M. Livny. Diskrouter: A Flexible Infrastructure for High Performance Large Scale Data Transfers. Technical Report CS-TR-2003-1484, University of Wisconsin–Madison Computer Sciences Department, 2003.

243. T. Kosar. *Data Placement in Widely Distributed Systems*. PhD thesis, University of Wisconsin–Madison, 2005.

244. T. Kosar and M. Livny. Stork: Making Data Placement a First Class Citizen in the Grid. In *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 342–349. IEEE Computer Society, Washington, 2004.

245. S. Krishnan and D. Gannon. XCAT3: A Framework for CCA Components as OGSA Services. In *9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004)*, pages 90–97. IEEE Computer Society, New York, 2004.

246. S. Krishnan, P. Wagstrom, and G. von Laszewski. GSFL: A Workflow Framework for Grid Services. Preprint ANL/MCS-P980-0802, 2002.

247. D. Kuo, P. Greenfield, S. Parastatidis, and J. Webber. Rules-based SSDL Protocol Framework. `http://www.ssdl.org/docs/v1.3/html/Rules%20SSDL%20Protocol%20Framework%20v1.3.html`, April 2005.

248. LCG2 middleware. `http://lcg.web.cern.ch/LCG/activities/middleware.html`.

249. Linked Environments for Atmospheric Discovery. `http://lead.ou.edu/`.

250. LEAD Year-2 Annual Report. `http://lead.ou.edu/pdfs/LEAD_Year-2_Report.pdf`, 2005.

251. W. Lee, A. McGough, and J. Darlington. Performance Evaluation of the GridSAM Job Submission and Monitoring System. In S. J. Cox and D. W. Walker, editors, *UK e-Science All Hands Meeting, 2005*, pages 915–922. CD Rom Proceedings, 2005.

252. W. Lee, S. McGough, S. Newhouse, and J. Darlington. A Standard Based Approach to Job Submission through Web Services. In S. J. Cox, editor, *Proceedings of the UK e-Science All Hands Meeting 2004*, pages 901–905. CD Rom Proceedings, 2004.

253. R. S. Levine, M. Q. Benedict, and A. T. Peterson. Distribution of *Anopheles quadrimaculatus* Say s.l. and Implications for Its Role in Malaria Transmission in the US. *Journal of Medical Entomology*, 41:607–613, 2004.

254. F. Leyman. Web Services Flow Language (WSFL) 1.1. Technical report, IBM Software Group, New York, 2001.

255. F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall, Englewood Cliffs, NJ, 1999.

256. LIGO Data Grid (LDG).
     `http://www.lsc-group.phys.uwm.edu/lscdatagrid`.
257. LIGO Data Replicator. See `http://www.lsc-group.phys.uwm.edu/ldr`.
258. LIGO home page. `http://www.ligo.caltech.edu`.
259. A. Lin, L. Dai, K. Ung, S. Peltier, and M. Ellisman. The Telescience Project: Applications to Middleware Interaction Components. In *Proceedings of The 18th IEEE International Symposium on Computer-Based Medical Systems (CBMS 2005)*, pages 543–548. IEEE Computer Society, New York, 2005.
260. A. W. Lin, L. Dai, J. Mock, S. Peltier, and M. H. Ellisman. The Telescience Tools: Version 2.0. In *E-SCIENCE '05: Proceedings of the First International Conference on e-Science and Grid Computing*, pages 56–63. IEEE Computer Society, Washington, 2005.
261. M. Little, J. Webber, and S. Parastatidis. Stateful interactions in Web Services: a comparison of WS-Context and WS-Resource Framework. *SOA Web Services Journal*, May 2004.
262. M. Litzkow, M. Livny, and M. Mutka. Condor — A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 104–111. IEEE Computer Society, New York, June 1988.
263. London e-Science Centre. A Market for Computational Services. Available at `http://www.lesc.ic.ac.uk/markets/`.
264. C. J. Lonsdale, H. E. Smith, M. Rowan-Robinson, J. Surace, D. Shupe, C. Xu, S. Oliver, D. Padgett, F. Fang, T. Conrow, A. Franceschini, N. Gautier, M. Griffin, P. Hacking, F. Masci, G. Morrison, J. O'Linger, F. Owen, I. Pérez-Fournon, M. Pierre, R. Puetter, G. Stacey, S. Castro, M. D. C. Polletta, D. Farrah, T. Jarrett, D. Frayer, B. Siana, T. Babbedge, S. Dye, M. Fox, E. Gonzalez-Solares, M. Salaman, S. Berta, J. J. Condon, H. Dole, and S. Serjeant. SWIRE: The SIRTF Wide-Area Infrared Extragalactic Survey. *Publications of the Astronomical Society of the Pacific*, 115(810):897–927, 2003.
265. M. Lorch and D. Kafura. Symphony — A Java-Based Composition and Manipulation Framework for Computational Grids. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, page 136. IEEE Computer Society Press, New York, 2002.
266. H. D. Lord. Improving the Application Environment with Modular Visualization Environments. *Computer Graphics*, 29(2):10–12, 1995.
267. P. Lord, P. Alper, C. Wroe, and C. Goble. Feta: A Light-Weight Architecture for User Oriented Semantic Service Discovery. In A. Gómez-Pérez and J. Euzenat, editors, *The Semantic Web: Research and Applications. Second European Semantic Web Conference, ESWC 2005*, volume 3532 of *Lecture Notes in Computer Science*, pages 17–31. Springer, Berlin, 2005.
268. P. Lord, S. Bechhofer, M. D. Wilkinson, G. Schiltz, D. Gessler, D. Hull, C. Goble, and L. Stein. Applying Semantic Web Services to Bioinformatics: Experiences Gained, Lessons Learnt. In *3rd International Semantic Web Conference (ISWC2004)*, volume 3298 of *Lecture Notes in Computer Science*, pages 350–364. Springer, Berlin, 2004.
269. P. Lord, C. Wroe, R. Stevens, C. Goble, S. Miles, L. Moreau, K. Decker, T. Payne, and J. Papay. Semantic and Personalised Service Discovery. In W. K. Cheung and Y. Ye, editors, *WI/IAT 2003 Workshop on Knowledge Grid and Grid Intelligence*, pages 100–107. CD Rom Proceedings, 2003.

270. D. Lorimer and M. Kramer. *A Handbook of Pulsar Astronomy*. Cambridge University Press, Cambridge, 2005.

271. Load Sharing Facility. See Web site at `http://accl.grc.nasa.gov/lsf/`.

272. B. Ludäscher, I. Altintas, C. Berkley, D. G. Higgins, E. Jaeger, M. Jones, E. A. Lee, and Y. Zhao. Scientific Workflow Management and the Kepler System: Research Articles. *Concurrency and Computation: Practice and Experience, Special Issue on Workflow in Grid Systems*, 18(10):1039–1065, 2006.

273. S. A. Ludwig, W. Naylor, J. Padget, and O. F. Rana. Matchmaking Support for Mathematical Web Services. In S. J. Cox and D. W. Walker, editors, *UK e-Science All Hands Meeting, 2005*, pages 391–399, Nottingham, September 2005. CD Rom Proceedings.

274. A. G. Lyne and G. Smith. *Pulsar Astronomy*. Cambridge University Press, Cambridge, 3rd edition, 2005.

275. M. Fowler. Inversion of Control Containers and the Dependency Injection Pattern.
`http://martinfowler.com/articles/injection.html#InversionOfControl`.

276. P. Maechling, H. Chalupsky, M. Dougherty, E. Deelman, Y. Gil, S. Gullapalli, V. Gupta, C. Kesselman, J. Kim, G. Mehta, B. Mendenhall, T. Russ, G. Singh, M. Spraragen, G. Staples, and K. Vahi. Simplifying Construction of Complex Workflows for Non-Expert Users of the Southern California Earthquake Center Community Modeling Environment. *ACM SIGMOD Record*, 34(3):24–30, 2005.

277. P. Maechling, V. Gupta, N. Gupta, E. Field, D. Okaya, and T. Jordan. Grid Computing in the SCEC Community Modeling Environment. *Seismological Research Letters*, 76(5):518–587, 2005.

278. P. Maechling, V. Gupta, N. Gupta, E. Field, D. Okaya, and T. Jordan. Hazard Map Calculations Using Grid Computing. *Seismological Research Letters*, 76(5):565–573, 2005.

279. M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. Wiley Series in Parallel Computing. John Wiley and Sons, Chichester, 1995.

280. E. Martínez-Meyer. *Evolutionary Trends in Ecological Niches of Species*. PhD thesis, University of Kansas, 2002.

281. E. Martínez-Meyer, A. T. Peterson, and W. W. Hargrove. Ecological Niches as Stable Distributional Constraints on Mammal Species, with Implications for Pleistocene Extinctions and Climate Change Projections for Biodiversity. *Global Ecology and Biogeography*, 13:305–314, 2004.

282. Matt Anderson. `http://relativity.phys.lsu.edu/postdocs/matt/`.

283. A. Mayer, S. McGough, N. Furmento, J. Cohen, M. Gulamali, L. Young, A. Afzal, S. Newhouse, and J. Darlington. ICENI: An Integrated Grid Middleware to Support e-Science. In V. Getov and T. Kielmann, editors, *Component Models and Systems for Grid Applications. Proceedings of the Workshop on Component Models and Systems for Grid Applications*, volume 1, pages 109–124. Springer, New York, June 2004.

284. A. Mayer, S. McGough, M. Gulamali, L. Young, J. Stanton, S. Newhouse, and J. Darlington. Meaning and Behaviour in Grid Oriented Components. In *GRID '02: Proceedings of the Third International Workshop on Grid Computing*, volume 2536 of *Lecture Notes in Computer Science*, pages 100–111. Springer-Verlag, London, 2002.

285. A. E. Mayer. *Composite Construction of High Performance Scientific Applications*. PhD thesis, Department of Computing, Imperial College, London, UK, 2001.

286. D. McDermott. Estimated-Regression Planning for Interactions with Web Services. In M. Ghallab, J. Hertzberg, and P. Traverso, editors, *6th International Conference on Artificial Intelligence Planning and Scheduling*. AAAI Press, Menlo Park, CA, 2002.

287. A. McGough, L. Young, A. Afzal, S. Newhouse, and J. Darlington. Workflow Enactment in ICENI. In S. J. Cox, editor, *Proceedings of the UK e-Science All Hands Meeting 2004*, pages 894–900. CD Rom Proceedings, 2004.

288. S. McGough, A. Afzal, J. Darlington, N. Furmento, A. Mayer, and L. Young. Making the Grid Predictable through Reservations and Performance Modelling. *The Computer Journal*, 48(3):358–368, 2005.

289. S. McIlraith and T. Son. Adapting Golog for Programming in the Semantic Web. In *Fifth International Symposium on Logical Formalizations of Commonsense Reasoning*, pages 195–202. In press, 2001.

290. M. McIlroy. Mass Produced Software Components. In P. Naur and B. Randell, editors, *Software Engineering: Report of a conference sponsored by the NATO Science Committee*, pages 79–87. Scientific Affairs Division, NATO, Brussels, 1968.

291. J. O. Meyneeke. Effects of Global Climate Change on Geographic Distributions of Vertebrates in North Queensland. *Ecological Modelling*, 174(4):347–357, 2004.

292. Microsoft Corporation. Web Services Specifications Index. `http://msdn.microsoft.com/webservices/understanding/specs/`.

293. Microsoft Corporation. Windows Workflow Foundation. `http://msdn.microsoft.com/winfx/reference/workflow/`, 2005.

294. L. Miles, A. Grainger, and O. Phillips. The Impact of Global Climate Change on Tropical Forest Biodiversity in Amazonia. *Global Ecology and Biogeography*, 13(6):553–565, 2004.

295. Millenium Ecosystem Assessment. Ecosystem and Human Well-Being Reports. Island Press, Washington, DC, 2005. `http://www.millenniumassessment.org`.

296. R. Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, Cambridge, 1999.

297. D. J. Mladenoff, T. A. Sickley, R. G. Haight, and A. P. Wydeven. A Regional Landscape Analysis and Prediction of Favorable Gray Wolf Habitat in the Northern Great Lakes Region. *Conservation Biology*, 9:279–294, 1995.

298. R. Monson-Haefel. *Enterprise Java Beans*. O'Reilly, Sebastopol, CA, 2001.

299. The Montage Project Web page. `http://montage.ipac.caltech.edu`.

300. Montage Version 1.7.x documentation and download. `http://montage.ipac.caltech.edu/docs/`.

301. Montage Version 1.7.x. Photometric and Calibration Accuracy. `http://montage.ipac.caltech.edu/docs/accuracy.html`.

302. Mopex, the Spitzer Science Center Mosaic Engine. `http://ssc.spitzer.caltech.edu/postbcd/doc/mosaicer.pdf`.

303. L. Moreau, Y. Zhao, I. Foster, J. Voeckler, and M. Wilde. XDTM: XML Dataset Typing and Mapping for Specifying Datasets. In *Advances in Grid Computing — EGC 2005*, volume 3470 of *Lecture Notes in Computer Science*, pages 495–505. Springer, Berlin, 2005.

304. MOTEUR: Home-Made Optimised Scufl Enactor. `http://www.i3s.unice.fr/~glatard`.

305. N. Mulyar and W. van der Aalst. Patterns in Colored Petri Nets. In *BETA Working Paper Series*, WP 139. Eindhoven University of Technology, Eindhoven, 2005.

306. D. Murray, J. McWhirter, S. Wier, and S. Emmerson. The Integrated Data Viewer — A Web-Enabled Application for Scientific Analysis and Visualization. In *19th Conference on Interactive Information Processing Systems (IIPS) for Meteorology, Oceanography, and Hydrology*, page 13.2. American Meteorological Society, Boston, February 2003.

307. P. Murray-Rust. Chemical Markup Language. *World Wide Web Journal*, 2(4):135–147, 1997.

308. myGrid. `http://www.mygrid.org.uk/`.

309. H. Nakada, S. Matsuoka, K. Seymour, J. Dongarra, C. Lee, and H. Casanova. A GridRPC Model and API for End-User Applications. Technical report, Global Grid Forum (GGF), July 2005.

310. W. Nejdl, B. Wolf, C. Qu, S. Decker, M. Sintek, A. Naeve, M. Nilsson, M. Palmer, and T. Risch. EDUTELLA: A P2P Networking Infrastructure Based on RDF. In *11th World Wide Web Conference (WWW2002)*, page 604. CD Rom Proceedings, May 2002.

311. National e-Science Centre. See Web site at `http://www.nesc.ac.uk/`.

312. P. Newton and J. Browne. The CODE 2.0 Graphical Parallel Programming Language. In *ICS '92: Proceedings of the 6th international conference on Supercomputing*, pages 167–177. ACM Press, New York, 1992.

313. H. A. Nix. A Biogeographic Analysis of Australian Elapid Snakes. In R. Longmore, editor, *Atlas of Elapid Snakes of Australia*, pages 4–15. Australian Government Publishing Service, Canberra, 1986.

314. J. Novotny, S. Tuecke, and V. Welch. An Online Credential Repository for the Grid: MyProxy. In *Proceedings of the Tenth International Symposium on High Performance Distributed Computing (HPDC-10)*, HPDC, pages 104–114. IEEE Computer Society Press: Los Alamitos, CA, 2001.

315. OASIS. OASIS Web Services Business Process Execution Language (WSBPEL) TC. `http://www.oasis-open.org/committees/wsbpel`.

316. OASIS. Web Services Base Notification 1.3 (WS-BaseNotification). `http://docs.oasis-open.org/wsn/wsn-ws_base_notification-1.3-spec-pr-02.pdf`.

317. OASIS. Web Services Composite Application Framework (WS-CAF). `http://www.oasis-open.org/committees/ws-caf/charter.php`.

318. OASIS. Web Services Context (WS-CTX). `www.iona.com/devcenter/standards/WS-CAF/WSCTX.pdf`.

319. OASIS. Web Services Resource 1.2 (WS-Resource). `http://docs.oasis-open.org/wsrf/wsrf-ws_resource-1.2-spec-cs-01.pdf`.

320. OASIS. Web Services Security (WS-Security). `http://www.oasis-open.org/committees/wss`.

321. OASIS. WS-Resource Properties (WSRF-RP), April 2006. `http://docs.oasis-open.org/wsrf/wsrf-ws_resource_properties-1.2-spec-os.pdf`.

322. Object Management Group (OMG). The Common Object Request Broker Architecture (CORBA). `http://www.corba.org`.

323. A. O'Brien, S. Newhouse, and J. Darlington. Mapping of Scientific Workflow within the e-Protein Project to Distributed Resources. In S. J. Cox, editor, *Proceedings of the UK e-Science All Hands Meeting 2004*, pages 162–163. CD Rom Proceedings, 2004.

324. Open Grid Forum (OGF). `http://www.ogf.org/`. Formally the Global Grid Forum (GGF).

325. Open Grid Services Architecture. `https://forge.gridforum.org/projects/ogsa-wg`.

326. T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: A Tool for the Composition and Enactment of Bioinformatics Workflows. *Bioinformatics*, 20(17):3045–3054, November 2004.

327. K. B. Olsen, S. M. Day, J. B. Minster, Y. Cui, A. Chourasia, M. Faerman, R. Moore, P. Maechling, and T. H. Jordan. Strong Shaking in Los Angeles Expected from Southern San Andreas Earthquake. *Geological Research Letters*, 2006.

328. The Open Science Grid Consortium. `http://www.opensciencegrid.org/`.

329. B. J. Owen. Search Templates for Gravitational Waves from Inspiraling Binaries: Choice of Template Spacing. *Physical Review D*, 53:6749–6761, 1996.

330. B. J. Owen and B. S. Sathyaprakash. Matched Filtering of Gravitational Waves from Inspiraling Compact Binaries: Computational Cost and Template Placement. *Physical Review D*, 60:022002, 1999.

331. OWL Web Ontology Language. `http://www.w3.org/TR/owl-features/`.

332. OWL Services Coalition. OWL-S Semantic Markup for Web Services. `http://www.daml.org/services/owl-s/1.1/`, 2004.

333. S. Panagiotidi, E. Katsiri, and J. Darlington. On Advanced Scientific Understanding, Model Componentisation and Coupling in GENIE. In S. J. Cox and D. W. Walker, editors, *UK e-Science All Hands Meeting, 2005*, pages 559–567, Nottingham, UK, September 2005. CD Rom Proceedings.

334. S. Parastatidis and J. Webber. CSP SSDL Protocol Framework. `http://www.ssdl.org/docs/v1.3/html/CSP%20SSDL%20Protocol%20Framework%20v1.3.html`, April 2005.

335. S. Parastatidis and J. Webber. MEP SSDL Protocol Framework. `http://www.ssdl.org/docs/v1.3/html/MEP%20SSDL%20Protocol%20Framework%20v1.3.html`, April 2005.

336. S. Parastatidis and J. Webber. The SOAP Service Description Language. `http://www.ssdl.org/docs/v1.3/html/SSDL%20v1.3.html`, April 2005.

337. S. Parastatidis, J. Webber, S. Woodman, D. Kuo, and P. Greenfield. An Introduction to the SOAP Service Description Language v1.3. `http://www.ssdl.org/docs/v1.3/html/SSDL%20whitepaper%20v1.3.html`, April 2005.

338. S. Parastatidis, J. Webber, S. Woodman, D. Kuo, and P. Greenfield. Using SSDL to Describe and Reason about Asynchronous and Multi-Message Interactions between Web Services. *IEEE Internet Computing*, January-February 2006.

339. Portable Batch System. See `http://www.openpbs.org/`.

340. R. G. Pearson, T. P. Dawson, P. M. Berry, and P. A. Harrison. SPECIES: A Spatial Evaluation of Climate Impact on the Envelope of Species. *Ecological Modelling*, 154:289–300, 2002.

341. S. Peltier, A. Lin, D. Lee, S. Mock, S. Lamont, T. Molina, M. Wong, M. Martone, and M. Ellisman. The Telescience Portal for Advanced Tomography Applications. *Journal of Parallel and Distributed Applications*, 63(5):539–550, 2003.

342. Penn State LIGO Data Processing Center. `http://ligo.aset.psu.edu`.

343. D. Pennington and W. Michener. The EcoGrid and the Kepler Workflow System: A New Platform for Conducting Ecological Analyses. *ESA Bulletin (Emerging Technologies)*, 86:169–176, 2005.

344. S. Perera and D. Gannon. Enabling Web Service Extensions for Scientific Workflows. In *Workshop on Workflows in Support of Large-Scale Science (WORKS)*. IEEE Computer Society, New York, 2006.

345. A. T. Peterson. Predicting the Geography of Species' Invasions via Ecological Niche Modeling. *Quarterly Review of Biology*, 78:419–433, 2003.

346. A. T. Peterson and D. A. Kluza. New Distributional Modeling Approaches for Gap Analysis. *Animal Conservation*, 6:47–54, 2003.

347. A. T. Peterson, E. Martínez-Meyer, C. González-Salazar, and P. Hall. Modeled Climate Change Effects on Distributions of Canadian Butterfly Species. *Canadian Journal of Zoology*, 82:851–858, 2004.

348. A. T. Peterson, M. A. Ortega-Huerta, J. Bartley, V. Sanchez-Cordero, J. Soberon, R. H. Buddemeier, and D. R. B. Stockwell. Future Projections for Mexican Faunas Under Global Climate Change Scenarios. *Nature*, 416:626–629, 2002.

349. A. T. Peterson, J. Soberón, and V. Sanchez-Cordero. Conservatism of Ecological Niches in Evolutionary Time. *Science*, 285:1265–1267, 1999.

350. A. T. Peterson and D. A. Vieglais. Predicting Species Invasions Using Ecological Niche Modeling. *BioScience*, 51:363–371, 2001.

351. C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für Instrumentelle Mathematik, Bonn, 1962.

352. P-GRADE Portal. `http://www.lpds.sztaki.hu/pgportal/`.

353. S. J. Phillips, M. Dudik, and R. E. Schapire. A Maximum Entropy Approach to Species Distribution Modeling. In *Proceedings of the International Conference on Machine Learning*. ACM Press, New York, 2004.

354. S. Pickles, J. Brooke, F. Costen, E. Gabriel, M. Müller, M. Resch, and S. Ord. Metacomputing Across Intercontinental Networks. *Future Generation Computer Systems*, 17(5–6):911–918, 2001.

355. S. M. Pickles, P. V. Coveney, and B. M. Boghosian. Transcontinental RealityGrids for Interactive Collaborative Exploration of Parameter Space (TRICEPS). Winner of SC'03 HPC Challenge Competition (Most Innovative Data-Intensive Application), November 2003.

356. The Pico Framework. `http://www.picocontainer.org`.

357. R. Pike and D. M. Ritchie. The Styx Architecture for Distributed Systems. *Bell Labs Technical Journal*, 4(2):146–152, April-June 1999.

358. C. Pinchak, P. Lu, and M. Goldenberg. Practical Heterogeneous Placeholder Scheduling in Overlay Metacomputers: Early Experiences. In *JSSPP*, pages 205–228. Springer, Berlin, 2002.

359. B. Plale, J. Alameda, B. Wilhelmson, D. Gannon, S. Hampton, A. Rossi, and K. Droegemeier. Active Management of Scientific Data. *IEEE Internet Computing*, 9(1):27–34, 2005.

360. B. Plale, D. Gannon, Y. Huang, G. Kandaswamy, S. L. Pallickara, and A. Slominski. Cooperating Services for Data-Driven Computational Experimentation. *Computing in Science and Enggineering*, 7(5):34–43, 2005.

361. V. D. Pope, M. L. Gallani, V. J. Rowntree, and R. A. Stratton. The Impact of New Physical Parametrizations in the Hadley Centre Climate Model— HadAM3. Technical report, Bracknell, Berkshire, UK, Hadley Centre for Climate Prediction and Research, 2002.

362. J. Postel and J. Reynolds. File Transfer Protocol (FTP). *Internet RFC 959*, October 1985.

363. T. Pratt and M. Zelkowitz. *Programming Languages—Design and Implementation*. Prentice Hall, Englewood Cliffs, NJ, 3rd edition, 1999.

364. S. L. Price. The Computational Prediction of Pharmaceutical Crystal Structures and Polymorphism. *Advanced Drug Delivery Reviews*, 56(3):301–319, 2004.

365. R. Prodan and T. Fahringer. Dynamic Scheduling of Scientific Workflow Applications on the Grid Using a Modular Optimisation Tool: A Case Study. In *20th Symposium of Applied Computing*, pages 687–694. ACM Press, Madison, WI, 2005.

366. Ptolemy II. See Web site at `http://ptolemy.eecs.berkeley.edu/ptolemyII`.

367. R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed Resource Management for High Throughput Computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC7)*, Chicago, IL, July 1998. IEEE Computer Society, NY.

368. Reality Grid Project. `http://www.realitygrid.org/`.

369. W. Reisig. *Primer in Petri Net Design*. Springer-Verlag, New York, 1992.

370. W. Reisig. *Elements of Distributed Algorithms: Modeling and Analysis with Petri Nets*. Springer-Verlag, New York, 1998.

371. M. Resch, D. Rantzau, and R. Stoy. Metacomputing Experience in a Transatlantic Wide Area Application Testbed. *Future Generation Computer Systems*, 15(5–6):807–816, 1999.

372. M. P. Robertson, N. Caithness, and M. H. Villet. A PCA-based Modelling Technique for Predicting Environmental Suitability for Organisms from Presence Records. *Diversity & Distributions*, 7:15–27, 2001.

373. A. Rowe, D. Kalaitzopoulos, M. Osmond, M. Ghanem, and Y. Guo. The discovery net system for high throughput bioinformatics. *Bioinformatics*, 19(90001):225i–231, 2003.

374. SAGA Research Group (GGF). See Web Site at `https://forge.gridforum.org/projects/saga-rg/`.

375. SAMRAI: Structured Adaptive Mesh Refinement Application Infrastructure. `http://www.llnl.gov/CASC/SAMRAI/`.

376. E. Saxon, B. Baker, W. Hargrove, F. Hoffman, and C. Zganjar. Mapping Environments at Risk Under Different Global Climate Change Scenarios. *Ecology Letters*, 8(1):53–60, 2005.

377. Southern California Earthquake Center (SCEC). `http://www.scec.org/`.

378. SCEC Community Modeling Environment (SCEC/CME) Project. `http://www.scec.org/cme`.

379. D. J. Schlegel, D. Finkbeiner, and M. Davis. Maps of Dust Infrared Emission for Use in Estimation of Reddening and Cosmic Microwave Background Radiation Foregrounds. *The Astrophysical Journal*, 500:525, 1998.

380. E. Schnetter, S. H. Hawley, and I. Hawke. Evolutions in 3D Numerical Relativity Using Fixed Mesh Refinement. *Classical and Quantum Gravity*, 21(6):1465–1488, 2004.

381. M. Senger, P. Rice, and T. Oinn. Soaplab—A Unified Sesame Door to Analysis Tools. In *Proceedings of UK e-Science All Hands Meeting*, pages 509–513, September 2003.

382. S. Shirasuna. X Baya Workflow Composer. `http://www.extreme.indiana.edu/xgws/xbaya`.

383. M. Siddiqui, A. Villazon, J. Hofer, and T. Fahringer. GLARE: A Grid Activity Registration, Deployment and Provisioning Framework. In *Supercomputing Conference*. ACM Press, Madison, WI, 2005.

384. Y. L. Simmhan, B. Plale, and D. Gannon. Performance Evaluation of the Karma Provenance Framework for Scientific Workflows. In *International Provenance and Annotation Workshop (IPAW)*. Springer, Berlin, 2006.

385. Y. L. Simmhan, B. Plale, and D. Gannon. Resource Catalog: An Information Service for Community Resources in LEAD. Technical Report 002, Linked Environments for Atmospheric Discovery, 2006.

386. G. Singh, S. Bharathi, A. L. Chervenak, E. Deelman, C. Kesselman, M. Manohar, S. Patil, and L. Pearlman. A Metadata Catalog Service for Data Intensive Applications. In *SuperComputing*, page 33. IEEE Computer Society, Washington, DC, 2003.

387. M. F. Skrutskie, R. M. Cutri, R. Stiening, M. D. Weinberg, S. Schneider, J. M. Carpenter, C. Beichman, R. Capps, T. Chester, J. Elias, J. Huchra, J. Liebert, C. Lonsdale, D. G. Monet, S. Price, P. Seitzer, T. Jarrett, J. D. Kirkpatrick, J. E. Gizis, E. Howard, T. Evans, J. Fowler, L. Fullmer, R. Hurt, R. Light, E. L. Kopan, K. A. Marsh, H. L. McCallon, R. Tam, S. Van Dyk, and S. Wheelock. The Two Micron All-Sky Survey(2MASS). *Astronomy Journal*, 131(1163):1163–1183, 2006.

388. E. Smith and P. Anderson. Dynamic Reconfiguration for Grid Fabrics. In *5th IEEE/ACM International Workshop on Grid Computing*, pages 86–93. IEEE Computer Society, New York, November 2004.

389. M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*. The MIT Press, Cambridge, MA, 1996.

390. Simple Object Access Protocol (SOAP) 1.2. Technical report, W3C, 2003.

391. J. Soberón and A. T. Peterson. Interpretation of Models of Fundamental Ecological Niches and Species' Distributional Areas. *Biodiversity Informatics*, 2:1–10, 2005.

392. The Spring Project. `http://www.springframework.org`.

393. The Scalable Robust Self-organizing Sensor (SRSS) network project. `http://pf.itd.nrl.navy.mil/srss/`.

394. B. R. Stein and J. Wieczorek. Mammals of the World: MaNIS as an Example of Data Integration in a Distributed Network Environment. *Biodiversity Informatics*, 1:14–22, 2004.

395. L. Stein. Creating a Bioinformatics Nation. *Nature*, 417:119–120, 2002.

396. R. Stevens, A. Robinson, and C. Goble. myGrid: Personalised Bioinformatics on the Information Grid. In *11th International Conference on Intelligent Systems in Molecular Biology*, volume 19(1) of *Bioinformatics*, pages i302–i304. Oxford University Press, Oxford, June 2003.

397. R. Stevens, H. Tipney, C. Wroe, T. Oinn, M. Senger, P. Lord, C. Goble, A. Brass, and M. Tassabehji. Exploring Williams Beuren Syndrome Using ^my Grid. In A. Bateman and A. Valencia, editors, *Intelligent Systems for Molecular Biology (ISMB) 2004*, volume 20 of *Bioinformatics*, pages i303–310. Oxford University Press, Oxford, 2004.

398. D. Stockwell and I. R. Noble. Induction of Sets of Rules From Animal Distribution Data: A Robust and Informative Method of Data Analysis. *Mathematics and Computers in Simulation*, 33:385–390, 1992.

399. D. Stockwell and D. Peters. The GARP Modelling System: Problems and Solutions to Automated Spatial Prediction. *International Journal of Geographical Information Science*, 13(2):143–158, 1999.

400. Sun Microsystems. Java Remote Method Invocation (RMI). `http://java.sun.com/products/jdk/rmi/`.

401. Sun StorEdge SAM-QFS. See `http://www.sun.com`.

402. C. Szyperski. *Component Software—Beyond Object-Oriented Programming*. Addison-Wesley, Reading, MA, 1998.

403. C. Szyperski and C. Pfistery. Why Objects Are Not Enough. In *Proceedings, International Component Users Conference*, Munich, Germany, 1996. SIGS.

404. B. Talbot, S. Zhou, and G. Higgins. Review of the Cactus Framework: Software Engineering Support of the Third Round of Scientific Grand Challenge Investigations, Task 4 Report - Earth System Modeling Framework Survey. `http://sdcd.gsfc.nasa.gov/ESS/esmf_tasc/Files/Cactus_b.html`.

405. Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka. Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing. *Journal of Grid Computing*, 1(1):41–51, 2003.

406. I. Taylor, M. Shields, and R. Philp. GridOneD: Peer to Peer Visualization using Triana: A Galaxy Formation Test Case. In *UK e-Science All Hands Meeting*, 2002.

407. I. Taylor, M. Shields, I. Wang, and A. Harrison. Visual Grid Workflow in Triana. *Journal of Grid Computing*, 3(3-4):153–169, September 2005.

408. I. Taylor, M. Shields, I. Wang, and O. Rana. Triana Applications within Grid Computing and Peer to Peer Environments. *Journal of Grid Computing*, 1(2):199–217, 2003.

409. I. Taylor, I. Wang, M. Shields, and S. Majithia. Distributed Computing with Triana on the Grid. *Concurrency and Computation: Practice and Experience*, 17(9):1197–1214, 2005.

410. I. J. Taylor, O. F. Rana, R. Philp, I. Wang, and M. S. Shields. Supporting Peer-2-Peer Interactions in the Consumer Grid. In *Eighth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'03)*, pages 3–14. IEEE Computer Society, New York, April 2003.

411. I. J. Taylor, M. S. Shields, I. Wang, and R. Philp. Distributed P2P Computing within Triana: A Galaxy Visualization Test Case. In *17th International Parallel and Distributed Processing Symposium (IPDPS 2003)*, pages 16–27. IEEE Computer Society, New York, 2003.

412. The TeraGrid Project. `http://www.teragrid.org/`.

413. Teragrid project. `http://www.teragrid.org`.

414. D. Thain, T. Tannenbaum, and M. Livny. Distributed Computing in Practice: the Condor Experience. *Concurrency and Computation: Practice and Experience*, 17(2–4):323–356, 2005.

415. S. Thakkar, J. L. Ambite, and C. A. Knoblock. Composing, Optimizing, and Executing Plans for Bioinformatics Web services. *VLDB Journal, Special Issue on Data Management, Analysis and Mining for Life Sciences*, 14(3):330–353, 2005.

416. S. Thatte. XLANG Web Services for Business Process Design. `http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm`, 2001.

417. The General Coupling Framework (GCF) Approach. `http://www.cs.man.ac.uk/cnc-bin/cnc_gcf.pl`.

418. The GridSAM Project. `http://gridsam.sourceforge.net`.

419. The Java Community Process. Portlet specification. `http://www.jcp.org/aboutJava/communityprocess/review/jsr168/`.

420. The K-Wf Grid Project. Knowledge-Based Workflow System for Grid Applications. `http://www.kwfgrid.net/`, 2006.

421. The K-Wf Grid Project. The Grid Workflow Description Language Toolbox. `http://www.gridworkflow.org/kwfgrid/gworkflowdl/docs/`, 2006.

422. The Kerrighed Project. `http://www.kerrighed.org/`.

423. The MathWorks. Matlab®. `http://www.mathworks.com/products/matlab/`.

424. The Open Mosix Project. `http://openmosix.sourceforge.net/`.

425. The Open SSI Project. `http://openssi.org/index.shtml`.

426. C. D. Thomas, A. Cameron, R. E. Green, M. Bakkenes, L. J. Beaumont, Y. C. Collingham, B. F. N. Erasmus, M. Ferreira de Siqueira, A. Grainger, L. Hannah, L. Hughes, B. Huntley, A. S. Van Jaarsveld, G. E. Midgely, L. Miles, M. A. Ortega-Huerta, A. T. Peterson, O. L. Phillips, and S. E. Williams. Extinction Risk From Climate Change. *Nature*, 427:145–148, 2004.

427. K. S. Thorne. Gravitational Radiation. In S. W. Hawking and W. Israel, editors, *Three Hundred Years of Gravitation*, chapter 9, pages 330–458. Cambridge University Press, Cambridge, 1987.

428. W. Tim Berners-Lee. Web Architecture from 50,000 feet. `http://www.w3.org/DesignIssues/Architecture.html`.

429. The Triana Project. `http://www.trianacode.org`.

430. UDDI Technical White Paper. Technical report, OASIS UDDI, September 2000.

431. UNICORE Forum. UNICORE: UNiform Interface to COmputing REsources. See Web Site at `http://www.unicore.org`.

432. University of Southern California — High Performance Computing and Communication. `http://www.usc.edu/hpcc`.

433. User Mode Linux. `http://user-mode-linux.sourceforge.net/`.

434. University of Wisconsin–Milwaukee LIGO Scientific Collaboration Group Medusa Cluster.
`http://www.lsc-group.phys.uwm.edu/beowulf/medusa`.

435. A. Vakali, B. Catania, and A. Maddalena. XML Data Stores: Emerging Practices. *Internet Computing*, 9(2):62–69, March/April 2005.

436. W. van der Aalst. Pi Calculus Versus Petri Nets: Let Us Eat Humble Pie Rather Than Further Inflate The Pi Hype. `www.bptrends.com`, 2005.

437. W. van der Aalst and A. ter Hofstede. Workflow Patterns: On the Expressive Power of (Petri-Net-Based) Workflow Languages. Technical report, Department of Technology Management, Eindhoven University of Technology, Eindhoven, The Netherlands, 2002.

510   References

438. R. A. Van Engelen and K. A. Gallivan. The gSOAP Toolkit for Web Services and Peer-to-Peer Computing Networks. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID '02)*, page 128, Washington, DC, 2002. IEEE Computer Society, New York.

439. J. Van Horn, J. Dobson, J. Woodward, M. Wilde, Y. Zhao, J. Voeckler, and I. Foster. Grid-Based Computing and the Future of Neuroscience Computation. In C. Senior, T. Russell, and M. S. Gazzaniga, editors, *Methods in Mind*, Cognitive Neuroscience. The MIT Press, Cambridge, MA (In Press), 2006.

440. The Virtual Data Toolkit. http://www.vdt.org.

441. B. Victor, F. Moller, M. Dam, and L. Eriksson. The Mobility workbench. `http://www.it.uu.se/research/group/mobility/mwb`.

442. J. E. Villacis, M. Govindaraju, D. Stern, A. Whitaker, F. Breg, P. Deuskar, B. Temko, D. Gannon, and R. Bramley. CAT: A High Performance Distributed Component Architecture Toolkit for the Grid. In *High Performance Distributed Computing*. IEEE Press, 1999.

443. G. von Laszewski. An Interactive Parallel Programming Environment Applied in Atmospheric Science. In G.-R. Hoffman and N. Kreitz, editors, *Making Its Mark, Proceedings of the 6th Workshop on the Use of Parallel Processors in Meteorology*, pages 311–325, Reading, UK, December 1996. European Centre for Medium Weather Forecast, World Scientific, Singapore.

444. G. von Laszewski. The Grid-Idea and Its Evolution. *Information Technology.*, 47(6):319–329, 2005.

445. G. von Laszewski. Java CoG Kit Workflow Concepts. *accepted for publication in Journal of Grid Computing*, 2006.

446. G. von Laszewski, K. Amin, S. Hampton, and S. Nijsure. GridAnt—White Paper. Technical report, Argonne National Laboratory, Argonne, IL., July 2002.

447. G. von Laszewski, S. Fitzgerald, I. Foster, C. Kesselman, W. Smith, and S. Tuecke. A Directory Service for Configuring High-Performance Distributed Computations. In *Proceedings of the 6th IEEE Symposium on High-Performance Distributed Computing*, pages 365–375, Portland, OR, August 1997. IEEE Computer Society Press.

448. G. von Laszewski, I. Foster, J. Gawor, W. Smith, and S. Tuecke. CoG Kits: A Bridge between Commodity Distributed Computing and High-Performance Grids. In *ACM Java Grande 2000 Conference*, pages 97–106, San Francisco, CA, June 2000. ACM Press.

449. G. von Laszewski and M. Hategan. Grid Workflow - An Integrated Approach. Draft Paper, 2005.

450. G. von Laszewski and D. Kodeboyina. A Repository Service for Grid Workflow Components. In *International Conference on Autonomic and Autonomous Systems International Conference on Networking and Services*. IEEE, October 2005.

451. G. von Laszewski, K. Mahinthakumar, R. Ranjithan, D. Brill, J. Uber, K. Harrison, S. Sreepathi, and E. Zechman. An Adaptive Cyberinfrastructure for Threat Management in Urban Water Distribution Systems. Technical report, Argonne National Laboratory, Argonne, IL, Jan. 2007. To be submitted.

452. G. von Laszewski, M.-H. Su, J. A. Insley, I. Foster, J. Bresnahan, C. Kesselman, M. Thiebaux, M. L. Rivers, S. Wang, B. Tieman, and I. McNulty. Real-Time Analysis, Visualization, and Steering of Microtomography Experiments

at Photon Sources. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, TX, March 1999. Society for Industrial and Applied Mathematics.

453. G. von Laszewski, T. Trieu, P. Zimny, and D. Angulo. The Java CoG Kit Experiment Manager. Technical report, Argonne National Laboratory, Argonne, IL, June 2005.

454. W3C. Semantic Web Activity Statement. `http://www.w3.org/2001/sw/Activity`.

455. W3C Recommendation: Architecture of the World Wide Web, Volume One. `http://www.w3.org/TR/webarch`.

456. W3C. Web Services Addressing (WS-Addressing). `http://www.w3.org/2002/ws/addr/`.

457. W3C. Web Services Description Language (WSDL) Version 2.0 Part 2: Adjuncts. `http://www.w3.org/TR/2005/WD-wsdl20-adjuncts-20050803`.

458. W3C. Web Services Choreography Description Language (WS-CDL) Version 1.0. `http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/`, 2004.

459. P. A. Walker and K. D. Cocks. HABITAT: A Procedure for Modelling a Disjoint Environmental Envelope for a Plant or Animal Species. *Global Ecology and Biogeography Letters*, 1:108–118, 1991.

460. I. Wang. P2PS (Peer-to-Peer Simplified). In *Proceedings of 13th Annual Mardi Gras Conference—Frontiers of Grid Applications and Technologies*, pages 54–59. Louisiana State University, February 2005.

461. G. Wasson and M. Humphrey. Exploiting WSRF and WSRF.NET for Remote Job Execution in Grid Environments. In *19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*. IEEE Computer Society, New York, 2005.

462. Web Tools Project. Eclipse Web Tools Platform Project. See Web site at `http://www.eclipse.org/webtools/`.

463. V. Welch, F. Siebenlist, I. Foster, J. Bresnahan, K. Czajkowski, J. Gawor, C. Kesselman, S. Meder, L. Pearlman, and S. Tuecke. Security for Grid Services. In *Twelfth International Symposium on High Performance Distributed Computing (HPDC-12)*, pages 48–57. IEEE Computer Society Press, New York, 2003.

464. Workflow Management Research Group (GGF). `https://forge.gridforum.org/projects/wfm-rg/`.

465. Wikipedia. Choreography—wikipedia, the free encyclopedia, 2006. `http://en.wikipedia.org/w/index.php?title=Choreography&oldid=33366853` (Online; accessed January 18, 2006).

466. Wikipedia. Orchestration — wikipedia, the free encyclopedia, 2006. `http://en.wikipedia.org/w/index.php?title=Orchestration&oldid=34882858` (Online; accessed January 18, 2006).

467. Wikipedia. Petri net—wikipedia, the free encyclopedia, 2006. `http://en.wikipedia.org/w/index.php?title=Petri_net&oldid=35563598` (Online; accessed January 18, 2006).

468. E. O. Wiley, K. M. McNyset, A. T. Peterson, C. R. Robins, and A. M. Stewart. Niche Modeling and Geographic Range Predictions in the Marine Environment Using a Machine-Learning Algorithm. *Oceanography*, 16:120–127, 2003.

469. M. D. Wilkinson, D. Gessler, A. Farmer, and L. Stein. The BioMOBY Project Explores Open-Source, Simple, Extensible Protocols for Enabling

Biological Database Interoperability. In *Virtual Conference on Genomics and Bioinformatics*, volume 3, pages 16–26, 2003.

470. B. Willke et al. The geo 600 gravitational wave detector. *Classical and Quantum Gravity*, 19(7):1377–1387, 2002.

471. D. Willock, S. Price, M. Leslie, and C. Catlow. The Relaxation of Molecular Crystal Structures Using a Distributed Multipole Electrostatic Model. *Journal of Computational Chemistry*, 16(5):628–647, 1995.

472. R. Wolski, N. T. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Future Generation Computer Systems*, 15:757–768, 1999.

473. K. Wolstencroft, T. Oinn, C. Goble, J. Ferris, C. Wroe, P. Lord, K. Glover, and R. Stevens. Panoply of Utilities in Taverna. In S. J. Cox and D. W. Walker, editors, *UK e-Science All Hands Meeting, 2005*, pages 471–475. CD Rom Proceedings, 2005.

474. S. Woodman, S. Parastatidis, and J. Webber. Sequencing Constraints SSDL Protocol Framework. Technical Report CS-TR-903, University of Newcastle, 2005.

475. R. P. Woods. Automated Image Registration. `http://bishopw.loni.ucla.edu/AIR5/`.

476. W. Woods. What's in a Link: Foundations for Semantic Networks. In D. Bobrow and A. Collins, editors, *Representation and Understanding: Studies in Cognitive Science*. Academic Press, New York, 1975.

477. A. Woolf, R. Cramer, M. Gutierrez, K. van Dam, S. Kondapalli, S. Latham, B. Lawrence, R. Lowry, and K. O'Neill. Semantic Integration of File-based Data for Grid Services. In *Workshop on Semantic Infrastructure for Grid Computing Applications*. IEEE, Piscataway, NJ, USA, 2005.

478. The Workflow Management Coalition. `http://www.wfmc.org/`.

479. Workflow Management Coalition. Terminology & Glossary. Technical report, WfMC, 1999. `http://www.wfmc.org/`.

480. C. Wroe, C. Goble, M. Greenwood, P. Lord, S. Miles, J. Papay, T. Payne, and L. Moreau. Automating Experiments Using Semantic Data on a Bioinformatics Grid. *IEEE Intelligent Systems*, 19(1):48–55, 2004.

481. C. Wroe, R. Stevens, C. Goble, A. Roberts, and M. Greenwood. A Suite of DAML+OIL Ontologies to Describe Bioinformatics Web Services and Data. *The International Journal of Cooperative Information Systems*, 12(2):597–624, 2003.

482. Web Services Description Language (WSDL) 1.1. Technical report, W3C, 2001.

483. WSRF::Lite — Perl Grid Services. `http://www.sve.man.ac.uk/Research/AtoZ/ILCT`.

484. Web services for remote portlets. `http://www.oasis-open.org`.

485. XML Process Definition Language (XPDL). Technical report WFMCTC-1025, Workflow Management Coalition, Lighthouse Point, Fl, USA, 2002.

486. M. Xue, K. K. Droegemeier, and V. Wong. The Advanced Regional Prediction System (ARPS)x – A Multi-scale Nonhydrostatic Atmospheric Simulation and Prediction Model. Part I: Model Dynamics and Verification. *Meteorology and Atmospheric Physics*, 75(3–4):161–193, 2000.

487. L. Young. Scheduling Componentised Applications On A Computational Grid. MPhil Transfer Report, 2004.

488. L. Young, A. McGough, S. Newhouse, and J. Darlington. Scheduling Architecture and Algorithms within the ICENI Grid Middleware. In *UK e-Science All Hands Meeting*, pages 5–12, Nottingham, UK, September 2003.

489. J. Yu and R. Buyya. A Novel Architecture for Realizing Grid Workflow using Tuple Spaces. In *Fifth IEEE/ACM International Workshop on Grid Computing*, pages 119–128. IEEE Computer Society Press: Los Alamitos, CA, 2004.

490. J. Yu and R. Buyya. A Taxonomy of Workflow Management Systems for Grid Computing. Technical Report GRIDS-TR-2005-1, Grid Computing and Distributed Systems Laboratory, University of Melbourne, March 2005.

491. O. Zaki, E. Lusk, W. Gropp, and D. Swider. Toward Scalable Performance Visualization with Jumpshot. *International Journal of High-Performance Computing and Applications*, 13(3):277–288, 1999.

492. K. Zhang, K. Damevski, V. Venkatachalapathy, and S. Parker. SCIRun2: A CCA Framework for High Performance Computing. In *Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004)*, pages 72–79, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

493. H. Zhao and R. Sakellariou. An Experimental Investigation into the Rank Function of the Heterogeneous Earliest Finish Time Scheduling Algorithm. In H. Kosch, L. Boszormenyi, and H. Hellwagner, editors, *Euro-Par*, volume 2790 of *Lecture Notes in Computer Science*, pages 189–194. Springer-Verlag, Berlin, 2003.

494. J. Zhao, C. Wroe, C. Goble, R. Stevens, D. Quan, and M. Greenwood. Using Semantic Web Technologies for Representing e-Science Provenance. In *3rd International Semantic Web Conference (ISWC2004)*, volume 3298 of *Lecture Notes in Computer Science*, pages 92–106. Springer-Verlag, Berlin, 2004.

495. L. Zhao, P. Chen, and T. Jordan. Strain Green's Tensors, Reciprocity and their Applications to Seismic Source and Structure Studies. *Bulletin of the Seismological Society of America*, 96(5):1753–1763, 2006.

496. Y. Zhao, J. Dobson, I. Foster, L. Moreau, and M. Wilde. A Notation and System for Expressing and Executing Cleanly Typed Workflows on Messy Scientific Data. *SIGMOD Record*, 34(3):37–43, 2005.

Index