FIRB "Grid.it" WP8 Working Paper

Grid-based PSE Toolkits for Multidisciplinary Applications

Mario Cannataro[3], Carmela Comito[1], Antonio Congiusta[1,2],

Gianluigi Folino[2], Carlo Mastroianni[2], Andrea Pugliese[1,2],

Giandomenico Spezzano[2], Domenico Talia[1], Pierangelo Veltri[3]

[1] *DEIS – Università della Calabria*

[2] *ICAR-CNR*

[3] *Università "Magna Græcia" di Catanzaro*

Table of Contents

1. Introduction

The goal of this report is to introduce Problem Solving Environments (PSEs), present the main features of PSE Toolkits, and propose a general component-based architecture for Grid-based PSE Toolkits.

Problem Solving Environments (PSE) have been investigated over the past 30 years. In the pioneering work “A system for interactive mathematics”, Culler and Fried in 1963 initiated to investigate automatic software systems for solving mathematical problems with computers focusing primarily on applications issues instead of programming issues. At that time, and for many years, the term applications indicate scientific and engineering applications that are generally solved using mathematical solvers or scientific algorithms managing vectors and matrices. More recently, PSE for industry, commercial, and business applications are becoming to appear.

Despite the time passed from that early research work and several research activities from then there in not a precise definition of what a PSE is. The following well-known definition was given by Gallopoulos, Houstis, and Rice: "*A PSE is a computer system that provides all the computational features necessary to solve a target class of problems*. …. *PSEs use the language of the target class of problems* " [Gall94]. They tried to specify the main components of a PSE by defining the following equation:

PSE = Natural Language + Solvers + Intelligence + Software Bus

where

- Natural Language is specific for the application domain and natural for domain experts,
- Solvers are software components that do the computational work and represent the basic elements around whose a PSE is built,
- Intelligence is useful mainly in the following phases: resource location, input wizards (with recommender interfaces), scheduling, and analysis of results,
- Software Bus is the infrastructure that supports the PSE and its use for solving scientific problems.

Other definitions that partially agree and partially disagree with the original definition have been given in the latest five year.

According to Walker et al., "*A PSE is a complete, integrated computing environment for composing, compiling, and running applications in a specific area*" [Walk00].

Schuchardt and his co-authors defined PSEs as "*problem-oriented computing environments that support the entire assortment of scientific computational problem-solving activities ranging from problem formulation to algorithm selection to simulation execution to solution visualization. PSEs link a heterogeneous mix of resources including people, computers, data, and information within a seamless environment to solve a problem*” [Schu02].

In 2003 Cunha describes a PSE as "*an integrated environment for solving a class of related problems in an application domain; easy to use by the end-user; based on state-of-the-art algorithms. It must provide support for problem specification, resource management, execution services*" [Cunh03].

As we can see, the listed definitions agree on the basic features of PSEs, but differ in the way they envision how PSEs can be composed and used. Moreover, the existence of several different definitions demonstrates that the term PSE is perhaps too generic and not completely investigated for reaching a full consensus in the scientific community.

The main motivation for developing PSEs is that they provide software tools and expert assistance to the computational scientist in a user-friendly environment, allowing more rapid prototyping of ideas and higher research productivity. A PSE provides users with an interface with high-performance computing resources, relieving them from hardware/software details and letting them concentrate on the application. *Collaboration*, *visualization,* and *knowledge* are the three main aspects distinguishing a PSE from a mere interface. All these three properties must be included in a PSE for considering it a high-level environment for solving complex problems.

As specified in the PSE definitions given above, a PSE is typically aimed at a particular computing domain. An advancement of the PSE concept is the PSE Toolkit concept. A PSE Toolkit can be defined as "*a group of technologies within a software architectures that can build multiple PSEs*." A PSE Toolkit allows for creation of meta-applications from preexisting modules to meet the needs of specific solutions. It may support the building of solving environments on different domains allowing designers to develop multidisciplinary PSEs.

PSEs can benefit from advancements in hardware/software solutions achieved in parallel and distributed systems and tools. One of the most interesting model in the area of parallel and distributed computing is the Grid concept. Grid computing represents an opportunity for PSE designers and users. It can provide a high-performance infrastructure for running PSEs and, at the same time, a valuable source of resources that can be integrated in PSEs and PSE Toolkits. Grids can be used for dynamically building geographically distributed collaborative problem solving environments and Grid-aware PSEs can search and use dispersed high performance computing, networking, and data resources.

The remainder of the report is organized as follows. In the rest of the section we give a review of existing PSEs, PSEs Toolkits, and Grid-based PSE Toolkits. Section 2 discusses main issues in designing Grid-based PSE Toolkits. Section 3 proposes a reference architecture for a PSE Toolkit and describes its basic components and their interfaces. Moreover, that section discusses main aspects in designing component-based PSE Toolkits such as programming models, workflow design, data management and information services, resource management and scheduling, and steering. Section 4 discusses some open issues like performance models, Grid services, ontologies, Agents, Security, Grid Portals, and Application domain description.

1.1. Problem Solving Environments

In this section we briefly survey some existing problem solving environments. As said before, the design of a PSE must tackle many different issues; the PSEs we mention in this section cover a broad range of approaches to architectural design, communication paradigm, extensibility w.r.t. other tools, support for parallel computation, and so on. At the end of the section, in Table 1, we summarize the main features of the surveyed PSEs.

MATLAB

MATLAB [Math03] is a programming language-based PSE for matrix computations; it is probably the most widely used PSE for computational science, so that many University programs assume its availability to students.

MATLAB, originally designed as a high-level interface to numerical linear algebra computations, is now available on most computing platforms, and includes toolboxes for control systems, simulation, databases, finance, fuzzy logic, signal processing, optimization, partial differential equations, statistics, symbolic mathematics, etc. Its design incorporates tools for interconnection with other systems, e.g. it makes it possible to use external Fortran or C modules. The benefits of this extensibility are manifold: for instance, there is an enormous amount of Fortran scientific libraries available, and it is possible to use mature compiler technology for code optimization in order to obtain high performance on the underlying computing platform.

MATLAB provides interfaces to a number of generic facilities such as Web browsers and ActiveX controls; through a *two-way interconnection*, moreover, MATLAB can be used as a computational engine from an external program. More recently, MATLAB has also been extended to handle parallel computation through exploitation of its programming language's operator overloading feature.

PELLPACK

PELLPACK is an evolution of the *ELLPACK* PSE [Pell03]. *ELLPACK* was designed to support the solution of second-order elliptic PDEs in two and three dimensions and to evaluate software for such computations. It follows a modular programming paradigm which is supported by a domain-specific PDE language and a variety of elliptic PDE solvers. The PDE language interface allows the user to develop high-level programs that can be used to solve nonlinear and time-dependent PDEs.

PELLPACK provides many PDE solvers and facilities for parallel processing, along with powerful discretization techniques and an advanced GUI. In PELLPACK, a problem is represented by the PDE objects involved: PDE model or equations, domain, conditions on the domain boundary, solution methods, and output requirements.

PELLPACK has a software architecture consisting of five main layers. The GUI allows to specify the PDE problem and supports the solution process and various postprocessing analyses, with the help of a knowledge-based system assisting users; a translation layer, for transforming specifications into a high-level PDE-oriented language; the compilation layer, where a preprocessor compiles the PELLPACK language into a Fortran driver program and then compiles and links it with numerical libraries containing the user-specified solver methods; the execution environment,

which assists users in running programs, and is responsible for locating and allocating hardware and software resources and for managing data movement operations; the solver library, provided with suitable interfaces that allow them to be composed and to interact with the whole system.

PELLPACK has been used for solving problems in physics (liquid crystal droplets, proton flux propagation), thermal field analysis, fluid dynamics, semiconductors, geophysical research, electromagnetic field analysis, thermo-elasticity, structural analysis, and other scientific and engineering applications.

An Internet-based client-server implementation of PELLPACK is *WebPDELab*. WebPDELab adds to PELLPACK further layers, specifically addressing network-based computing issues. The WebPDELab server is accessed from a Web site, by using a Java-enabled browser. Moreover, a PELLPACK GUI is made available through a Java-based remote display system using the TCP/IP protocol. A WebPDELab manager controls all the user-server interactions. Users may download files generated by PELLPACK to their own machines before terminating their computing sessions, and they may upload files to WebPDELab at the start of subsequent sessions.

ECCE

ECCE [Schu02] is part of the *Molecular Science Software Suite* (*MS3*) [Mol03] developed at the Pacific Northwest National Laboratory. MS3 is an integrated suite for chemical applications that makes use of advanced computational chemistry techniques on high-performance, parallel computing systems. MS3 consists of three components: *NWChem* providing advanced computational chemistry techniques, *ParSoft* providing efficient and portable libraries and tools that enable NWChem to run on a wide variety of parallel computing systems, and ECCE that is a suite of tools integrated within a PSE. Such tools are used in many phases: management of projects, construction of complex molecules and basis sets, selection of input options, distributed execution of computational models, real-time monitoring, and post-run analysis.

The ECCE architecture addresses component-based application development, distributed code execution, and data and metadata management. Each ECCE component is designed to assist the user with a single aspect of the research process. The architecture is rather complex but it provides many benefits, including the capability to deploy tools independently while simplifying the development process. ECCE uses a publish/subscribe event system to coordinate activities between applications. ECCE jobs can be launched to UNIX workstations or any cluster or supercomputer running different batch systems. A data management component for persistently tracking the data and metadata associated with the chemistry research process underlies the ECCE framework. ECCE is deployed at multiple sites around the world. Its users have access to a wide variety of computing environments ranging from major government-run computing centers with large numbers of resources, to individual researchers running primarily on their desktops, to commercial companies running entirely behind a firewall.

CAMEL

CAMEL [Digr96] is a software environment based on the cellular automata model, and designed to support the development of high-performance applications, which

has been successfully used for many problems such as the simulation of the lava flows of the last Etnean eruption and other fluid flow models, for image processing, freeway traffic simulation and combustion process modeling. In CAMEL, cellular automata models are defined through a programming tool called *CARPET* (*CellulAR Programming EnvironmenT*). CARPET uses a high-level sequential language related to C with some additional constructs to describe the rules of the transition function of a single cell of a cellular automaton. The main features of CARPET are the possibility to describe the cell state as a structured variable composed of typed fields (substates), and the simple definition of complex neighbourhoods that can also be time-dependent. Finally, CAMEL provides a user interface that allows users to steer simulations and graphically visualize their output.

CAMEL exploits the computing power of a highly parallel computer hiding architectural issues from users in programming computational science applications. It has been implemented on a message-passing MIMD parallel computer, and this maked it very scalable w.r.t. the number of processors or the size of automata. The architecture comprises three main components: the *Master Node*, the *Graphic Node*, and the *Parallel Engine*. In particular, the Parallel Engine consists of a number of identical nodes on which the cellular automaton is executed. The Master Node contains a *Controller*, which coordinates the file system, the UI, and the individual processes implementing *Macrocells*, i.e. clusters of elementary cells that constitute a partition of a cellular automaton. Each macrocell executes the state transition function defined by a user for all the cells that compose a partition. On the Graphic Node, the *Graphical Interface process* displays on a screen the state of the simulated system, along with information such as the displayed substates, the current iteration step, the visualization step, and the step of saving. Finally, since for a large class of cellular automata problems the areas of active cells are restricted to one or few domains, CAMEL uses a load balancing strategy that also avoids to compute the next state of cells that belong to an inactive region.

SCIRun

SCIRun [Scir03] was initially an architecture designed to solve specific problems in computational medicine; afterwards, it was made applicable in other computational science and engineering problem domains. To this end, SCIRun has been made extensible through *bridging* constructs; also entire derived systems appeared, such as *BioPSE* and *Uintah* (described in the following). Three types of bridging are provided in SCIRun: (*i*) *assimilation* bridging, where existing data or functionality are rewritten for the SCIRun system; (*ii*) *library* bridging, where calls are made into third-party libraries; (*iii*) *I/O* bridging, where data are exchanged through files, sockets, or databases, and external functionality are invoked through system calls.

The SCIRun architecture is now the basis for a suite of scientific PSEs that allow for the interactive construction, debugging, and steering of large-scale scientific computations. SCIRun allows the user to interactively control scientific simulations while the computation is in progress. This control allows the user, for example, to vary boundary conditions, model geometries, and/or various computational parameters during simulation. SCIRun uses a data-flow programming model; a *dataflow network* is assembled by connecting together *modules*, such that each module performs a computational or visualization algorithm. A network is constructed in a visual programming environment that allows for a convenient and natural approach to both application construction and runtime steering.

In SCIRun, task parallelism is implemented by simultaneously executing multiple modules according to the dataflow graph. Moreover, it allows for the explicit parallelization of various modules in a data-parallel (SPMD) fashion. A set of worker threads are mapped to various processors and cooperate in accomplishing the function of the module. SCIRun uses an object-oriented design, leveraging a powerful toolbox of C++ classes that have been tuned for scientific computing and operation in a multi-threaded environment. SCIRun uses threads to facilitate parallel execution, to allow user interaction while computation is in progress, and to allow the system to change variables without interrupting a simulation. A layer provides a simple C++ interface to threads and provides abstraction from the standard used to implement them.

	Application domain	**Main features**
MATLAB	Matrix computations	• *Toolboxes* for different applications • Interfaceable with libraries written in other languages, or usable as underlying computational engine • Native support for parallel computation
PELLPACK	Second-order elliptic partial differential equations	• Facilities for parallel processing • Knowledge base for helping users • Execution environment managing resource allocation and data movement • *WebPDELab*: Internet-based client-server version
ECCE	Computational chemistry	• Part of the *Molecular Science Software Suite* (*MS3*) • Component-based application-development • Publish/subscribe event system for coordinating applications • Data/metadata tracking component
CAMEL	Cellular automata	• Cellular automata modeling based on a high-level programming language • Complete abstraction from architectural issues • Implementation on a message-passing MIMD parallel computer • High scalability w.r.t. the number of processing units and automata states
SCIRun	Computational science and engineering	• Extensibility through *assimilation* (rewriting of functionalities), *library* (direct calls), and *I/O* (standard I/O mechanisms) *bridging* • Object-oriented design • *Dataflow network* of computation/visualization *modules* • Inter- and intra-module parallelization • *BioPSE*: bioelectric field-specific extension • *Uintah*: CCA programming model and support for shared-memory/message-passing models

Table 1. Problem-solving environments

Two interesting PSE built atop SCIRun are *BioPSE* and *Uintah*. BioPSE is designed to support bioelectric field research problems. The BioPSE package consists of modules and data structures specifically customized for bioelectric volume conductor problems. These pieces include modules for bioelectric field finite-element approximation, boundary condition assignment, and inverse source localization. The major undertaking of the BioPSE project, however, has been to apply solid software

engineering approaches to make the SCIRun code and underlying software architecture more robust. Uintah extends the SCIRun architecture by adding a CCA's interchangeable component programming model, and support for running under a mixed shared-memory/message-passing model. These additions make Uintah a scalable and high-performance system capable of solving large-scale complex scientific problems.

1.2. PSE Toolkits

The PSE environments discussed in the previous paragraph are all tailored to a specific application domain. However, in many cases these tools replicate the same functionalities and the same solutions to solve similar problems in different domains. Indeed, only limited parts of PSE environments are strictly related to specific domains and hence not replicable from one system to another; many other parts and components are common and should be reused.

Several PSE environments were designed with the purpose of providing users a set of tools for building multiple PSE systems for a large range of application domains. These environments are referred to as "PSE Toolkits". In designing a PSE Toolkit, developers provide the most important components for building up PSEs and a way to compose them when a specific PSE must be developed. Therefore, PSE Toolkit allows for developing domain-specific PSEs, thus creating meta-applications from pre-existing modules on different domains. In the following we introduce some widely adopted PSE Tookits.

NetSolve

The NetSolve project [Nets03], underway at the University of Tennessee at Knoxville and the Oak Ridge National Laboratory, was initially tailored to alleviate domain scientists to the tedium of installing and managing the software on heterogeneous machines. Today, NetSolve is evolved into a PSE tollkit providing uniform access to a wide assortment of software libraries.

NetSolve is a client/agent/server software in which the client issues requests to agents who allocate servers to satisfy those requests; the server(s) then receives inputs for the problem, does the computation and returns the output parameters to the client. NetSolve allows clients to use limitless software resources and to gain access to remote computers with complete opacity.

The client interface is available in C, Fortran, Matlab. Interfaces allow the user to control execution of remote procedures, and provide two basic functions to launch synchronous (blocking) or asynchronous (non-blocking) requests.

The NetSolve agent keeps track of what resources are available and on which servers they are located. A database maps software resources to hardware components, thus providing a complete picture of the NetSolve system. To maintain the database, each server registers to an agent, and sends to that agent a "problem description" for each problem it can solve. The NetSolve acts as a resource broker: it uses both static and dynamic information (speed and number of processors, solution algorithms, server loads, network delays, input data sizes etc.) to decide which server component should be used to serve a request.

Three major features are mandatory for NetSolve servers: uniform access to software, configurability, and managing of software installation on heterogeneous computers.

Applications that have taken advantage of the NetSolve paradigm range from mathematical solvers to microbiology, image visualization, socio-economical applications etc.

PSEWare

The PSEWare project [Psew03], funded by the U.S. National Science Foundation, involves the California Institute of Technology, the Indiana University, The Los Alamos National Laboratory, the Drexel University, and the University of California at Irvine. The project is focused on symbolic specifications of applications, methods of reuse of object structures for user interfaces and parallel execution, component technologies for PSEs, and collaboration technologies for problem solving.

The view of PSEWare is that scientists and engineers should be able to specify a problem symbolically with the notation that they use in communication with each other. PSEs should help them to refine their symbolic specification to efficient parallel object-oriented programs. Object-oriented interfaces facilitate the reuse of components for large classes of scientific applications, and the collaboration among a distributed group of users working at the same application.

PSEWare research is focused on methods and tools to develop PSEs as opposed to PSEs for a specific application. Generic PSEWare components and tools give non experts in computer science the ability to rapidly and easily construct their own PSEs.

PSEWare application areas include cosmology modeling, sparse linear system analysis, collaboration systems for mixed symbolic-numeric computing, soliton exploration.

VDCE

The Virtual Distributed Computing Environment [Topc97] is a meta-computing environment developed at Syracuse University. VDCE can be used as a problem solving environment for large scale network applications, enabling users to focus on the solution approach rather than caring about technical details.

VDCE supports software development in three phases:

1. application design and development phase;
2. application configuration and scheduling phase;
3. application execution and runtime phase.

In the first phase, a program is defined as a directed graph where nodes denote computations and links denote communication and synchronization between nodes. The Application Editor module of VDCE is a web-based graphical user interface for designing application flow graphs describing parallel and distributed applications.

In the second phase each task of the application is scheduled to the best available resource. This phase is managed by the VDCE Runtime System, in particular through the Mapping Module (that individuates the candidate resources), and the Estimation Module (that estimates the performance of each candidate resource and

identifies the one which gives the best performance in terms of total execution time). The scheduling approach uses a combination of performance analysis, measurement and benchmarking techniques to estimate the execution time of a task running on VDCE hosts under varying load conditions.

In the application execution/runtime phase, the application is started, run and managed on the assigned machines. The VDCE Runtime System of VDCE sets up the execution environment for each submitted task, managing the execution to meet the requirements of the application.

The VDCE problem solving environment provides a large set of task libraries to solve applications in different domains, such as C3I (command, control and communication) applications and matrix algebra applications.

Ninf

The Ninf Project [Ninf03] aims at developing Grid technologies which allow users to access various resources such as hardware, software and scientific data on the Grid with an easy-to-use interface. The Ninf system is a Grid Remote Procedure Call (RPC) system which has been developing by the Ninf Project. The Ninf system provides RPC facilities designed to provide a programming interface similar to conventional function calls and enable the user to build Grid-enabled applications.

Ninf-G is a reference implementation of a Grid RPC system using the Globus Toolkit. Ninf-G provides Grid RPC APIs which are discussed for the standardization at the Advanced Programming Models Research Group of the Global Grid Forum.

	Application domains	Main features
NetSolve	• mathematical solvers • microbiology • image visualization • socio-economical applications	•uniform access to software libraries •resource broker functionalities •client/agent/server paradigm: agents allocate server resources requested by clients •remote execution control through client interface
PSEWare	• cosmology modeling • sparse linear system analysis • symbolic-numeric computing • soliton exploration	•symbolic specification of applications •tools for building efficient parallel object-oriented programs •reuse of components for large classes of scientific applications
VDCE	• C3I (command, control and communication) • matrix algebra applications	•support for three-phases software development: application design, scheduling and execution •support for workflow modelling •estimation and mapping modules to individuate and allocate candidate resources
Ninf	• data-intensive applications • matrix calculation • network simulation	•based on the Remote Procedure Call paradigm •access to hardware and software resources with easy-to-easy interfaces •Ninf-G is a reference implementation of a Grid RPC system using the GlobusToolkit

Table 2. PSE Toolkits

2. Towards a Grid-based PSE Toolkit

PSEs are typically designed with a specific application domain in mind. This approach simplifies the designer task and generally produces an environment that is particularly tailored for a particular application class. On the other hand, this approach does limit portability of solutions. That is, the resulted PSE cannot be generally used in different application domains without re-designing and re-implementing most or all the environment. PSE portability, extensibility, and flexibility can be provided using the PSE Toolkit model. According to this approach a general framework is provided for developing PSEs.

A PSE Toolkit is a group of technologies within some software architectures that can build multiple PSEs. In designing a PSE Toolkit, developers provide the most important components for building up PSEs and a way to compose them when a specific PSE must be developed. Therefore, a PSE toolkit allows to develop domain-specific PSEs, thus creating meta-applications from pre-existing modules on different domains. According to Walker, the main features of a PSE Toolkit are:

- Problem-oriented;
- Integrated view;
- Graphic (visual);
- Flexible and open;
- Collaborative;
- Distributed; and
- Persistent.

PSE users, which often expect to be interfaced with a tightly integrated environment, must have transparent access to dispersed and de-coupled components and resources. Handling with distributed environments is another main issue in PSEs design and use. Distributed and parallel computing systems are used for running PSEs both to get high performance and using distributed data, machines, or software modules.

Today the Grid is a high-performance distributed infrastructure that combines parallel and distributed computing systems. The Grid is a new distributed computing infrastructure whose main goal is resource sharing and coordinated problem solving in *"dynamic, multi-institutional virtual organizations".*

A Grid definition by Foster clarifies the main goal of the Grid: *"By providing scalable, secure, high-performance mechanisms for discovering and negotiating access to remote resources, the Grid promises to make it possible for scientific collaborations to share resources on an unprecedented scale, and for geographically distributed groups to work together in ways that were previously impossible".*

The role of the Grid is fundamental, since it potentially provides an enormous number of dispersed hardware and software resources that can be transparently accessed by a PSE toolkit. In Grid environments instruments may be connected to the computing,

data, and collaboration environments and all of these may be coordinated for simultaneous operation. Moreover, vast quantities of data may be received from instruments and simulations, and catalogued, archived, and published.

Through the combination of PSE Toolkit issues and Grid features exploitation for distributed PSE design we obtain the possibility to design Grid-based PSE Toolkits. Although some PSE Toolkits that use the Grid have been recently defined, the mechanisms for producing Grid-based PSEs are not yet standardized.

Objective of this section is to identify a set of guidelines and general requirements useful to define a reference architecture for a grid-based PSE toolkit. Along this direction we first give a short review of some of the major grid-based PSE toolkits, so as to illustrate their main characteristics and the different kind of approaches adopted.

2.1. Grid-based PSE Toolkits

WebFlow

The WebFlow toolkit [Akar98] is implemented as an object Web three-tier system. Tier 1 is a high-level front end for visual programming, steering, run-time data analysis, and visualization, built on top of Web and object oriented commodity standards. A distributed object-based, scalable, and reusable Web server and an object broker middleware form tier 2. Back-end services compose tier 3. In particular, high-performance services are implemented using the Globus toolkit. WebFlow provides a job broker to Globus, while Globus takes responsibility of actual resource allocation.

The WebFlow front-end allows to specify user's task in the form of an Abstract Task Descriptor (ATD). The ATD is constructed recursively and may comprise an arbitrary number of subtasks. The lowest level, or atomic, task corresponds to the atomic operation in the middle tier, such as instantiation of an object, or establishing interactions between two objects through event binding.

When specifying a task, the user does not have to indicate the resources to be used to complete the task, but instead may specify requirements that the target resource must satisfy in order to be capable of executing the job. The identification and allocation of the resources is left to the discretion of the system. Once the resources are identified, the abstract task descriptor becomes a concrete job specification.

A mesh of CORBA-based WebFlow servers (WS) currently gives the WebFlow middle tier. One of these servers, the gatekeeper server, facilitates a secure access to the system. The middle-tier services provide the means to control the life cycles of modules and to establish communication channels between them. Services provided by the middle tier include methods for submitting and controlling jobs, methods for file manipulation, methods for providing access to databases and mass-storage, as well as methods to query the status of the system, the status of the users' applications, and their components.

WebFlow applications range from land management systems to quantum simulations and gateway seamless access.

GridPort

GridPort [gPort023] is a collection of services, scripts and tools that allow developers to connect Web-based interfaces to the computational grid behind the scenes. The scripts and tools provide consistent interfaces between the underlying infrastructure and security, and are based on grid technologies such as Globus and standard Web technologies such as CGI and Perl. GridPort is an open architecture that is designed to be flexible and capable of using other grid services and technologies as these become available. In addition, GridPort is intended to provide a framework that other application developers and computational scientists can use to write their own web pages that access and use GridPort Services.

GridPort is designed so that multiple application portals share the same installation of GridPort, and inherit connectivity to the computational Grid that includes interactive services, data, file, and account management, and share a single accounting and login environment.

GridPort is the most well-known toolkit widely used for building Grid portals. The representative example is Hot-Page, which provides users with a view of distributed computing resources and allows individual machines to be examined about status (up or down), load, etc. Besides examining machines, users can access files and perform routine computational tasks. Users should use Perl when registering a new Grid application. Back-end application programs must be built in Globus API, which takes a great deal of efforts for the user.

XCAT Science Portal

The XCAT Grid Science Portal [Kris01] is an implementation of the NCSA Grid Science Portal concept, that is a problem solving environment that allow scientists the ability to program, access and execute distributed applications using grid resources which are launched and managed by a conventional Web browser and other desktop tools. In such portals, scientific domain knowledge and tools are presented to the user in terms of the application science, and not in terms of complex distributed computing protocols.

XCAT-SP is based on the idea of an “active document” which can be thought of as a “notebook” containing pages of text and graphics describing the science of a particular computational application and pages of parameterized, executable scripts. These scripts launch and manage the computation on the grid, and results are dynamically added to the document in the form of data or links to output results and event traces. XCAT-SP is a tool which allows the user to read, edit, and execute these notebook documents.

The portal is a workstation-based specialized “personal” web server, capable of executing the application scripts and launching remote grid applications for the user. The portal server can receive event streams published by the application and grid resource information published by Network Weather Service (NWS) or sensors. Notebooks can be “published” and stored in web based archives for others to retrieve and modify. The XCAT Grid Science Portal has been tested with various applications, including the distributed simulation of chemical processes in semiconductor manufacturing and collaboratory support for X-ray crystallographers.

Cactus

The Cactus Code and Computational Toolkit [Cact03] is the result of a common effort (by many computer scientists and physicists) to provide computational physicists with a flexible, modular, portable and importantly easy-to-use, programming environment for large-scale simulations. One of the design requirements for Cactus was to provide application programmers with a high level set of APIs able to hide features such as the underlying communication and data layers. These layers are implemented in modules (in Cactus terminology *thorns*), which can be chosen at runtime, using the best available technology for a given resource, e.g., MPI, PVM, pThreads, SHMEM, OpenMP for communication, or HDF5, IEEE IO, Panda IO for parallel data I/O.

Much of the Cactus architecture is influenced by the vast computing requirements of some of the main applications using the framework, including numerical relativity and astrophysics. These applications, which are being developed and run by large international collaborations, require Terabyte and Teraflop resources, and will provide an ideal test-case for developing Grid computing technologies for simulation applications.

Main features of Cactus are: (i) automatic and configurable compilation system for most machine architectures, (ii) core code and toolkits written in ANSI C for portability, (iii) parallel I/O capabilities compatible with distributed simulations, (iv) parallel checkpointing and recovery of simulations, including distributed simulations, (v) steering interface for dynamically changing the values of parameters during a simulation, (vi) existing applications already trivially Grid-enabled using the Globus MPI implementation MPICH-G2, and, (vii) existing modules to implement remote visualization (streaming data with HDF5), remote monitoring and steering of simulations (*e.g.* using a module which provides a simulation with its own web server), parallel I/O, etc.

DataCutter

DataCutter [Beyn00] is an application framework, developed at University of Maryland. It provides support for developing data-intensive applications that make use of scientific datasets in remote/archival storage systems across a wide-area network. DataCutter uses distributed processes to carry out a rich set of queries and application specific data transformations. DataCutter also provides support for sub-setting very large datasets through multi-dimensional range queries. It uses a multi-level hierarchical indexing scheme, based on R-tree indexing methods, to ensure scalability to very large datasets.

The programming model implemented in DataCutter is loosely based on the *stream-based programming model*. In the filter-stream programming model, part of an application is represented by a collection of *filters*. A filter is a portion of the full application that performs some discrete function. Filters can pre-disclose dynamic memory and scratch space needs so that the required space can be allocated by the underlying runtime system on behalf of the filter. Communication with other filters is solely through the use of *streams*. A stream is a communication abstraction that allows fixed sized un-typed data buffers to be transported from one filter to another.

A runtime system infrastructure, called the DataCutter Filtering Service, provides support for the execution of applications that are structured in the filter-stream programming model. The lifetime of a filter is defined by the amount of work required by the application. Within this lifetime, a filter can process multiple logically distinct portions of the total workload. This is referred to as a *unit-of-work*, and provides an

explicit time when adaptation decisions may be made while an application is running. A unit-of-work starts with the submission of a work description to a running set of filters, and ends when the last filter finishes processing the work. A collection of running filters that operate collectively to process a unit-of-work is referred to as a *filter instance*. An application may have multiple concurrent filter instances.

System	Main features
WebFlow	• visual programming, steering, run-time data analysis, and visualization • implemented as a Web three-tier system • based on Globus Toolkit
GridPort	• is the most well-known toolkit widely used for building Grid portals • allow developers to connect Web-based interfaces to the computational grid behind the scenes • Perl is used to register a new Grid application, while Back-end application programs must be built in Globus API
XCAT Science Portal	• based on the NCSA Grid Science Portal concept • an "active document" describes the science of an application along with parameterized, executable scripts
Cactus	• programming environment for large-scale simulations from numerical relativity to astrophysics • parallel I/O, checkpointing and recovery, remote monitoring and steering • support for MPICH-G2, PVM, pThreads, OpenMP, IEEE IO
Data Cutter	• provides support for developing data-intensive applications including sub-setting of very large datasets through multi-dimensional range queries • uses a multi-level hierarchical indexing scheme, based on R-tree indexing methods, to ensure scalability • implements the *stream-based programming model*
Knowledge Grid	• provides tool and services for Parallel and Distributed Knowledge Discovery • includes a metadata management for the description of resources characteristics, such as data sources, data mining software, results of computations, etc. • the RAEMS services tries to find a mapping between an execution plan and available resources on the grid, satisfying user, data and algorithms requirements

Table 3. Grid-based PSE Toolkits

Knowledge Grid

The Knowledge Grid [Cann03b] is a software infrastructure for Parallel and Distributed Knowledge Discovery (PDKD). The Knowledge Grid uses basic grid services such as communication, authentication, information, and resource management to build more specific PDKD tools and services. Knowledge Grid services are organized into two layers: *core K-grid layer*, which is built on top of generic grid services, and *high level K-grid layer*, which is implemented over the core layer. The core K-grid layer comprises two basic services: *Knowledge Directory Service* (*KDS*) and *Resources Allocation and Execution Management Service* (*RAEMS*). The KDS manages the metadata describing characteristics of relevant objects for PDKD applications, such as data sources, data mining software, results of computations, data and results manipulation tools, execution plans, etc. The goal of

RAEMS services is to find a mapping between an execution plan and available resources on the grid, satisfying user, data and algorithms requirements and constraints.

The high level K-grid layer comprises the services used to build and execute PDKD computations over the grid. The *Data Access Service* (*DAS*) is used for the search, selection, extraction, transformation and delivery of data to be mined. The *Tools and Algorithms Access Service* (*TAAS*) is responsible for the search, selection, downloading of data mining tools and algorithms. The *Execution Plan Management Service* (*EPMS*) is a semi-automatic tool that takes the data and programs selected by a user, and generates a set of different possible execution plans. Execution plans are stored in the KEPR to allow for the implementation of iterative knowledge discovery processes, e.g. periodical analysis of the same data sources. The *Results Presentation Service* (*RPS*) specifies how to generate, present and visualize the PDKD results (rules, associations, models, classification, etc.), and offers methods to store in different formats these results in the KBR.

2.2. Roadmap to Grid-based PSE Toolkits

Several aspects of Grids may simplify the process of developing and maintaining PSEs. Taking into account the main Grid properties and services, a PSE Toolkit architecture has to be investigated, by individuating the components required, the middleware that integrates them, the techniques that alleviate the user in building efficient and effective solutions.

In designing the architecture of a PSE Toolkit is crucial to determine what components of a PSE are common, and what are tied to a specific application domain. The common components can be designed and implemented in a PSE Toolkit and they can be used when a specific PSE must be developed. Design of Application-bound components and their interfaces must be considered in the Toolkit, but its implementation will change depending on the PSE in which they will be included. Most of the PSE toolkit components are generic. Specific components are:

- Components that define goals, resources, actors of the application domain;
- The ontology of the domain components;
- Domain knowledge;
- Performance history;
- The interface between the application domain and the PSE toolkit (see Section 4.8).

Most of the rest of a PSE infrastructure is generic and can be used across multiple application domains.

Some main issues to deal with in designing a Grid-based PSE Toolkit are as follows:

- Generic and specific components identification,
- Understanding and evaluating global properties,
- Component integration,
- Handling with distribution of resources and components,

- Component middleware, technology, and infrastructures;
- Adaptivity and heterogeneity,
- Standardization,
- Efficiency.

Grid-based PSEs may change the way high-performance computing resources are used to solve problems. Grids will offer a wide-area environment where to search for PSE components and where to run PSE applications different of any other infrastructure we used before. Together with computing-intensive applications, the Grid will allow users to run data-intensive applications that make use of very large data repositories and access remote instruments and sensors. Data-intensive applications are growing in importance, and introduce interesting new dimensions such as irregularity, data representation and storage, distributed data access, and heterogeneity, that have not been so critical in scientific and numerical computing.

Grid-based PSEs may succeed in meeting more complex applications requirements both in terms of performance and solution complexity. They integrate heterogeneous components into an environment providing transparent access to distributed resources, collaborative modeling and simulation, Grid portals.

3. A Reference Architecture for a PSE Toolkit

As mentioned in the previous section, PSE portability, extensibility, and flexibility can be provided through the use of a PSE Toolkit. The design of a software infrastructure easing the building and deployment of powerful PSEs is the main goal of this report. To achieve this goal, here we identify the main components of a PSE Toolkit and how these components should interact for implementing PSEs in a distributed setting.

A minimal reference architecture of a PSE Toolkit should include:

- A graphical user interface;
- A repository of usable components, the repository can include a set of user-defined applications;
- A metadata-based description/information system, possibly based on application domain- and software components ontologies;
- A metadata repository, possibly including an ontology repository;
- A search/discovery system;
- An execution/resource manager.

Figure 1 shows all these components and the interactions among them. In particular, the Component Repository represents a library of objects used to build PSE applications, the Metadata Repository is a knowledge base describing such library, whereas the remaining components are services used to search, design, and execute PSE applications.

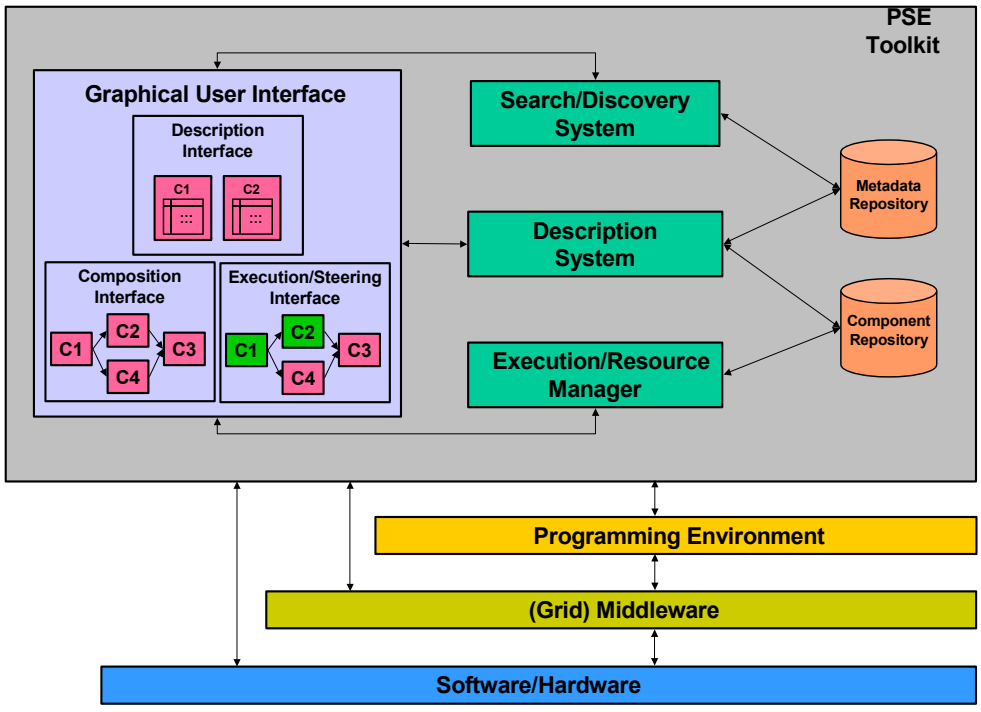

Figure 1. A reference architecture of a PSE toolkit

Graphical User Interface. The Graphical User Interface allows for:

- The description of available components, the semi-automatic construction of the associated metadata and their publishing. For doing this, the GUI interacts with the Description System. These actions can be driven by the underlying ontology that helps user in classifying components.
- The construction of applications and their debugging. For doing this, it goes through the following steps:
    1. Interaction with the Search and Discovery System for letting users pose queries and collect results about available components potentially usable for the PSE composition, or directly browse the metadata repository. As before, both querying and browsing can be guided by ontology.
    2. Design of an application, through the visual facilities provided.
    3. Interactive validation and debugging of the designed application.
    4. Invocation of the Execution Manager, passing it the global executable code.
- The execution, dynamic steering of applications and result visualization. Here, the GUI interacts with the Execution Manager for monitoring the PSE execution and showing its results, and for passing it the user commands for the PSE application steering.
- The storing of useful applications and results, annotated with domain knowledge. In this way, applications and their associated domain knowledge can be shared among PSE users.

In order to seamlessly integrate facilities of external tools, the Graphical User Interface must also allow for the direct use of their native GUIs, in all the phases of interaction with users seen above. Indeed, it may be needed to perform particular interactions with external tools, provided through their native GUIs only, or whose integration in the Toolkit's GUI would result in unnecessary complexity.

Component Repository. The repository of usable components holds the basic elements/modules that are used to build a PSE. Examples of such components are software tools, data files, archives, remote data sources, libraries, etc. The repository must be able to integrate a possibly pre-existing one provided by the programming environment used to implement the PSE Toolkit, also comprising components defined by different languages and/or tools. The structure of this repository depends on the class of architecture taken into account. It can be designed as a traditional database in case sequential machines are considered, whereas it must be designed as a parallel/distributed database if a parallel architecture is used or the PSE Toolkit is to be implemented on a computer network or on a Grid. In the Grid case, the Component Repository must be able to manage dynamic connection/disconnection of Grid resources offering components. In fact, in this case computing nodes and network connections must be considered as components that can be used in the implementation of a PSE. The component repository can include complete user-defined applications that, when published in the repository, enhance the PSE capability to solve specific problems.

Metadata Repository. The Metadata Repository stores information about components in the Component Repository. Such information comprises components' owners and providers, access modalities, interfaces, performances (offered or

required), usage history, availability and cost. For the description of component interfaces, the PSE Toolkit provides an IDL every component must comply with in order to be usable by the Toolkit. Thus, a component's provider either makes it accessible directly using the IDL (through wrappers), or describes in the associated metadata how to translate the invocations written using the Toolkit IDL into actions to be taken and/or into invocations written in the component's native language. Finally, also reusable *solutions*, i.e. modules of applications for solving particular sub-problems, that are typical for PSEs, may be managed and described as resources. The Metadata Repository can be implemented by using a Domain Ontology, i.e. a conceptualization, in a standard format, of component's metadata, component's utilization, and components relationships. Such ontology can be extended to describe the user-defined applications.

Description System. The Description System must be able to offer a clear description of each element a PSE can be composed of. In the proposed architecture, it is based on the Metadata Repository. An important issue to be considered for the Description System is the use of ontologies for enriching the semantics of components' descriptions. This layer extends the metadata facilities and may provide the user with a semantic-oriented view of resources. Basic services of the Description System are *component classification* through taxonomies, *components annotations*, for example indicating which problem they are useful for, and *structured metadata description*, for example by using standard, searchable data. Whenever new components/applications are added to the Component Repository, new knowledge is added to the Metadata Repository through the Description System.

Search/Discovery System. The Search and Discovery System accesses the Metadata Repository to search and find all the resources in the environment where the Toolkit is run, which are available to a user for composing a PSE. Similarly to the Component Repository, while the implementation of this system on a sequential machine is straightforward, in a parallel/distributed setting the Search/Discovery System should be designed as a distributed service able to search resources on a fixed set of machines. Moreover, a dynamically changing heterogeneous collection of computers must be searched if a Grid computing environment is considered. Basic services of the Search and Discovery System are: *ontology-based search* of components, that allows for searching components on the basis of belonging taxonomies as well as specifying constraints on their functions, and the key-based search. The former operates on a portion of the knowledge base, as selected through the ontology, whereas the latter operates on the entire knowledge base.

Execution/Resource Manager. The PSE Toolkit Execution/Resource Manager is the run-time support of generated PSEs. Its complexity depends on the components involved in the PSE composition and in the hardware/software architecture used. In particular, if a multiprocessor or a Grid is used, the Execution/Resource Manager is composed of several processes/threads that concurrently execute the PSE code and coordinate their activities for exploiting hardware resources. In these cases, the Execution/Resource Manager must tackle several issues, e.g. the selection of the actual component instances to be used at each PSE execution, the suitable assignment of processes to heterogeneous computing resources (scheduling), and the coordination of their execution, possibly adapting the running applications to run-time (unpredictable) changes in the underlying hardware/software infrastructure. If a Grid computing architecture is considered, the Execution/Resource Manager, besides interaction with the software/hardware system, has a tight connection with the Grid fabric environment and with Grid middleware. Note that the Execution/Resource Manager is also responsible for performing all the actions

needed to activate the available components, driven by their metadata. These actions may comprise, e.g., compiling a piece of code or a library before using them, or launching a program onto a particular virtual machine. The execution manager must have tight relationships with the run-time support of the programming languages used for coding the components.

In summary, the main characteristics of the proposed PSE Toolkit architecture are: (*i*) the existence of a knowledge base built around pre-existing components (software and data sources); (*ii*) the PSE application composition conducted through the searching and browsing of the knowledge base; and (*iii*) a distributed and coordinated application execution on the selected distributed platform, that can be a traditional distributed system or a Grid. Specific information about the technologies and techniques used to describe the PSE Toolkit architecture can be found in the remainder of this section.

3.1. Components and Programming Models/Environments

A variety of programming methodologies and strategies have been proposed and developed to address applications development over the grid. Often they were derived from models available in parallel and distributed computing systems, with minor changes to make them adhere to the dynamism of grid environments. A detailed classification of such techniques has been also discussed by the GGF Programming Models working group [Lee01].

In the following we briefly describe some models we analyzed to choose the most suitable for a grid-based PSE Toolkit.

Shared space models provide a global shared memory even if the real implementation may be distributed. Two interesting implementations extended to grid environments, comprise JavaSpaces [Free99] and OpenMP [Ope97].

Message passing models on the contrary do not share a common address space. The processes send messages using two-sided or one-sided communications in order to exchange data. Currently there is an implementation of MPI (Message Passing Interface standard) for grid environment that uses the Globus services (MPICH-G2) [Fost98].

Peer-to-peer computing and in particular the JXTA [jxta03] implementation enables the programmer to cope with a model in which all the processors are considered at the same level and there is not a hierarchical organization.

Object oriented technologies, coupled with remote procedure calls, provide an useful model to operate on the grid. Typical examples are Corba and JavaRMI.

In the *component model*, applications are assembled, even at run-time, from components selected from a component pool. Examples of this model include Corba Component Model [Corb03], a model based on distributed software components, evolution of Corba technology; JavaBeans Component Architecture [Jb03], developed by Sun and based on Java; Common Component Architecture (CCA) [Arms99], a project raised by the Department of Energy (DOE) to apply component model to high performance computing, XCAT [Xcat02] is an implementation of this model with some extensions adopting web services as a basic architecture for grid environments.

Evaluating these models and implementations in the optic of requirements and problems of a PSE framework in grid environments, the *component model* and in particular many ideas derived from CCA seem to be the most appropriate solution. There are many aspects that played a role in this evaluation, like code reusability, modularity and encapsulation, portability, performance adaptability, interoperability, need of hiding differences and providing a unified abstraction able to manage a heterogeneous environment.

The Common Component Architecture

The main idea in building applications using the *component paradigm* is the assemblage of components. The way two CCA components are assembled is by connecting together their "ports". A component may own two types of ports: *provides ports* and *uses ports*. Provides ports represent functionalities a component provides to other components; uses ports represent functionalities a component may need. When a uses port is connected to a provide port, it gain access to the functionality exported by the related component. Ports are identified by names and can be connected each others only if they are of the same type.

Unlike some other component models, in the CCA model ports can be added, removed, and connected at run-time. This makes CCA a more powerful and flexible model, able to be employed in problem-solving environments and to fulfill the more general requirement of software adaptability. Figure 2 shows a sample connection between a uses and a provides port.

For effectively using a component architecture it is important to describe the various components that constitute an application along with their interconnections. Recently, two different categories of composition have been proposed: composition in time and composition in space [Govi03].

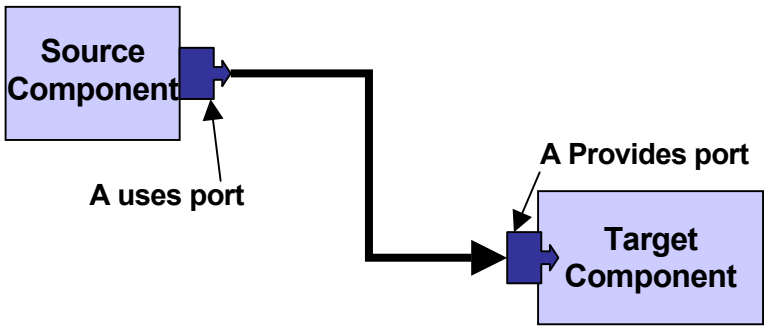

Figure 2. Uses and provides ports

Composition in space. Component instances are created on specified hosts and then connected together. It is also possible to create *meta-components* which are themselves created by composing a number of components, which constitute a new component.

Composition in time. While the composition of components in space defines how the components are logically connected, workflow systems define ways in which flow of control and data can be expressed taking into account the time dimension. These two compositions are orthogonal, thus they can coexist.

To accomplish the first kind of composition, some APIs can be used directly by the user to write simple programs that can use remote component instances. For example the Java programming language and the Jython scripting language are used in XCAT. Usually a program can invoke the creation service to create components and obtain references to running instances. Afterwards, through the connection service it is possible to connect the provides and uses ports of these components.

About the latter kind of composition formalisms derived from workflow technology can be employed to describe the time ordering of components execution along with other dependencies needed to fulfil a given business process (see Section 3.2).

Component interoperability

Components can be implemented in different programming languages, can export data in different formats (including components using legacy applications), can have or not some characteristics (a steering interface, a performance model, etc..), and can present different metadata descriptions. So, interoperability under various aspects represents a big problem in designing a component architecture and should be analyzed considering the different basic layers. Transport may be supplied by HTTP protocol for standard application and by TCP for high performance components, i.e. components that need to exchange data in a timely fashion. SOAP could be the standard method for messaging among components along with other techniques useful in particular cases (for example message passing and file transfer). Metadata description could use an extension of WSDL permitting description of aspects derived from the high performance nature of some components (like parallel and container components, see below, and/or performance models). A more complex issue concerns the discovery service: its implementation may be supported by a peer-to-peer system or by a hybrid system using also hierarchical parts (see Section 3.4 for more details). An ad hoc publish-subscribe system could be designed for the PSE toolkit so as to provide a reliable mechanism for events notification. It must supply reliability even if we have changes in the system (a subscriber changes its location or a publisher is restarted) and could be based on XML and SOAP to guarantee interoperability or could consider peer-to-peer systems and JXTA protocol as an alternative implementation.

The component integration can be performed by using message passing, remote method invocation, web services etc. In any case, the use of XML is recommended to define metadata about components, interfaces, and messages.

In a PSE toolkit some high level components are needed to accomplish the building of complex applications. At least three kind of basic components should be supplied by the system:

- *Data management components* handle data, they comprise components for accessing data, creating replica, moving and partitioning data;
- *Computing components* elaborate information, they may be sequential, parallel and distributed or container (graph composition of homogeneous components to be executed using a particular form of parallelism). Naturally, parallel and distributed components and container components should have metadata describing the particular execution environment;
- *Components* provided from the PSE framework for a *specific task* (for instance visualization components, steering components, etc..) or for a *specific application domain* (see Section 4.8).

High level visual environments

In a grid based PSE toolkit it is important to include high level facilities and visual instruments able to support end users with different backgrounds and with little or none expertise about distributed systems. This kind of tools can offer a set of abstractions and functionalities useful to hide grid related issues, letting the user concentrate on the problem resolution. Moreover, they can represent an appropriate and effective way to propose the component programming model to users and developers.

Some systems offering such characteristics are listed and briefly analyzed below. They are surely a reference point to better understand the role they may play in the overall system we are proposing here and how it could be possible to exploit these enhanced instruments as supporting environments for the component programming model.

GECCO (*Graph Enabled Console COmponent*) is a graphical tool developed at Argonne National Laboratory [Lasz00, Lasz00a]. GECCO is based on the Globus CoG Kit [COG03] and provides facilities to specify and monitor the execution of sets of tasks with dependencies between them. In particular it allows to specify the jobs dependencies graphically, or with the help of an XML-based configuration file, and to execute the resulting application. Each job is represented as a node in a graph. A job is executed as soon as its predecessors have successfully completed. It is possible to set up the specification of the job while clicking on the node: a specification window pops up allowing the user to edit the Resource Specification Language, the label, and other parameters. Editing can also be performed at runtime (job execution), hence providing for simple computational steering.

VEGA (*Visual Environment for Grid Application*) was designed to support the planning and execution of data-intensive applications upon grid environments and is a component of the Knowledge Grid [Cann03]. VEGA provides a set of high-level functionalities ranging from *design facilities* to *consistency checking*, *execution management*, *credentials management*, and *projects management*. All these features have been originally developed to support the design of data analysis and knowledge discovery applications, but they are suitable to satisfy the requirements of most general purpose applications. Key concepts in the VEGA approach to the design of a grid application are the *visual language* used to describe in a component-like manner, and through a graphical representation, the jobs constituting an application, and the possibility to group these jobs in *workspaces* to form specific interdependent stages. A consistency checking module parses the model of the computation both while the design is in progress and prior to execute it, monitoring and driving user's actions so as to obtain a correct and consistent graphical representation.

WebFlow [Akar98]. Although from many classified among portal systems WebFlow is an interesting system offering a set of useful authoring facilities. WebFlow has a three-tier Java-based architecture that could be considered a visual dataflow system. The front-end uses applets for authoring, visualization, and control of the environment. Application integrators use visual tools to link outputs of the source modules with inputs of the destination modules, thereby forming distributed computational graphs (or compute-webs) and publishing them as composite WebFlow modules. A user activate these compute Webs by clicking suitable hyperlinks, or customizing the computation either in terms of available parameters or by employing some high-level commodity tools for visual graph authoring. The backend of WebFlow is implemented using the Globus Toolkit. WebFlow is based on a mesh of Java-enhanced web servers (Apache), running servlets that manage and

coordinate distributed computation. Each servlet can communicate with others via sockets; the servlets are persistent and application independent. WebFlow has been employed into the Gateway Computational Web portal and the Mississippi Computational Web Portal.

3.2. Workflow Management

A workflow (WF) is defined as a set of *activities involving the coordinated execution of multiple tasks performed by different processing entities* [Rusi94].

Workflow technology dates back to late 70's, when it has been adopted in office automation and batch processing areas to get the work "well" distributed among several persons/systems in accordance with a specified procedure. In recent years, workflow technology received much attention due to its aptitude for modeling the execution and management of business processes. Also the development of information systems towards infrastructures based on multiple stand-alone computers linked by networks, poses a problem to allow systems, not necessarily designed to work together, to be involved in a common work coherently and in an efficient way. Workflow systems are being effectively employed in this field, often as a result of a business process reengineering.

Problem solving applications can be effectively described as workflows that define relations and dependencies among application tasks and describe the logic that will drive the application execution. While the workflow defines the "schema" of the application, a single application to be executed is a "workflow instance"; the term "workflow" can refer to both meanings, depending on the context.

Over time WF gained a broad application in several domains; this brought to the need of categorizing WF systems. A brief analysis of WF classification is helpful to understand differences between different systems and to show how WF can adapt to various contexts.

Main categories are:

- *collaborative workflow*, used in professional and administrative areas, is characterized by negotiation about who will do the work, and provides an efficient control of the process;
- *production workflow* supports high production volumes in the execution of fixed or slightly modifiable procedures; provides control of the process and improves productivity;
- *horizontal and vertical workflow*, it represents a further segmentation about the structure of WF. A horizontal WF moves the work through the organization, from person to person or among different departments or systems (it is often called "routing"); once the work gets to an organization area, a vertical WF drives the execution of a number of steps within that area.

Workflow Management System

The execution of the multiple tasks included in a workflow specification, performed by different processing entities, may be controlled by a human coordinator or by a software system called W*orkFlow Management System* (WFMS). A workflow management system manages the execution of workflow processes by means of a

scheduler and *task agents*. The scheduler provides the ability to submit tasks for execution, track their progress, and enforce policies and consistency constraints (inter-task dependencies). A task agent is associated to each task to apply actions decided by the scheduler and report state changes.

In a Grid environment, a workflow instance gains access to Grid hardware and software resources; Grid specific issues (e.g. resource individuation, security and authorization requirements, etc.), have to be tackled prior to executing a workflow instance. As a consequence, a WFMS operating in a grid environment must integrate and/or implement such functionalities. The process of integrating workflow policies and functions with grid resources may bring to a full combination of both technologies. For example, the emerging Open Grid Services Architecture (OGSA), in which the basic entity is the *grid service* (see Section 4.2), allows to define and publish new grid services as a combination (in a workflow like manner) of simpler ones. This concept can be better understood if we think at grid services as a particular class of *components*, like those defined by the Common Component Architecture.

A key aspect in defining a workflow is the specification of each task, comprising the description of the characteristics the processing entity must provide in order to accomplish the task execution.

A task in a workflow is a unit of work that can be processed by a processing entity. The specification of a task is a description of its structure. A task structure can be defined by providing: a set of externally visible execution states, a set of transition between these states and the conditions that enable state transitions.

A WF can be naturally defined by a Directed Acyclic Graph (DAG) of tasks, with a single root and one or more terminal nodes. A number of formalisms and systems, aimed at providing a more or less powerful and flexible way to specify and describe a WF, exists. Indeed, the Workflow Management Coalition [WfMC03], the body in charge of the development of international standards for WF interoperability, never proposed a precise specification model.

DAGMan. The Directed Acyclic Graph Manager (DAGMan) is a meta-scheduler for the Condor opportunistic grid environment [DAGM03]. DAGMan allows to specify a set of task dependencies and to submit multiple jobs to Condor in the proper order. The tasks (basically software programs) are nodes in the graph, and the edges identify the dependencies. Task dependencies in DAGMan consent to specify a priority relation regarding input, output, and execution of programs. A configuration file, defined prior to submission, describes the DAG; in addition a Condor submit-description file for each program in the DAG is used by Condor during the execution.

GridAnt. GridAnt is a framework for specifying and orchestrating grid tasks with complex task dependencies. The implementation of GridAnt is based on the commodity tool Apache Ant [Ant03], hence, it provides the ability to use features from Ant such as the XML specifications of tasks, the control flow specification through conditional, sequential, and parallel constructs, and the availability of a workflow processing engine.

WSFL. The Web Services Flow Language is an XML language for the description of web services compositions as part of a business process definition [WSFL03]. WSFL considers two types of web services compositions: the first type specifies an executable business process known as “flow model”, the second type specifies a

business collaboration unit known as “global model”. WSFL provides extensive support for the recursive composition of services: in WSFL, each composition of web services can itself become a new web service, and can thus be used as a component exploitable in new compositions.

Execution plan. It is a formalism adopted by the Knowledge Grid, a software infrastructure for distributed and parallel knowledge discovery on the grid [Cann03]. It is constituted by an XML document describing the tasks of a knowledge discovery application. The specification comprises a list of tasks and task links, which are specified using the XML tags *Task* and *TaskLink*, respectively. Task elements specify each basic task at a high level, without physical information about resources, which can be identified through metadata references. TaskLinks are used for linking various tasks to form the overall task flow.

Workflow Scheduling

Inter-task dependencies define task coordination requirements, specifying how the execution of a task is related to that of others or to external variables. Inter-task dependencies may be specified as constraints on the occurrence or the temporal ordering of significant events generated by the involved tasks. The terms *execution dependencies*, *data dependencies*, and *temporal dependencies* are used in the literature to refer to various kind of dependencies. In a WFMS that manages dynamic workflows, the specification of dependencies, as well as tasks, can be added at run-time.

The workflow scheduler is the component responsible for assigning single tasks to different processors and computers, enforcing task-dependencies, and coordinating the execution of tasks in the workflow. In [Taha99] a distinction is made among three scheduling approaches: centralized (a single scheduler schedules the tasks of all concurrent workflows), partially distributed (one scheduler for each workflow), and fully distributed (no scheduler is used, but task agents coordinate their execution by communicating with each other).

In a Grid environment, it is natural to map a workflow task to a “Grid job” that can be assigned and executed on a specific Grid host. Furthermore, a workflow processing entity can be any Grid resource able to perform a workflow task (computers, storage systems, sensors etc.). In such a context, a workflow is managed by a Grid *meta-scheduler*, which assigns and schedules jobs to different machines, while single tasks are managed by job schedulers (e.g. PBS, LSF etc.) located at specific Grid hosts. Unfortunately any scheduling and timing decision taken by the meta-scheduler might possibly be made ineffective by local scheduling strategies of Grid hosts. The issue of interrelating the meta-scheduler and the different host schedulers is important and not well investigated till now.

Grid workflow can be specified at different abstraction levels: in [Deel03], differences are highlighted between *abstract* workflows, which specify resources using logical files and logical component names, and *concrete* workflows, where the user needs to specify the exact executables and resources to be used. In [Mast03] a similar distinction is made, for data mining applications, between *abstract* and *instantiated* execution plans.

In our opinion, the definition of workflow in the context of generic PSE toolkits suggests the addition of a further abstraction level, to help PSE experts defining their domain specific applications. Three abstraction levels are therefore individuated:

1. **Domain specific workflow**. A domain scientist or engineer should be able to define its application using his/her own domain language, without being compelled to know the details of the Grid environment: at this level a workflow is built upon domain specific components and services well known by the domain expert.
2. **Abstract workflow**. An abstract workflow looses any explicit reference to the PSE domain: a workflow is built upon hardware and software resources available on the Grid. This implies that a mapping has to be made between domain specific tasks and Grid resources. However, at this level, resources are not determined, but are defined by a set of constraints and requirements to be satisfied. For example, a host can be denoted by requirements on processor speed, amount of RAM memory or disk space, etc; a Grid software can be specified by requirements on the input data it processes or on the type of platform on which it can be executed.
3. **Concrete workflow**. When the workflow is going to be executed, all resource constraints have to be evaluated and resolved on a set of available Grid resources, in order to choose the more appropriate resources for the current status of the Grid environment. Of course, due to the dynamic nature of the Grid, an abstract workflow can be "instantiated" into different concrete workflows at different times. A concrete workflow should be defined in XML, to assure interoperability and human-readability: however, before being executed, it should be translated in a specific job definition language (e.g. DAGMan or Globus RSL). Depending on the features of the specific grid language, a concrete workflow can be directly expressed in such a language, or minor modifications have to be made: for example, if the job specification language does not provide powerful workflow functionalities, a complex concrete workflow should be translated in one or more pieces of that language, whose execution needs to be coordinated by the WFMS operating on the Grid.

A workflow can be designed at each of the abstraction levels described above, depending on the user knowledge of the Grid environment, on the type of workflow, and on the services available on the PSE toolkit. For example, if the workflow designer wishes to execute a workflow on predetermined Grid resources, he/she can directly build a concrete workflow. On the other hand, if the designer has little knowledge of the resources available on the Grid, and/or desires to concentrate on the workflow logic, he/she could better design a domain specific workflow, and rely on PSE tools to obtain the corresponding abstract and concrete workflows.

An important issue is how the PSE toolkit can translate a domain specific workflow to an abstract workflow, and then to a concrete workflow. An efficient way to perform the first kind of translation can exploit the features of ontology systems (see Section 4.3): a domain specific ontology is aware of the semantics of domain components and services, and therefore should be able to map these components and services onto Grid resources.

The second step, i.e. the instantiation of an abstract workflow into a concrete workflow, is performed by the Grid meta-scheduler that takes charge of finding the Grid resources that better satisfy constraints specified in the abstract workflow. The

meta-scheduler may performs a *static* scheduling if the instantiation can only be made before workflow execution, or a *dynamic* scheduling if it is possible to modify the concrete workflow during execution, i.e. at run time. For more details see Section 3.5.

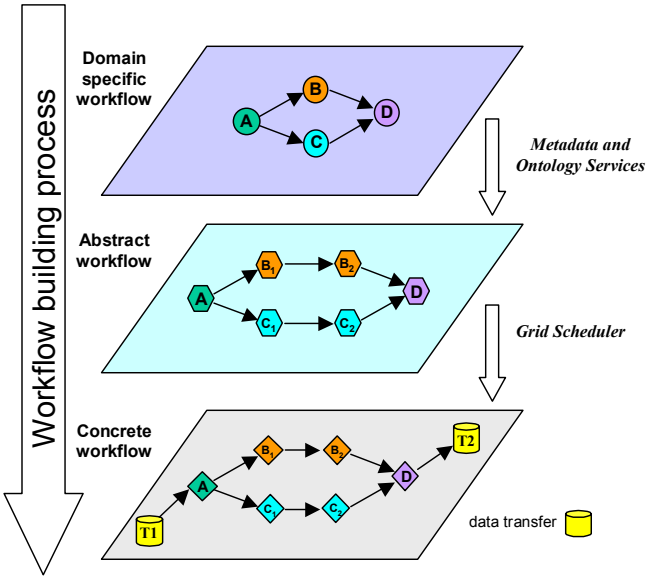

Figure 3. Workflow abstraction levels

Figure 3 shows an example of workflow building process that permits to obtain a concrete workflow starting from a domain specific one. Notice that, along this building process, a single domain specific task may be instantiated into one or several abstract tasks; furthermore, at the concrete workflow level, data transfer jobs may be added by the grid scheduler as a consequence of its mapping process.

3.3. Data Management

Scientific problem solving environments seek to integrate the activities necessary to accomplish high level domain tasks. They may include support for managing scientific workflow, tracking, transforming and filtering data, automating feature extraction, and annotated records management. Pulling these and other capabilities together to provide systems that support scientific problem solving requires a flexible and dynamic data management architecture. The data architecture serves as the underlying glue that ties together long running research processes and widely distributed collaborators. Moreover, it is necessary to provide an architecture able in storing, indexing, and provide access to flat data, structured and semi-structured data, remote resource identifiers with their description, as well as software components.

In the late 1980's and throughout the 1990's, Relational Database Management Systems (RDBMS) and Object Oriented Database Management Systems (OODBMS) technologies emerged as the leaders for traditional data management solutions. These architectures are well suited for organizations that are centrally located and which have control over the client platforms and systems. The data is located in one centralized server or several replicated servers and managed by one organisation. In

both models, data respect a uniform structure defined in metadata (structured data) and can be accessed by formulating queries using data structures. Data in PSEs can be organized in structured format, but to provide access to heterogeneous data and information, structuring data is not a sufficient solution. Indeed, data in PSEs are shared by many organizations with no central control. Data can be stored in multiple databases, files systems or both. As we will see later, traditional architecture requires agreement and enforcement of ontologies and schemas that are then mapped into the underlying technology.

Even using metadata and ontologies, requires a upper level user to be able in managing data and PSE structure changing. I.e., (i) as the scope of a PSE increases, the number of parties that must agree upon the ontology become large; (ii) as components are incorporated in a PSE, negotiation is required between the components developers and the PSE framework designers and data administrators; (iii) as best practices evolve or PSEs are extended to support users with different goals, the schema and data structures must be changed simultaneously and existing data migrated.

We report on the data management topic in PSE mainly from four levels:

- Data Sources and Data components repository
- Data Description: Ontologies and Metadata
- Data and Component Access Services
- Information Services

We claim that each node in a Grid may have a data management architecture as the one depicted in Figure 4.

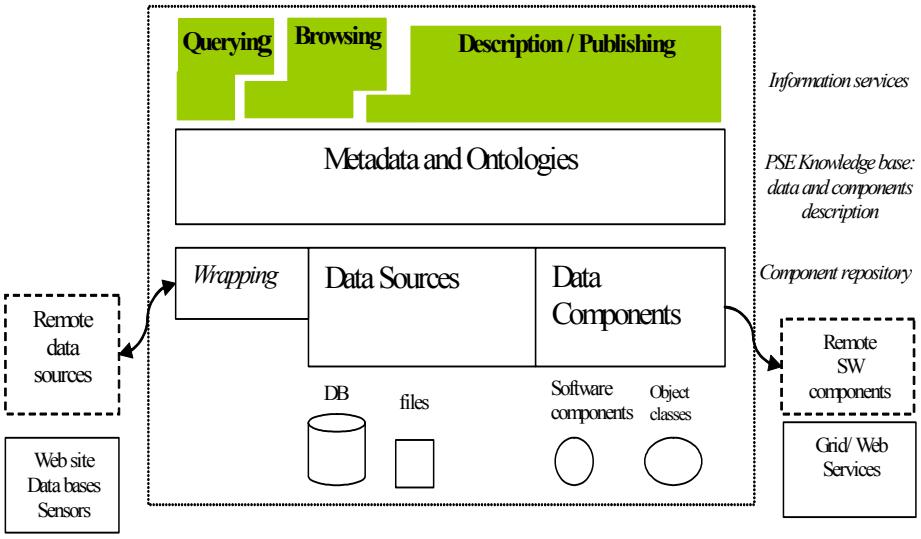

Figure 4. Data Management layers in a PSE Grid Node

Data Sources and Software Components Repository

Each node in the GRID may contain information that can be used while composing applications. Data can be static information (structured and semi-structured

databases, data files, or URL links to remote data sources) as well as software components (classes, linkable objects, byte codes, or remote software components, such as web and grid services).

Data Sources and components are depicted in lower layer of Figure 4.

To build an application in a PSE it is necessary to know existence of data, their localisation, and how to access or use them. It is then necessary a data description and data access policy allowing using data and information. Data can be accessed through their metadata descriptor or allowing a direct access to data. For instance in case of structured data (as in the relational case), application may use a common access policy guarantying data consistency. If data are unstructured, or information are originated from heterogeneous databases, we may provide a uniform view on the data that can be used by the application as they were part of a single uniform database.

Data and components are usually pre-existing to the PSE, so it is not convenient to copy them in a real repository: so the data and component repository can be viewed as the set of physical location hosting them (e.g. file systems, database partitions, external web sites, etc.). Their description can be accomplished by using different approaches:

(i) Data and components may be described by using abstract XML views to describe data and software components properties. The view may be a tree structure representing data properties and software functionalities. The XML description may keep trace of data format, structure, validity. For each software component adopting such XML view may describe how to use software as well as component properties.

(ii) Adding a virtual view to data may limit performance of the application. Another possibility may be to materialize data or components where necessary before running application. Data materialization consists in copying data and information adapting them to structure imposed by the application. The problem of integrating data in a uniform format has also to be faced in case of data wrapping from remote resources. (see low part of Figure 4). Our data management module needs to be enriched by a data resource location and a data mapping mechanism able in converting data from different sources to common structured or unstructured data.

(iii) Data structure representation and description may be also partially solved using self-describing data format (e.g. as in semi-structured data bases) where data arrives with their metadata. In the latter case, applications can extract information from data, mapping data sources to a common format. Finally for self-describing data, it is possible to support data evolution or schema evolution, without changing entirely data store. For PSE toolkit it is necessary to study the problem of data evolution both in structure and in contents. For this reason it could be better to materialize heterogeneous data on the fly when application requires fresh data. Nevertheless, data materialization is a well known bottleneck problem by database community.

As shown in upper part of Figure 4, querying browsing and data description services can be applied to data sources to furnish access to data. Moreover, for data sources we want to provide a data restructuring to return data in a format or schema required by applications. Data description is required also by application looking for software components.

As shown in Figure 4, we consider both software classes that need to be compiled or linked, as well as binary code that can be run on given platforms.

Components can be stored locally or can be located on remote nodes. A PSE node must be able in retrieving and composing software modules to build application. In case of remote modules, web protocols (as SOAP or UDDI) may be used to specify format of data input and output. Applications can load modules locally from the data management services (see upper part of Figure 4) or may ask a node to retrieve data applications.

As in for raw data components, even for software components and modules, it is necessary to provide formalism to describe information associated to each module. In particular for each component we need to provide:

- Functionalities
- Input and output data format
- Function interfaces (i.e. parameter passing and message information)
- Module type and platforms (i.e. binary code, byte code, compiling softwares)

All such functionalities need to be described by metadata and ontology modules.

Data Description: Metadata and Ontologies

A PSE toolkit uses resources heterogeneous both in data and software specifications (application). To furnish a "solving environment", a PSE need to be enriched by data and software abstract description to permit users to build their applications using a *knowledge base (KB)* that explains how elements of the component repository are organized.

The problem of managing heterogeneous data and components, is well known to database community as data integration problem [Lenz02]. Accessing heterogeneous resources can be partially solved using metadata, i.e., logical level data and application description, that, through an accurate categorization of resources, provide useful information about the features of resources and their effective use. A metadata level (see Figure 4) may extend the Globus MDS-2 model based on the LDAP protocol. Portable and simply accessible abstract description can defined using semi-structured formalism. E.g., XML metadata representation is used in [Mast03] to represent classes of Data Mining tools, (e.g. data mining software, data sources, execution plans etc.), whereas abstract XML view are defined in [Agui02] to allow querying large scale heterogeneous XML data. Web accepted language for data and service exchanging, i.e., XML, allows the usage of standard and widely adopted query protocols such as XQuery, and standard transformation protocols such as XSLT.

Metadata layer is also in charge of describing *locally* both data sources and software components. Local data sources need metadata about the access method and the access policies. On the other hand, remote data sources need further metadata to describe if they can be copied on another node, or materialized on the local node, or must only be accessed on the remote node. Similarly remote software components can be moved (byte code) locally, or can be copied and installed locally or have to be invoked on the remote owner node (e.g., through Remote Procedure Call or Web/Grid service protocols). If object classes and source code are also considered,

further metadata about how to compile and/or link them on the requesting node should be provided.

Ontologies can be used to build different views over the knowledge base. They can be used to model, at least, the two main classes of resources:

- software components and
- data components

Ontologies for software components should explain:

- which problem/task the component is suitable for;
- how a component can be invoked or linked (for example for an object file)
- what are the main requirements to execute a component.
- how a data source can be accessed; if it can be copied, or if it is stored remotely, but accessed through a local wrapper, etc.

Ontologies for data sources, should explain how the data is accessed, access policies, protection, format, and so on. The role of ontologies in PSE is explained later in this document.

Data and Component Access Services

Major issues in managing the access to data and components in a Grid-based environment are:

- **Virtualization** of data and components, i.e. allowing applications (e.g. developed using the PSE) to discover, access and update data in a manner independent by the data/component format and data/component location (format and location transparency).Different techniques play a role in virtualization, e.g. caching, replication, schema and format mappings (for data).
- **Management** of the way data and components are provided to applications with respect to performance levels and agreed global quality of service.
- **Integration** of data within the Grid infrastructure, i.e. providing with such data the same basic mechanisms of the Grid. For example, it should be possible to provide for data the necessary tracing, monitoring, and accounting information.

The Global Grid Forum Data Area working groups [GGF] are developing a suite of standards or adopting preexisting ones from different contexts, for the different aspects of data virtualization, management and integration in the Grid, with focus on data (databases, files, semistructured data sources, data streams, and so on). Although many of those works can be well adapted to the management of software components (i.e. the data file considered contain software), few efforts are devoted to specific aspects of software component management.

With respect to data, the following issues are studied and standardized [Mala03].

The **DAIS-WG** (Data Access and Integration Services Working Group) is working around a service-based interface for accessing and integrating on the Grid data available in pre-existing relational and XML databases. The planned services are to be implemented

outside of the database system and the access to such data sources happens using some constructs of the proposed model:

- A **Data Resource Manager** service represents a DBMS or XML management system. A user wishing to access to data contained in it has to create an instance of such service.
- A **Data Resource** service represents a database (i.e. a set of tables in a relational database or a collection of XML documents). A user wishing to access to a table/document in a database has to create an instance of such service.
- A **Data Access Session** service is the relationship between a client and a data source. Also in this case a user wishing using with a database has to create an instance of such service.
- A **Data Request** contains the SQL, XPath or XQuery request to be executed on a data resource. This construct specifies a data format that user has to use to send request to the data service.
- A **Data Set**: represents the output result returned in response to a Data Request. Also such construct specifies a format of data, i.e. the results.

The **OREP-WG (**OGSA Replication Services Working Group) is exploring data replication technologies for the grid. Initially focused on large files, it is also working on replicating generic files and databases. The first **OREP** specification regards a Replica Location Service (RLS). The RLS maintains a map of where data and replicas are stored and the availability of more copies of a given data can allow both to implement quality of service, and different level of access performance. If copies are updated, the problem of propagating updates between them has to be faced.

An important issue of data replication is data movement: how data is moved between Grid nodes to implement replicas. The **GridFTP-WG** working group works on a reliable file transfer protocol built on the top of the usual FTP (File Transfer Protocol).Equally important is the work undergone by the **GFS-WG** (Grid File Systems Working Group) that is working on the concept of a Grid file system. It is obvious that if work conducted by these last groups will drive implementation of native mechanisms for efficient data movement, effective storing of files, and transparency with respect to data location, the previous services could be more easily implemented and made available.

Finally, the **DFDL-WG** (Data Format and Description Language Working Group) is working on the management of data streams, i.e. how describe and annotate a set of data that constitute a stream. Annotating or labeling data allows adding information to original data to better explain the used encoding, the meaning of data, etc. The DFDL (Data Format and Description Language) is based on XML. Although **DFDL** is well suited for labeling and describing scientific data it could also be used to describe relational data sources (e.g. the layout and content of a database) or a query result set (e.g. described as a labeled data stream).

In summary, in the three layer Data Management presented so far, component repository contains the useful data and software eventually physically distributed over the Grid. Component ontologies and metadata form a PSE knowledge base whose main services are intelligent searching and browsing for data and software.

OGSA-DAI

Open Grid Services Architecture Data Access and Integration (OGSA-DAI) [OGSA-DAI] is a project aimed at providing a component library for accessing and manipulating data in a Grid environment. It is a partnership between NeSC, EPCC, IBM UK, IBM US, Oracle and the NE and NW regional e-Science centres. It is funded by the DTI e-Science core funding programme.

The aim of OGSA-DAI is to allow a new standard way to access databases and other data-related resources and tools, providing a Grid-enabled middleware implementation of interfaces and services to access and control data sources and sinks extending those defined in the OGSI specification. The services introduced by OGSA-DAI take as input complex XML documents and include:

- *Grid Data Service* (GDS), which provides standard mechanism for accessing data resources;
- *Grid Data Service Factory* (GDSF), which creates a GDS on request;
- *Database Access and Integration Service Group Registry* (DAISGR), which allows clients to search for GDSFs and GDSs that meet their requirements.

Currently, data sources/sinks are restricted to be relational and XML database management systems (DBMS); other data sources, such as file systems, could be accessed through the same interfaces.

OGSA-DAI main functionalities are:

- basic services for accessing data sources within the OGSA framework;
- a higher level of data integration services built on top of the basic services, that allow data federation and distributed queries to take place within a Virtual Organization (VO);
- to leverage emerging Grid infrastructure for security, management, accounting etc.;
- to provide a reference implementation of the Global Grid Forum (GGF) recommendation for Database Access and Integration Services (DAIS);
- to standardize data access interfaces based on the requirements from the GGF DAIS WG.

Although data access has been the primary focus for OGSA-DAI, data integration has also been considered. Currently, OGSA-DAI has only attempting to examine how to run a query across distributed data resources presented to the user as a single data resource.

3.4. Information Services

Information services provide information about resources, including people, machines, software, data, and services, available in a Grid environment. The design of information services is made challenging by the diversity and heterogeneity of resources involved, the range of queries required, and the dynamic nature of VO membership and resource status. Resources can be basic Grid resources or composed resources aggregating a number of basic (or possibly composed) resources, domain independent or provided by a particular application domain.

The Globus information system, MDS-2 (Monitoring and Discovery System, [Mds03]) is based on a directory structure and LDAP data representation and on a hierarchical data model where Grid information is stored into entries collected into virtual organizations and organizational units. The MDS-2 architecture consists of two basic elements [Czaj01]: (i) a distributed set of generic information providers that collect information about Grid resources, and structure them according to the LDAP standard data model, and (ii) higher-level services (e.g. aggregate directory services) that collect, manage and index information. In particular the *Grid Resource Information Service* (GRIS) information provider can answer queries about the resources of a particular Grid node, while the *Grid Index Information Service* (GIIS) combines the information provided by a set of GRIS services managed by a virtual organization.

With respect to basic Grid information systems, a PSE toolkit information system should be enhanced in order to deal with the heterogeneity and complexity of the involved resources. In particular, an accurate metadata management is essential to build efficient information services.

In the remainder of this section, we will first examine basic information services needed in PSE: publishing, discovery, browsing, and querying. Then, we will analyze the information service architecture, i.e. how a number of information servers can be deployed and organized on the Grid, and how such servers can cooperate to provide efficient information services. Finally, we will introduce to the management of information services with the emerging Open Grid Services Architecture.

Basic Information Services

The main information services that should be provided by the Information System of a Grid-based PSE Toolkit are the following:

- Publishing service
- Browsing service
- Discovery service
- Querying service

All these services use the PSE knowledge base (metadata and ontologies) to navigate inside the Grid resources.

The Information System is fed by the publishing service that is used by software developers and host administrators to declare the availability of resources that can help to solve PSE problems. Resources are registered on an index information server accessible to the clients of the PSE toolkit. Due to the large heterogeneity of involved resources, it is essential to operate an accurate categorization of resources. When a new resource is published, it should be associated to a sufficient amount of metadata, in particular classification metadata, in order to facilitate its individuation.

Browsing services permits to view the content of information indices and servers, and to explore the features of resources. The use of browsing can be limitative because it requires a pre-existent knowledge of the servers where needed resources could be found.

The discovery service allows the clients to discover and access resources registered on information servers and possibly located in different Grid hosts. After being

discovered, resources can be viewed by means of browsing services in order to explore their features.

The query service is a high level service that allows for searching resources having particular characteristics: a client can specify resource features by means of powerful query expressions, i.e. written in an XML query language. The query service uses lower level services (e.g. discovery services) to perform search operations, and returns a list of candidate resources that match search criteria.

PSE information services provide access to both application-independent resources, e.g. basic Grid resources such as hosts, networks etc, and to resources belonging to particular application domains. For an efficient management of domain specific resources, it is essential a tight integration between the information services and the ontologies that describe the various application domains. If this integration is well organized, domain scientists and engineers should not access resources via generic information services, but through specific domain ontology services.

As an example, a user should be able to request resources specified not only with syntactic descriptions, but also through semantic information related to the application domain. Such a request is not directly forwarded to an information server, but has to be interpreted and translated by ontology services into a query that only contains conditions and constraints comprehensible for the information services. Analogously, the reply given by the information services should be interpreted and translated by the ontology system prior to be delivered to the user.

Information services should clearly separate the metadata information model and the metadata access model, so that redesigning the access model will not affect the representation of resource metadata. For example, the information model can be XML-based, while the access model can be LDAP if the Globus 2 version is used, or based on web services standards (WSDL, UDDI) if an OGSA compliant architecture is adopted.

The Knowledge Directory Service (KDS) [Cann03b] provides a publishing and discovering service matching the above discussed requirements in the context of a domain-specific (data mining) PSE toolkit, namely the Knowledge Grid. KDS includes: a Knowledge Metadata Repository, which stores XML metadata describing the resources located in a single node; a metadata editor used to create and modify resource metadata; a KDS information provider that reads the metadata information stored into the KMR and publishes it in the GRIS server, in a format specified by the KDS schema.

Even if KDS was developed for a domain-specific PSE toolkit, its structure is flexible enough to be generalized and reused for multi-purpose PSE toolkits.

Architectures for Information Servers

A PSE toolkit designed to be used in different application domains should allow clients to request and access a very large number of resources; the organization of information servers is crucial for the efficient management of resource discovery.

A number of different organizations are used:

- **Hierarchical organization**. Information servers are accessed and queried at different hierarchical levels. For instance, a low level information server can provide detailed information about resources maintained by a single Grid host, while high level information servers can collect information about resources managed by a number of hosts belonging to the same Virtual Organization. In general, a request is made to a low level server (e.g. a Globus GRIS) if the location of the needed resources is known, otherwise the request is forwarded to a higher level server (e.g. a Globus GIIS) to search a wide set of hosts. This scheme requires that a number of information servers register to one or more higher level servers, and resource metadata are replicated according to a push mechanisms (low level servers send metadata to the servers which they are registered to), or a pull mechanism (higher server poll lower level server to obtain metadata).

- **Peer to peer organization**. Opposite to the hierarchical model, information servers are not organized in a hierarchy, but contact each other according to a P2P logic [Iamn02]. Information servers are given equal roles and responsibilities, and resources are published on information servers without a predetermined strategy. When a client wants to perform a query or a discovery operation, it contacts an information server; if this has information about the needed resource, it responds to the client, otherwise it forwards the request to a number of peer information servers, that can recursively contact other peers. This kind of organization can be efficient in a widely distributed and dynamical information system, and scales better than the hierarchical organization because there is no need of powerful and expensive centralized servers. However it can cause an increase of network load and a less predictability of the performance of search operations.

- **Hybrid organization**. This kind of organization aims to exploit both the advantages of hierarchical and P2P models. The concept of superpeer is introduced [Yang03]: a superpeer is an information provider that collects information concerning the hosts of a Virtual Organization, and represents the interface between these hosts and the overall information systems. Superpeers exchange information with each other according to the P2P model. When a superpeer receives a request, it either responds directly, if the information required is possessed by one of the hosts managed by the superpeer, or forwards the requests to other superpeers. A super-peer information system has the potential to combine the efficiency of a centralized search with the autonomy, load-balancing and robustness provided by distributed search.

Hints to the information model of OGSA

The foreseen wide adoption of the OGSA [Ogsa03] architecture based on the web services technology will have an impact on the architecture of the information system, since OGSA allows to expose all services and resources as Grid Services (see Section 4.2). The information model of OGSA is essentially based on two features:

- Information about Grid Service instances is stored into XML-encoded Service Data Elements and can be queried through standard interface methods, such as the FindServiceData operation exposed by the GridService portType.

- Information is collected and indexed by means of Index Services, which replace the MDS-2 GRIS and GIIS services, and can be organized following one the approaches described above (hierarchical, P2P, hybrid). Index Services can be implemented through the mechanisms provided by the Web Services technology, in particular through the UDDI and WSIL standards. UDDI (Universal Description, Discovery and Integration Service [UDDI03]) is a specification for a registry that can be used by a service provider as a place to publish WSDL (Web Service Description Language) documents. WSIL (Web Services Inspection Language [WSIL03]) provides a simple way to find WSDL documents, not listed on UDDI registries, on a web site.

3.5. Resource Management and Scheduling

Resource Management [Nabr03] is fundamental in a distributed computing system for managing a pool of available resources such that a system- or job-centric performance metric can be optimized. A *resource* is intended as a (generally reusable) entity employable to fulfill a *job*. Resources comprise processing units, storage and other devices, network bandwidth, software, data, etc. They are in general very heterogeneous w.r.t. their owner, provider, access policies, performance and costs. The resource management system must take into account such heterogeneity (and possibly leverage it) to effectively exploit resources at best.

A resource management system must meet several requirements in order to be usable in a PSE toolkit, among which: adaptability to different heterogeneous environments; scalability; interoperability among systems with different administrative policies (while preserving autonomy); fault tolerance; support for high-level languages; support of Quality-of-Service; reservation guarantee; monitoring and storing of dynamic information; reliability.

The *Global Grid Forum* [GGF] is currently working on the definition of a resource management architecture that may be used as the basis for a Grid implementation of the PSE Toolkit. The architecture is service-based, and foresees a set of services each of which may have a different internal organization (centralized, decentralized, hierarchical, peer-to-peer, etc.):

- *Data Management Service*, maintaining information about the existence and location of data; may include a *Replica Management Service* granting access to a replica catalog;
- *Network Management Service*, collecting information about network resources and corresponding reservations;
- *Extended Information Service*, extending current information services including dynamic information about data, network and hosts;
- *Job Supervisor Service* monitoring the job schedule before and during its execution;
- *Accounting and Billing Service*, associating resource reservations and usage with costs;
- *Scheduling Service*.

The scheduling service provides one of the most important features of resource management systems, i.e. the capability of “suitably” assigning jobs to resources. In general, a scheduler is a software component which, on the basis of knowledge or

prediction about computational and I/O costs, tries to improve some performance described through an objective function. Depending on the particular function adopted, three main categories of schedulers can be identified. *Job-oriented* schedulers (also known as *high-throughput*) try to enhance the overall performance of the system. *Resource-oriented schedulers* consider e.g. some fairness criteria in resource utilization. *Application-oriented schedulers* optimize the performances of individual applications.

The process of scheduling consists of the following basic steps:

1. Examining the structure of jobs/applications to be scheduled;
2. gathering information about available resources;
3. preselecting resources usable to execute jobs (also known as *location* or *selection*);
4. determining *schedules*, i.e. and assignment of jobs to computing resources along with timing constraints (*mapping*, *allocation*, *placement*), possibly also on the basis of the results coming from the replica manager querying;
5. distributing computation and data;
6. ordering the execution of jobs;
7. supervising the execution and possibly adapting the schedules to new significant occurred events.

In order to effectively perform such steps, the scheduler makes use of:

- A *programming model* provided to the user for building and describing applications. A program may be expressed in dataflow or workflow style, thus mainly using structures as job dependency graphs, or using formalisms specifically targeted to particular computing environments.
- A *resource model* describing characteristics, performance, availability and cost of resources. Schema- or object-oriented data models may be used for this purpose. Important requirements of resource models are the timeframe-specificity of predictions, the use of dynamically-updated information, and the capability to adapt to changes. For resources whose state is not completely known, a predictive estimation may be employed, e.g. based on heuristics, pricing models or probabilistic analysis.
- An application- and/or system-*performance metric*, for measuring the performances the scheduler must improve. A typical application metric is the completion time; the most commonly-used system performance metrics are overall throughput and resource utilization.
- A *scheduling policy*, for driving scheduling actions both w.r.t. their timing (compile-time, run-time, or dynamic with rescheduling), and w.r.t. the used techniques (optimal or sub-optimal, deterministic or non-deterministic, etc.). Exact optimal techniques, such as integer-linear or constraint programming, are rarely usable because of the tremendous size of the search space to be dealt with. Several suitable heuristics are employable, either general-purpose, e.g. based on random/round-robin, basic local search, graph decompositions, genetic algorithms or simulated annealing, or specific, e.g. based on the concepts of priority list and critical path.

In the literature are reported three main architectural models for schedulers in distributed environments. Essentially, each of this model proposes a different way to organize the modules that take decisions, deal with faults, and provide co-allocation.

Centralized model. There is one scheduler that is responsible for the system-wide decision making. Advantages include easy management, simple deployment and the ability to co-allocate resources. The disadvantages of this model are the lack of scalability and fault-tolerance, and the difficulty in accommodating multiple policies.

Hierarchical model. There is a number of schedulers, organized in a hierarchy, where higher level ones manage larger sets of resources and lower level ones manage smaller sets. This model addresses both scalability and fault-tolerance issues. It also retains some of the advantages of the centralized scheme such as co-allocation. One of the key issues with the hierarchical scheme is that it does not provide site autonomy and multi-policy scheduling. This means that the various resources that participate in the system are not able to preserve control over their usage.

Decentralized model. The decentralized model naturally addresses several important issues like fault-tolerance, scalability, site-autonomy, and multi-policy scheduling. However, it introduces several problems like management, usage tracking, and co-allocation. Moreover, it is necessary for the schedulers to coordinate with each other via some form of resource discovery or resource trading protocols. Depending on the overhead introduced by these protocols the scalability of the overall system might be reduced.

It should be noted here that schedulers must often make decisions with an objective different from those of other components (i.e. other schedulers) of a resource managements system. For instance, pushing the performances of single applications may worsen those of other ones and of the overall system (and vice versa); or, some fairness criteria could be established by some components that interfere with application- or job-oriented schedulers; finally, schedulers operating at different levels (local, application or system) could miss to adequately take into account the presence of one other. In the classical scenario of parallel processing, schedulers at all levels make assumptions about the underlying system that are not applicable to the context of distributed/Grid PSE implementations. For instance, they assume to be in control of an entire resource pool, typically invariant and composed of resources having very similar performances. Furthermore, they often disregard the impact other applications running on the system have on the performances of the controlled resources. None of these assumptions is realistic in a Grid context, so for an environment implemented on top of a Grid architecture, the scheduler must take into account the inherent greater heterogeneity.

An application-oriented scheduling model

In the following we propose a model for application-oriented scheduling in distributed heterogeneous environments. The model addresses most of the features a scheduler for a PSE Toolkit must offer in order to increase the performances of generated PSE applications.

In the proposed model, a *host* is described through a set of attribute/value pairs describing the host's location, processor (model, clock, free percentage, count), operating system (name, release, job manager), memory (total RAM, free RAM, total

disk, free disk), etc. A *dataset* has instead a description providing information about the modalities for accessing it (location, size and so on), its logical and/or physical structure, etc. Moreover, a dataset is associated with the host at which it resides. The description of *software* regards the kind of data sources the software works on, the relationships between input and output, etc. A software is associated with the host offering it.

The *job* is the basic building block of applications. It is composed by a software component, a set of input datasets, the parameters to be used and the host at which it has to be executed (except for the case of data movement jobs, where the host is the one towards which a dataset or a software is moved). At any given time, a job is in one of the following states: *abstract* if some of its characteristics are not yet specified, *ready* if it is completely "ground" and thus ready to run (typically this happens after the job has been scheduled), *active* if it is running, *ghost* if its execution has been completed.

Finally, an *application* is a directed acyclic graph (*DAG*) of jobs, or any equivalent representation of a workflow instance. In the application specification, many data movement jobs can be left implicit, that is it is allowed to define jobs indicating their components as defined above, and disregarding the fact that software and/or datasets could require to be moved from different hosts. The data movement operations needed to execute the application are eventually generated accordingly by the scheduler.

Software, data, hosts and jobs are said to be *abstract* if they have only some constraints specified; an application is abstract if it contains at least one abstract resource (see Section 3.2).

The scheduler's input consists of (*i*) the sets of available datasets, software and hosts; (*ii*) the set of jobs to be executed, partially ordered for expressing precedence relationships among jobs; (*iii*) an *execution time estimation function* associating each quadruple <*dset, sw, host, start-time*> with the time needed to execute the software *sw* on the host *host* with the input dataset *dset* starting at time *start-time*; (*iv*) a *communication time estimation function* associating each quadruple <*host-s, host-e, dset-size, start-time*> with the time needed to transfer a dataset of size *dset-size* from host *host-s* to host *host-e*, starting at time *start-time*; (*v*) an *output size estimation function* associating each pair <*dset, sw*> with the size of the output produced by software *sw* when executed on input dataset *dset*.

The scheduler yields as output (*i*) a set of non-abstract datasets, one for each dataset in the input set, possibly along with some dataset copies to be used for improving performances, as reported by the replica manager; (*ii*) a set of non-abstract software codes, obtained as above; (*iii*) a set of non-abstract hosts, one for each host in the input set; (*iv*) a set of non-abstract jobs, one for each job in the input set, possibly adding the needed data movement jobs; (*v*) a *timing function*, associating each job with the time at which it must be started during the application execution.

The scheduler's output must meet several strict requirements. First, resource constraints must be satisfied, i.e. each job has to be really executable (considering the characteristics of software, data and host composing it). Precedence constraints must be satisfied as well, i.e. if a job j_i precedes another job j_k w.r.t. the partial order given in input, then j_k's starting time has to be chosen after j_i's completion (and after

other possible data movement operations). Finally, the overall completion time must be as low as possible.

In the proposed model, no assumption is made on the particular scheduling policy adopted. It should be noted, however, that a good solution is that of providing an open interface for the scheduling policy, so as to have “pluggable” timing and techniques. In the case of a Grid-based implementation, to better take into account the dynamic nature of the environment, it could be useful to evaluate the instantiation of abstract resources not only before the application execution, but also during the execution itself (thus adopting a *dynamic with rescheduling* approach). In fact, during the execution, some resources may become available or unavailable on an unpredictable basis, as e.g. new potentially-candidate resources could be freed from concurrent applications, or previously-chosen resources could fail. A possible solution to this problem could be that of associating a list of matching concrete resources with every abstract one. In this way, should a resource become unavailable during the execution, it could be immediately replaced without re-invoking the scheduler. Furthermore, a number of *resource checkpoints* could be inserted within an application and, when a resource checkpoint is reached, the scheduler is invoked in order to re-schedule the application from the checkpoint on.

Finally, we believe that two possible useful extension of the proposed model are (*i*) the adoption of resource cost functions, in order to have, given a maximum completion time, a set of resource reservations to be made for minimizing the cost of an application execution, and (*ii*) the use of *job checkpoints*, that is the possibility to suspend jobs and re-schedule their execution, if possible without restarting them, for improving performances or in case of failures.

3.6. Steering

Computational steering is an emergent technology that provides a mechanism for the integration of simulation, data analysis, visualization and post-processing. It can be defined as the interactive control over a computational process during the execution. Application for which steering is useful are typically long-running, complex simulation, modeling or control programs executing in parallel or distributed environment. Three types of computational steering exist: *exploratory*, *algorithmic* and *performance* steering.

Exploratory steering allows users to examine the state of a simulation as it proceeds through the interactive visualization of intermediate results and to guide the computation by the modification of input parameters. In this case, the decisional process is carried out by human users who interpret the data visualized and then issue, by a GUI, steering commands.

The **algorithm steering** approach automates the steering loop replacing the human with a decisional algorithm written in a steering language.

Performance steering allows users to change applications to improve application performance. Manual load balancing or re-schedule of the processes of the application are examples of interactive performance steering. In general, performance steering is performed by algorithms. According to this approach, for example the input parameters of the load balancing algorithm can be modified during the runtime of the simulation.

In a PSE toolkit, computational steering can be provided for within the composition interface (see Section 3) by inserting a loop that contains either *a user interface component* (UIC) or a *programmable component* (PC) into the task graph.

The UIC allows to implement the exploratory steering. By UIC a user can switch a steerable component between the paused and non-paused states, change parameters, resolution or representation on the fly and enables the visualization of intermediate results as soon as they are available.

The PC handles in an automatic way the steering of the application. A user defines a program that contains significant events using the variables and the parameters that is interesting to control and allows to take actions when an event is detected.

Performance steering includes the runtime adjustment of performance–relevant parameters to improve application performance. The PC can also be used to define algorithms that decide what steering actions to exercise based on both inputs from the monitoring system and, possibly, inputs from other external sources.

Steering and PSE

We plan to insert all these types of steering in the PSE toolkit. Basic components of the PSE framework can have or not the propriety to be steerable described in its metadata.

When is present a steerable component a user by UIC can receive data from other components (steerable or not), can place sensors and actuators in their source code specifying the program variables that sensors should monitor and the control points that can be modified by actuators. Usually interaction with a running application can occur by changing the input to one or more components by the user interface component.

An algorithmic steering component can be introduced in the architecture of PSE. Using PC we can write, with an appropriate language (what-if conditions, ….), an algorithm driving the computation towards interesting scenarios. The algorithm is used to change adaptation of program components or the decomposition geometry or even to replace some components in order to improve performance or avoid faults.

However, to tackle the Grid dynamicity, it could be useful to re-schedule workflow instances (i.e. to switch one or more concrete resources during the execution), or even to change the workflow schema, in order to force instances to follow a different application logic.

The computational steering can be used to define steerable algorithms that decide how to re-schedule workflow instances or generate new allocation schema by events defined using data coming from the monitoring system and the internal variables or parameters.

4. Open Issues

4.1. Performance Models

Performance models are needed to predict and evaluate the performances of single tasks, of the entire applications and of the overall system. They are useful in different contexts:

1. for the selection and scheduling of appropriate software and hardware resources;
2. for the evaluation of application performances prior to their execution;
3. during application execution, e.g. to facilitate the application steering. to increase performance by dynamically load balancing the system and furthermore for fault detection and diagnosis;
4. after application execution: performances can be evaluated, visualized and used to refine the performance model.

In Figure 5, some architectural modules for performance modeling are shown. With respect to the general PSE Toolkit architecture presented in Section 3, in Figure 5 we detail some components of the Resource/Execution Manager and a Performance Models Repository which is part of the Metadata Repository.

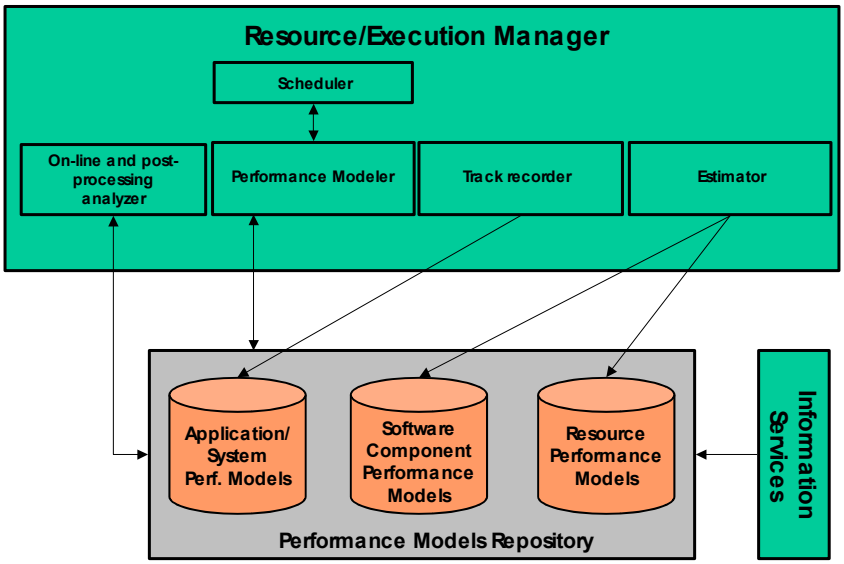

Figure 5. A performance modeling architecture

The performance modeler is the main and more complex component of the performance model architecture. It builds performance models of overall applications using information provided by the repository, containing performance models concerning software and hardware components. Software Component PM are

typically provided by the software engineer building the specific component. Note some components can lack performance models or can have an incomplete one; in such a case the Estimator module (described below) must provide a fairly accurate SPM. Resource PM are supplied by the forecaster module of the Information Services. This module should minimize the intrusiveness, consider dynamic changes over time (load, priority, availability, etc..) and be scalable. In [Bala00] some grid monitoring tools are compared and NWS [Wols99] seems to satisfy these requirements.

On-line and post-processing analyzer pursue a twofold aim: enhancing the performance of the application during the execution, and later analyzing the accuracy of the model and the performance obtained in order to improve future executions. Interactive steering can be useful in changing adaptation of program components or the decomposition geometry or even in replacing some components (see Section 3.6 for more details). Post-processing analysis verifies the validity of the application performance model, not only evaluating the error between the correct behavior and the obtained one, but also permitting corrections by apposite algorithms or by the user itself.

The track recorder stores a history of the Application/System performance models into the repository in order to reuse them in analogous or similar applications. Furthermore, information about the failure of some components or entire applications in particular circumstances can avoid these problems in the future or modify the degree of reliability of the software/hardware component.

The estimator module tries to obtain un alternative performance model when a component has none or an incomplete one. The difficult task is succeeding in forecasting without using many resources and not wasting too much time. The design of this module is still in study; we think to carry out using techniques as the multivariate one, where easy and cheap measures permit to obtain more complex ones.

Hardware component performance models describe the performances of available hardware components such as processing cycles, storage and network links. Performances are evaluated dynamically and are timeframe-specific; predictive estimation about future performances are provided as well.

Software component performance models describe the performances of available software components. Here the word *performances* refers to the performances *required* to execute the components, e.g. with the number of CPU cycles needed to run a certain algorithm as a function of the input size, or the particular hardware characteristics the software is tailored to.

Application/System performance models describe the performances that in general are to be improved by using performance models. A typical application performance model is related to the application completion time, while the most commonly-used system performance models are concerned with overall throughput and resource utilization. The choice of a suitable performance model to be adopted here depends obviously on the particular application and on the software/hardware infrastructure available. Application performance models should be designed to be easily composable, i.e. it should be easy to build the performance model of an application composed of sub-applications with a known performance model, as it is usual in PSE environments to build applications through the composition of pre-existing solution methods.

It should be noted that useful information can be extracted by properly combining the performance models described above. For instance, by combining resource and component performance models, two estimation functions can be built: (i) an execution time estimation function, evaluating the time needed to execute a given software on a given host with a given input dataset; (ii) a communication time estimation function evaluating the time needed to transfer a dataset of a given size from an host to another. These functions, as described in Section 3.5, can provide valuable information to the scheduling component of the Toolkit resource management system.

4.2. Grid Services

Many Grid-based middleware and applications, such as Grid portals, search engines, data Grids, and authorization services, have been developed to provide services on the Grid infrastructure. However, till today all the services provided are separate and not interoperable.

On the basis of the integration and interoperability requirements of the increasing number of applications, Grid technologies are evolving towards an open Grid architecture, called the *Open Grid Services Architecture* (*OGSA*) [Ogsa03], in which a Grid provides an extensible set of services that virtual organizations can aggregate in various ways.

OGSA defines a uniform exposed-service semantics, the so-called *Grid service*, based on concepts and technologies from both the Grid computing and Web services communities. Web services define a technique for describing software components to be accessed, methods for accessing these components, and discovery methods that enable the identification of relevant service providers. Web services are in principle independent from programming languages and system software; standards are being defined within the *World Wide Web Consortium* (*W3C*) [W3C] and other standards bodies. The OGSA model adopts three Web services standards: the *Simple Object Access Protocol* (*SOAP*) [SOAP03], the *Web Services Description Language* (*WSDL*) [WSDL03], and the *Web Services Inspection Language* (*WS-Inspection*) [WSIL03].

Web services and OGSA aim at interoperability between loosely coupled services independent of implementation, location, or platform. OGSA defines standard mechanisms for creating, naming, and discovering persistent and transient Grid service instances, provides location transparency and multiple protocol bindings for service instances, and supports integration with underlying native platform facilities. The OGSA effort aims to define a common resource model that is an abstract representation of both real resources, such as processors, processes, disks, file systems, and logical resources. It provides some common operations and supports multiple underlying resource models representing resources as service instances.

Differently from Web services, that address discovery and invocation of persistent services, the OGSA model also support *transient* services instances, created and destroyed dynamically. Thus a Grid service is a, potentially transient, Web service based on Grid protocols expressed using WSDL.

In OGSA all services adhere to specified Grid service interfaces and behaviors defined in terms of WSDL interfaces and conventions and mechanisms required for creating and composing sophisticated distributed systems. Service bindings can

support reliable invocation, authentication, authorization, and delegation. To this end, OGSA defines a Grid service as a Web service that follows specific conventions on the use for Grid computing and provides a set of well-defined WSDL interfaces.

Grid services implement one or more interfaces, corresponding to WSDL *portTypes*. Only one interface is mandatory (*GridService*) and deals with querying information sources about service instances (through the *FindServiceData* operation) and managing their termination (*setTerminationTime* and *Destroy* operations). Other (optional) interfaces are:

- *Factory*, dealing with the creation of service instances;
- *HandleResolver*, returning *Grid Service References*, containing protocol- or instance-specific information about services, associated with *Grid Service Handles*, i.e. unique URL associated with services for their entire lifecycle;
- *Registration,* for soft-state (un-)registration of Grid service handles;
- *NotificationSource* for the subscription to notifications of service-related events, based on message type and interest statement;
- *NotificationSink* for the asynchronous delivery of notifications.

As the OGSA model is under development, standard interfaces for authorization, policy management, concurrency control, and the monitoring and management of potentially large sets of Grid service instances will be defined in the near future.

The OGSA model has been adopted by the Globus project for the designing of the GT3 (Globus Toolkit 3) environment [Ogsa03]. Technically, OGSA enables the refactoring of Globus protocols (GRAM, MDS, GridFTP) through the Grid Services technology, while preserving all GT features.

Figure 6 shows the GT-OGSA architecture. Grid Services can be defined at three different abstraction levels, and consequently belong to three layers: the System-Level Services layer, the Base Service layer and the User-Defined Services layer.

System Level Services are general-purpose services that facilitate the use of other Grid Services in production environments. The GT3 distribution currently includes an Administration Service, a Logging Service and a Management Service.

Base Services include Grid Services that implement basic functionalities of the Globus Toolkit formerly implemented with other technologies. They include:

- Resource Management Grid Services, which are used for job submission and management and correspond to the GT2 (Globus Toolkit 2) GRAM architecture;
- Information Services. In particular, the Index Service is used as a caching aggregator service that caches service data form other grid services, and as a host for service providers. The Index Service replaces the GRIS and GIIS information providers of GT2.
- Reliable File Transfer (RFT), which exposes GridFTP control channel functionalities. Note that, while RFT is a Grid Service, the GridFTP client and server are still GT2 compliant.

User Defined Services can be defined and published by end users using GT3 development tools. Very interesting is the opportunity of defining a Grid Service that is a specialization or a composition of existing Grid Services, thus enabling the reuse of the newly defined Grid Service.

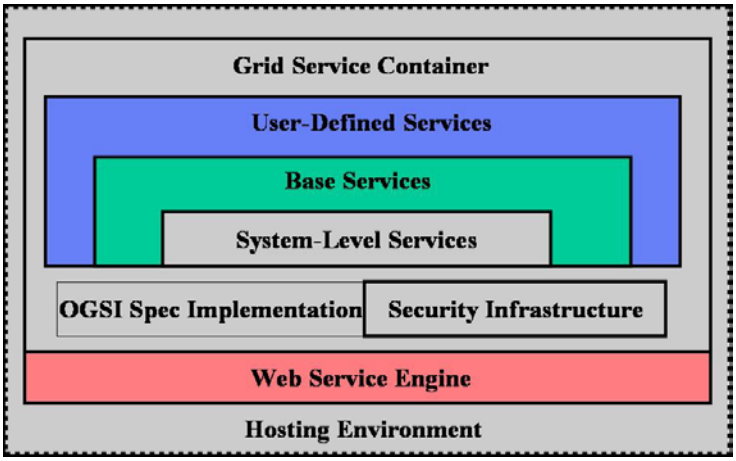

Figure 6. A picture of the Globus Toolkit OGSA architecture

Of course, a large number and variety of User-Defined Grid Services can be defined by PSE and application domain experts in order to facilitate PSE users in building specific solutions. Therefore it is imperative to provide an efficient information system that assists users in the task of discovering the User-Defined Grid Services that better can satisfy their requirements. To this end, two issues have to be tackled:

1. information providers (Index Services in OGSA) should be efficiently interrelated and organized in a hierarchical, peer-to-peer, or hybrid structure;
2. an efficient metadata model should provide a useful representation of heterogeneous Grid Services. In [Mast03] a metadata model is defined to classify resources utilized in the context of a PSE toolkit focused on data mining applications. This model can be generalized to classify Grid Services that interface different kinds of resources, in particular hardware resources, software, data sources, workflows.

The PSE Toolkit architecture proposed in Section 3 can be designed to comply with the OGSA model, in terms of both internal and external interfaces. More specifically:

- The Metadata Repository could be wrapped and made accessible through OGSA database services, allowing for querying and browsing data.
- In the Component Repository, data could be made accessible through OGSA database services as well, while software components should provide OGSA interfaces, possibly wrapping their native ones. Furthermore, as it is natural in OGSA, processing cycles and network bandwidth could be reserved and used as services.
- The features provided by the Description System, the Search/Discovery System and the Execution/Resource Manager are directly exposable as Grid Services.
- Finally, a service could be provided for accepting user requests about more complex tasks, such as the composition of applications and their steering. This

"User Service" should also be made extensible through an open interface to let users specify and automate their usage patterns or repetitive actions, such as the automatic refresh of metadata about provided components, or the extraction of new potentially useful information from the Metadata Repository.

As a final remark, the adoption of the Grid Services technology has an important impact on all the topics and issues involved in the designing of a PSE toolkit. In particular, Grid Portals are a privileged interface to Grid Services, since portlets can be associated to single or collective Grid Services: see Section 4.7 for more details.

4.3. Ontologies

"An ontology is an explicit specification of a conceptualization" [Grub93]. Specification refers to an explicit representation by some syntactic means. Ontologies try to capture the semantics of domain expertise by deploying knowledge representation primitives, enabling a machine to understand the relationships between concepts in a domain. Additional knowledge can be captured by logical axioms or rules which derive new facts from the existing ones. An inference engine then can draw conclusions based on the rules or axioms to create new knowledge and eventually to solve problems.

In other words, Ontology is a shared understanding of some domains of interest, which is often conceived as a set of classes (concepts), relations, functions, axioms and instances. Concepts in the ontology are usually organised in taxonomies.

Knowledge Management in a PSE

Problem Solving Environments (PSEs) are becoming more and more complex and knowledge intensive. To produce satisfactory results, a PSE should be designed using the best domain practice and following decisions made by skilled engineers in practical situations. A knowledge based approach could be used to acquire this knowledge from existing sources and model it in a readable way.

Ontologies can be used for the knowledge management in a Grid-based PSE environment allowing the building of semantically enriched knowledge bases. These knowledge bases can be regarded as the basic resources for the various knowledge services available and integrated into a PSE to assist users to exploit resources for the design and execution of applications.

A Grid-based PSE is composed of a set of PSE specific components such as software libraries, repositories, and all the resources associated to a Grid environment. Moreover a PSE could be tailored towards a specific domain thus we have to manage domain specific knowledge too. In summary, in a Grid-based PSE we have to model knowledge about:

- PSE specific application domain
- PSE components
- Grid resources and services

To reduce the complexity of resource modeling and to leverage existing ontologies, a suite of ontologies could be built (or reused) for the management of the heterogeneous knowledge in a Grid-based PSE, providing a full range of descriptions

for the full range of users and domains expertise. In this direction, it is possible to identify the following ontologies.

Domain, application and task ontologies. A first set of ontologies should model the basic features of the specific application domain as well as the components of the PSE allowing the development of systems that perform the functions of a domain expert. These functions refer to what things are in the domain and when and how these things are related (*domain ontology*). Moreover the descriptions of relevant tasks of the domain (e.g. retrieval and analysis) should also be modelled. To this aim a mixture of all the different aspects of the domain should be modelled: general PSE knowledge, domain specific knowledge, domain specific tasks and applications. The *domain ontology* should be used to model all the resources of the PSE as software, data and host Since a PSE toolkit is used for different application domains, the domain ontology can be composed of two parts, a specific domain part and a core part. The domain specific part is an explicit description of domain specific terms, characteristics and components, whereas the core part models knowledge common to different application domains and provides domain independent primitives to build domain specific ontology.

The core part should describe the common basic structure of PSE components characterizing their input/output interfaces and their semantics in the following terms: (1) the task performed by the component; (2) the problem it solves, if the component is a complete application; (3) specification of who may use the component; (4) how information is passed from the component to another one; (5) a set of constraints associated to the component such as on what platforms it is licensed to run, whether it requires generic software to be linked later in order to run, the kind of input it requires; (6) information on a component's purpose, the algorithm it uses, and other pertinent explanatory data optionally associated with a component; (7) how a piece of executable may be associated with a component: e.g. the component may be available only as an executable specific to a certain type of hardware, or the component may be available as source code and in this case the PSE must arrange for it to be compiled for a target architecture; (8) the server implementation to which the component is linked to be executed [Walk00]. A *task ontology* is an ontology specifying a problem solving process in terms of concepts and relations appearing in a task of interest. The advantage of task ontology is that it specifies not only skeleton of the problem solving process but also context where domain concepts are used. An *application ontology* describes and classifies applications represented as workflows and contains information about application's results and comments about user experience. Application ontologies describe concepts depending of both a particular domain and a task and are usually a specialization of them. An application is the composition of more tasks.

Grid resource ontology. The ontology for Grid resources can be useful for the resources matchmaking [Paol02, Tang03] and scheduling. At any given time the ontology should maintain an accurate picture of the capabilities, loads and accessibility of the hardware resources available and the network connecting them. This ontology should provide an abstract model for describing resources (e.g., computer system, operating system), their properties (e.g., the total physical memory of operating systems) and their relationships (e.g., a particular operating system must run on a particular computer system). The ontology should provide resource information including configuration details about resources such as CPU speed, disk and memory space, number of nodes in a parallel computer, or the number and type of network interfaces available; instantaneous performance information, such as point-to-point network latency, available network bandwidth, and CPU load.

Ontology-based Operations

Once the semantic knowledge bases have been built, there are several ways in which a PSE can take advantage of ontologies reasoning (inference):

- Formulation of the problem to be solved in terms of modeled tasks, components or pre-existing applications;
- Semantic search along the ontology for the location and access of components needed to solve a problem.
- Scheduling of Grid jobs, e.g. software components, on the basis of modeled scheduling policies.
- Request/resource matchmaking. Ontologies describing resources characteristics and job requests can be used to perform semantic resource selections. Before a resource can be allocated to satisfy a request (e.g. find a node where to run a software component), the system has to match request requirements with available resources characteristics.

Problem formulation

Ontologies can help users in problem formulation, offering a view of the best domain practices to solve problems in that domain. In this way the accumulated knowledge can be accessed by new PSE users.

Semantic Search

Semantic search discovers semantically equivalent or related resources [Cann03a]. This approach differs from traditional crawler based search mechanisms because it relies on resources expressed as a combination of entities and relationships connecting them. Semantically equivalent terms convey similar meanings in terms of semantics although being syntactically unequal. A user through an ontology-based search engine can query very detailed information about resources annotated in the ontology. The result set of the query is very accurate, because the semantic content of the terms searched is clearly indicated by concepts from the underlying ontology. An ontology-based search engine will support several kinds of simple inference that can serve to broaden queries including equivalence, inversion, generalization, and specialization. *Equivalence* relations are used to restate queries that differ only in form, whereas *generalization* and *specialization* relations are utilized to find matches or more general or more specific classes and relations (e.g., in DAML+OIL [DAML03, DAML03a] *subPropertyOf* and *subClassOf* relations). If the result set of a query is empty, the user can at least find objects that partially satisfy the query: some classes can be replaced by their superclasses or subclasses. Both narrowing and broadening the scope of the query are possible due to the ontological nature of the domain description.

Moreover an ontology-based tool can also help the user in the query formulation. Users encounter difficulties when having to provide terms that best describe their information need (vocabulary problem). In ontologies the classes that describe the domain of interest are explicitly shown, making the vocabulary choice much easier.

Matchmaking

The request/resource matching problem involves assigning resources to jobs in order to satisfy jobs requirements and resources policies [Tang03]. Ontologies can be created to explicitly describe resources and job requests allowing to perform semantic matching using terms defined in those ontologies. Resource descriptions (e.g. the Grid resource ontology described above), resource advertisements and job requests should be expressed as concepts of ontologies while matchmaking rules should be added to the ontology to define when a resource matches a job description (i.e. a request). Thus, to realize ontology-based request/resource matchmaking, an ontology describing requests, properties of the request (such as the request's owner), characteristics of the request (e.g., the type of the job requested) and the resource requirements (e.g., number of CPUs, minimum physical memory dimension) should be provided (*Request ontology*). Moreover an ontology capturing the resource authorization and usage policies (e.g., a set of accounts that are authorized to access a specified computer system) is also needed.

Scheduling

The scheduler has to determine the right scheduling policy for a given job as well as on which resources to run the components of the job. An ontology describing scheduling policies and specific terminologies could be used to choose the best policy for a given application class (for example data- vs CPU-intensive applications), whereas an ontology-based matchmaker could solve the job/resource mapping problem. Ontology of the Grid Scheduling domain, such that addressed by the Grid Scheduling Ontology Working Group [GSO-WG03], should support the scheduling of Grid resources done by local and distributed instances of software subsystems like schedulers and brokers. The ontology will allow classification of schedulers, reasoning about schedulers or mapping semantics of different scheduling systems. The ontology will be based on the Grid Scheduling Dictionary developed by the Grid Scheduling Dictionary Working Group [SD-WG03].

Ontology-driven User Interfaces

An ontology-driven PSE uses ontologies to offer to users a sort of expert assistance: to guide users in the task composition process, to help users to search and use components, to match user requirements to component types. This expert assistance could be made available through a graphical interface through which ontology authors can create, import and edit ontologies, and at the same time end users can describe domain and problem solving knowledge based on those ontologies. The interface should support the exposure of the ontological model and the query formulation. During the query formulation process the model may be browsed to find what can sensibly be said of a concept of interest.

The PSE should allow end users: (i) to describe their own problem solving processes in terms of human friendly primitives; (2) to observe the task execution process and debug their own description. In terms of ontology, the environment will be able to capture the end users' conceptual model of problem solving on the level of abstraction and provide them with useful programming guidance.

In the formulation of the problem, the work of an end-user is to specify his/her own problem knowledge. An ontology-based broker could mediate between end-users and the PSE [Benj98]. Basically it has to provide support in building or reusing a

domain ontology and in relating this ontology to an ontology that describe generic classes of application problems. This application ontology has to be linked with problem-solving method-specific ontologies (*Task ontologies*) that allow the selection of a solution method. The broker uses three different ontologies: domain ontologies, applicaton ontologies, and task ontologies. These ontologies provide the contents of the communication processes between users and the broker. First, end users could fill in some fields in the visual workplace for a problem specification. Then, to lighten their initial load, the broker should retrieve a set of similar task cases from the PSE library based on the specification and show them to end users. End users can refer to or reuse them to describe their own problem solving knowledge. During the composition of a task description, an end-user inputs the domain process using the domain ontology terminologies then selects the appropriate terms from the vocabulary (description of generic processes) shown in the task ontology browser and composes the generic process sentences.

A typical communication among end users and a PSE by means of user interfaces driven by a broker is characterized by the following communication flows:

- **Sending a *request* from the user to the broker**. Terms of the domain and application ontologies are the content of the message. The user interface uses a domain ontology to guide the interaction process with the user and it sends selected expressions to the broker.
- **Sending a *negotiation* from the broker to the user interface.** The broker might need further clarification from the user before it can finish the selection process of problem-solving method. This clarification may ask more precise definitions of terms and their relationships necessary to derive an element of the application/task ontology from an element of the domain ontology.
- **Sending a *query* from the broker to a component interface**. After having translated the domain-specific terminology into a problem-specific terminology the broker has to derive an expression in a problem-solving method-specific ontology. Then, this expression is passed as a query to the component interface.
- **Sending a *response* from a component interface to the broker**. The response of a component interface may have two forms: providing a simple yes that the required service can be provided or the wish to introduce further assumptions on the problem that make it tractable and/or the introduction of requirements on domain knowledge that has to be provided by the user.

Ontology Languages and Tools

At this stage major efforts regard the development of languages and technologies for the standard and agreed modeling and implementation of metadata and ontologies.

Currently the most common used ontology languages are the following:

- DAML+OIL [DAML03a] is an ontology language designed for the Web built upon XML (eXtensible Markup Language) and RDF (Resource Description Framework) [XML, RDF]. DAML+OIL is modeled through an object-oriented approach, and the structure of the domain is described in terms of classes and properties. The axioms supported by DAML+OIL allow to assert *subsumption* or *equivalence* with respect to classes or properties, the

disjointness of classes, the *equivalence* or *non-equivalence* of individuals, and various properties of properties. Classes can be combined using conjunction, disjunction and negation. Within properties both universal and existential quantification are allowed, as well as more exact cardinality constraints. Range and domain restrictions are allowed in the definition of properties, which themselves can be arranged in hierarchies.

- The Web Ontology Language (OWL) [OWL03] is a semantic markup language for publishing and sharing ontologies on the World Wide Web. OWL is developed as a vocabulary extension of RDF and is derived from the DAML+OIL Web Ontology Language. OWL adds more vocabulary for describing properties and classes, among others: cardinality (e.g. "exactly one"), equality, richer typing of properties, characteristics of properties (e.g. symmetry), and enumerated classes.

Some examples of tools and techniques for ontology manipulation and navigation are the following:

- *Ontology editing tools* that allow users to define and edit ontologies (e.g. DUET, OilEd, OntoEdit). OilEd [Bech01] is a simple graphical tool that supports the construction of OIL/DAML+OIL/OWL-based ontologies. Basic OilEd functionalities allow the definition and description of classes, properties, individuals and axioms through graphical means. OilEd uses FaCT reasoner which allows the user to produce classification hierarchies and check classes for inconsistency.
- *Ontology manipulation tools* that allow navigating, querying and manipulating ontologies. Jena is an open source Java framework for building Semantic Web applications. It provides a programmatic environment for RDF, RDF Schema and OWL, including a rule-based inference engine [Jena03]. The DAML API is a collection of Java interfaces and utility classes that implements an interface for using and managing DAML ontologies [DAML03A].
- *Ontology-based annotation tools*, for annotating web resources according to an ontology. For example, the UML Based Ontology Toolset (UBOT) [UBOT03] supports translation from UML class diagrams to DAML ontologies.
- *Ontology learning tools*, for learning ontologies from natural language documents (e.g. Corporum, Text-To-Onto); CORPORUM is a document and information management system [Corp00]. The CORPORUM system is founded on CognIT's Mímír technology developed in Norwegian research labs. This technology focuses on meaningful content rather than odd data or standardized document parameters. The Text-To-Onto system provides an integrated environment for the task of learning ontologies learning from text [Text03].

To address the ontology browsing and querying in [Cann03a] we have designed DAMON-MAP, a tool that allows the manipulation of DAML+OIL encoded ontologies, in particular DAMON (Data Mining Ontology), an ontology for the Data Mining domain. The manipulation of an ontology can be realized both by a user that accesses DAMON-MAP through a graphical user interface, and by a Java-based component through DAMON APIs [DAML03a]. The API implementation is realized for accessing and querying the ontology: the API will provide a set of object-oriented

abstractions of ontology elements such as Concept, Relation, Properties, and Instance objects providing query facilities.

4.4. Agents

Generally, a PSE contains:

- *application development tools* that enable an end user to construct new applications, or integrate libraries from existing applications and
- *development tools* that enable the execution of the application on a set of resources.

This means that a PSE must include *resource management tools* in addition to *application construction* tool, albeit in an integrated way. Moreover, a Grid-based PSE must consider the ever-changing nature of their resource base and so the resource management must become more flexible and responsive to changing resource availability and resource demands.

Areas as *resource discovery*, *resource monitoring* and *software propagation* are some of the basic functions that require decentralized, scalable, dynamic and fault tolerant solutions for a efficient resource management on a infrastructure for grid computing.

Component based implementation technologies provide a useful way of achieving this objective, and have been the focus of research in PSE infrastructure. However, today's software systems are becoming more net-centric, distributed, and heterogeneous and require a more sophisticated paradigm to handle with dynamicity, adaptivity and fault tolerance.

A solution to design these new services can be the extension of the component model to **mobile agents** that provide interesting features to describe these services. For example, in MyGrid project, a bioinformatics grid, the experiments are expressed as a workflow script by the scientist. Services can be viewed as being provided by agents and workflow can be seen as an agent interaction script. Agents are able to complex interactions that include negotiation and collaboration, which allow agents to adapt their behavior to the environment.

Mobile agents are flexible, autonomous components that are able to cope with dynamism and openness typical of the grid.

A mobile agent has the unique property that during its lifetime it can be halted, its state and code moved to another computer on the same network, and then continue executing from where it stopped executing on the previous computer.

A mobile agent is ***autonomous*** because it may decide itself where it will go, what it will do there, and how long it will exist for. However, its environment or other mobile agents may also influence it.

Mobile agents are ***asynchronous***. Therefore when a mobile agent is dispatched there is no need to wait for it to return. Indeed the original node does not even to remain connected to the network while the mobile agents are out. The mobile agents can wait until original node is back on the network before attempting to return to it.

Information is being disseminated at every node that the mobile agent visits. Ever node benefits from accepting a visiting mobile agent, because the mobile agent will

have either new or more recent information about resources. Also, every agent benefits from visiting a node because it will learn of either new or updated resources. If the mobile agents do not contain any new information they may be destroyed.

Mobile agents may easily be ***cloned*** and dispatched in different directions. This allows them to function in parallel. Although this causes more mobile agents to be active on the network, it does ensure that the network resource discovery is completed sooner, and therefore the mobile agents spend less time on the network.

A mobile agent based solution is very ***fault tolerant***. Even if some of the mobile agents are destroyed, all surviving ones will have a positive impact.

There are two possible way to design mobile agents for complex problem solving

- to create a few highly advanced (intelligent) agents or
- try to divide and distribute the problem solving task between a number of less complex agents.

First approach requires the construction of complex agents that are based on knowledge models and reasoning technique that belong to the field of Artificial Intelligence.

The latter strategy consists in the use of a high number of (unintelligent) simple agents managed collectively to perform a number of complex task. This kind of collective intelligence is termed **swarm intelligence.**

Swarm Intelligence (SI) is the property of a system whereby the collective behaviours of (unsophisticated) mobile agents interacting locally with their environment cause coherent functional global patterns to emerge. SI provides a basis with which it is possible to explore collective (or distributed) problem solving without centralized control or the provision of a global model.

An *ant colony* is such an example where a high number of less intelligent creatures manage collectively to perform a number of complex tasks.

Swarming agents are mobile agents that exhibit a collective intelligent behavior that may be used to define new decentralized, adaptive, self-organizing and fault tolerant algorithms for resource management on a Computational Grid. A swarming agent is an entity capable of sensing its environment and undertaking simple processing of environmental observations in order to perform an action chosen from those available to it. These actions may include modification of the environment in which the agent operates.

Intelligent behavior frequently arises through indirect communication between the agents using the principle of stigmergy, where something is deposited in the environment that makes no direct contribution to the task being undertaken but is used to influence the subsequent behavior that is task related. Stigmergy is for example used in ants colonies, termites, etc. This mechanism is a powerful principle of cooperation. It is based on the use of the environment as a medium of inscription of past behavior effects, to influence the future ones.

Swarming agents draw inspiration from biological processes and develop techniques and tools for building robust, self-organizing and self-repairing algorithm as ensembles of autonomous agents that mimic the behavior of social insects. What renders this approach particularly attractive from a computational grid perspective is

that global properties like adaptation, self-organization and robustness are achieved without explicitly programming them into the individual artificial agents.

Swarming agents can be used for

- search resources over a grid. In this case the agents can use a foraging behavior. Foraging search is an attempt to emulate animals that solve the above problem in hunting for food, water, mates and other resources.
- resource discovery, to maintain an up-to-date picture of the resources available to the PSE and to seek out new resources. Discovery is a special case of search it differs from the generic search function because the resources of interest are specified very broadly and the search of locations of interest proceeds locally.
- resource monitoring: resources used by the PSE should be monitored, e.g. for rescheduling purposes. Monitoring can be carried out through the production of local indices, obtained from local measurements without recourse to a centralized mechanism.
- software propagation: having the source code available, agents should compile and install a software component on remote PSE hosts.

4.5. Security

Security issues should be managed at two different levels:

1. Grid Security Services, i.e. at infrastructure level. The Globus GSI [GGSI03] allows to perform common tasks in a secure environment. The security infrastructure is used to specify what resources are shared, and what users can access them. Data are transferred in a secure environment and a proxy mechanism is used to assure the Single Sign On feature.

2. More sophisticated security issues are to be tackled at a higher level. The PSE toolkit should allow the definition and management of classes of users ("roles") that are given different tasks and authorizations. Classes of users that can be specified here are: application domain managers, solution designers, supervisors, end users etc.

Security is an important issue that has to be considered in any distributed environments, i.e. any time users share data and applications (or services) in a common environment we may define security policies to allow access and using resources. In a distributed Grid-based Problem Solving Environment, even documents describing data and services location needs to be protected by unsafe accesses. Main security topics may be identified with (but not limited to) protecting: data, software components, and resource locations from malicious access and use. Thus, managing policy has to be defined to guarantee security policy in a distributed (Grid based) PSE for:

1. resources location and definition;
2. access and authentication policy for data (read, write and update);
3. use and execution of remote applications (job applications);
4. secure message and I/O passing.

The first point is associated to adding protection to resource location and rights definition (i.e., what are the resources and which users may access to them). Applications asking access to data may embed user signature to identify the access and prevent unauthorized access. Communication access may be protected using secure access procedures (see [Milo99]) Safety access must be considered as an application layer in any node interface.

Applications in a Grid based PSE, may run on a server node of the Grid returning data results to the client node. Security policy must be identified on a server side to identify the client and to authenticate it hosting and running the applications. Output results need to be packed with client identification and returned to it. Identifying client or users with their rights to access to data or software resources, needs to be faced in a Grid environments where authentication involves many nodes that can be at the same time client and servers. (see [Lasz01]). Authentication processes are necessary to control access to nodes. Grid security infrastructure (GSI) policy authentication is used in [Lasz01], where Accessing a node in a Grid, needs authentication process. Authentication is the process to verify the identity of an entity. It is a challenging task to use standard algorithms for security goals in Grid-based Problem Solving Environments, where users and resources are potentially large and dynamic.

Security coupling may be defined while any node asks for accessing to data or software resources, using authentication and authorization policies. A node asks for a service identifying itself, and the answering node identifies it and authorizes access or job execution following security roles. In our environment it is necessary a standard protocol to identify nodes and client (identification) and to define authorization policy. Currently Globus GSI [GGSI] uses the Secure Sockets Layer (SSL) for its mutual authentication protocol. SSL is also known by a new, IETF standard name: Transport Layer Security, or TLS. Indeed, even authorization in using resources may be defined in levels.

In security policy suggested by Globus, authorization to use a particular Grid resource can be controlled via a grid-map file and appropriately specified group permissions controlled by the local system administrators.

Finally, we may take account of main guideline of GGF activities in Security area. The security working group is considering Web services experiences for managing security policy (see www.ggf.org, security working group). Key features for securing data and component accessing has to include [Mala03] :

Establishing the identity (e.g., the person) that is accessing data and resources in order to enforce access policies based on that identity.

Ensuring data isolation, e.g. ensure that an application is able to access data associated with another application executing within the same environment.

Enforcing data privacy such that only users who have rights to view their own data or particular data (e.g., medical records) can view such data.

4.6. Roles

Users in a Grid Based PSE should have roles related to their tasks and positions within the grid. These roles can give information on allowed resources, available

information and possible ability of readjusting assumptions of the problem-solving environment.

As in Database community, roles depend on kind of users in PSE. In a distributed environment, sharing data and software resources has to be managed by a set of rules that classify users by their capabilities and rights on resources (e.g., read, write, update, generate resource databases, and so on). (group of) Users may have different roles and different access level facilities to the system functionality. According to [Lasz01], users (and group of users) in a PSE can be classified as:

- *Novice* science or problem solving environment users;
- *Expert* science or problem solving environment users;
- *Developers* of application.

Following such distinction, first class refers to novice users able to use available solutions in PSE without knowing the strategies adopted. Expert science are users who know a specific domain and use furnished solutions, eventually extending the PSE resources, e.g., adding data components, software or experimental results. They can also provide data classifications or information to ontologies. Developers users provide components used by other users.

Roles must have a hierarchical organizations reflecting kind of users. The decision power associated with the roles is derived from these positions. This determines the ability of setting and adjusting goals and associated targets.

4.7. Grid Portals

A Grid portal is a web portal specifically designed to provide a single and ubiquitous point of access to a large number of Grid resources and services. A Grid portal hides the details of the Grid environment and offers a uniform and graphical interface as well as high-level functionalities, thus helping the domain expert in concentrating about the nature of the problem and not about the technical issues involved.

A well designed Grid portal should meet the following requirements [Lasz01], [Thom01]:

- *Universal access*: portals should be accessed anywhere and by any kind of web browsers, without the need of requiring downloads, plug-ins or helper applications.
- *Problem-oriented*. A Grid portal should provide functionalities that help users in building solutions to domain specific problems.
- *Use of common Grid technologies and standards* to minimize the resource administration burden.
- *Support of a flexible and scalable infrastructure* to facilitate the updating of Grid resources and services as well as the adding/removing of Grid users.
- *Security.* A Grid portal should guarantee a strong authentication mechanism to prevent unauthorized access to the Grid.
- *Client applications and portals services should be able to run on separate webservers*, enabling scientists to build their own application portals and use existing portals for common infrastructure services.

Services provided by Grid Portals

A Grid portal designed for a particular scientific domain, or as the portal of a multi-purpose PSE toolkit, should provide access to generic PSE services (in this sense it can be seen as a horizontal portal), and to domain specific services (in this sense it is a vertical portal for that domain). Services can be roughly categorized in four classes [Lasz01]:

1. **Common Web portal service.** These services include filterable e-mail, collaborative services, personalized information services, advanced search engines etc.
2. **Grid-oriented services.** These include all kinds of services provided by the Grid infrastructure, either client-server services or Grid Services defined within the OGSA architecture. As we will see later, Grid Services are particularly easy to integrate in the Grid Portal architecture.
3. **Multi-purpose GUI components.** These services are defined to be used in generic PSE toolkits, not tailored to a specific application domain: they include resource metadata editors and browsers, job and workflow editors and browsers, monitoring tools etc.
4. **Application-specific GUI components**, designed for specific application domains. Examples are a stock market monitor, a visual tool for designing data mining applications, a specialized search engine for climate data etc.

In particular, main Grid-oriented services provided by a Grid Portal are the following:

- *Security services* (e.g. Globus GSI services) provide user's authentication and authorization with the system.
- *Job submission and monitoring*. The portal provides job submission and monitoring services. To execute a job through a Grid Portal, the following steps are required: (1) the user authenticates with the system, (2) the user's environment is established, (3) the user proxy is verified or recreated, (4) the job command is parsed, (5) the command is issued to the remote host (6) job state is monitored during the execution, and (7) results are parsed, formatted, and returned to the web browser on the user's workstation.
- *Remote File Management*. Portal users must be able to move files between local and remote Grid hosts. They must also be able to manipulate remote files transparently. Grid Portal supports file transfer, third-party file transfer, parallel file transfer and file browsing capabilities.
- *Context Management*. The context of a Grid portal user consists of remote files, on-line data sources, directory services and applications, and remotely running jobs and workflows distributed over the entire set of Grid resources. In order to manage the user's context the portal has to store information about recent Grid objects created or destroyed by the user while interacting with the Grid.
- *Information services*. Grid Portal provides access to the Grid information services (e.g. the Globus Monitoring and Discovery service), that allow users to publish and retrieve static and dynamic information about Grid resources.

A general architecture of a Grid Portal could be the three tier one shown in Figure 7 [Gann03]

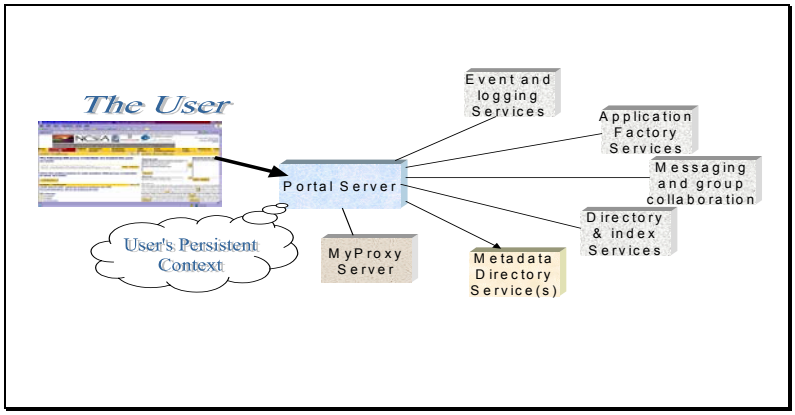

Figure 7. The three-tier architecture of a Grid portal

The first tier includes the end-user's workstation running a web browser. The middle tier (Portal Server) is a secure web application server responsible for handling HTTP requests coming from the client browser. The Portal Server mediates between client requests and Grid services and invokes Grid services on user's behalf. The third tier is the computational Grid that provides Grid services and Grid computational resources.

Grid Portal designing and modeling

Given the large number and variety of services offered by a Grid Portal, it is convenient to exploit a designing methodology based on autonomous and reusable components. This can be accomplished with portlets [Gann03]. A portlet is a portal server component that provides a basic functionality rendered in a user configurable window within a portal pane (a portal pane is a view of the portal provided to a user or a set of users).

Each portlet can be associated to a Grid Service, with the following benefits:

- it is very easy to add new services;
- different research groups can contribute portlets which can be plugged into a portal;
- each user or group can select and configure the portlets he/she wishes to use;

Of course not every Grid Service should be associated to a portlet: many services are designed to be invoked only by other services. So, a portlet can act as an agent that manages a set of services on behalf of the user. Currently, many research groups use the portlet model for their portal servers: examples are the Indiana, Argonne, Michigan, NCSA and Texas+GridSphere portal groups.

The MVC (Model View Controller) model is currently used by all main portal groups to coordinate the execution of portlets, and hence of portal services. This MVC model is an efficient way to separate graphical issues from the application logic and control.

- The View of an application defines the corresponding user interface
- The Model defines the business logic of an application
- The Controller is the entity that sequences view-model interactions

If using the portlet model, this means:

- use a markup/template language to describe the details of how information is presented to the users
- use a set of back-end classes to define the way the portlet does its computations
- let the control of the two be handled by a third party like application or by a specialized “action” class.

Grid Portal technology

In the last years a number of frameworks and toolkits have been developed to assist users in modeling and developing Grid Portals:

- the Apache Jakarta Jetspeed [Jets03] engine is based on the Java portlet model. Jetspeed is an Open Source implementation of an Enterprise Information Portal, using Java and XML. The data presented via Jetspeed is independent of content type: for example XML or RSS data can be integrated with Jetspeed. The actual presentation of the data is handled via XSL and delivered to the user.
- the Commodity Grid (CoG) [Lasz00] toolkits provide a set of programmatically accessible Grid services. The CoG toolkit defines and implements a set of general components mapping Grid functionalities onto different application development frameworks (Java, C, CORBA).
- the Grid Portal Development Kit (GPDK) [Gpdk03] provides high-level services, based on the Java CoG toolkit, to build Grid Portals. In particular the kit offers reusable components for accessing Grid services exposed by the Globus middleware.
- the Grid Portal Toolkit (GridPort) [Thom01] is a collection of technologies designed to aid in the development of Grid Portals. GridPort is based on PKI and Globus technologies to provide secure, interactive Grid services. The web pages and data are built from server-side Perl modules or libraries, and simple HTML/JavaScript on the client side.

Ubiquitous access to the Grid: MyProxy

As stated above, Grid Portals are used as a ubiquitous way to access Grid Services. To perform “grid operations”, the user has to delegate the Grid Portal to use his/her credentials. In other words, to access the Grid, the portal must use a proxy certificate signed by the user. This can be accomplished with the MyProxy technology [Novo01], currently exploited by major projects focused on Grid Portals.

MyProxy provides a repository of online credentials, designed to overcome the incompatibility of Web and Grid security infrastructures, thus enabling Grid Portals to use Grid protected resources in a secure and scalable manner, also ensuring the Single-Sign-On property.

The use of MyProxy can be summarized as follows:

1. The user, logged on a Grid host, creates a proxy certificate, sends it to a secure MyProxy repository, and associates the proxy with a password. The user also specifies the amount of time that the proxy will be valid.

2. Later, the user, this time logged on another generic host, gives the portal server the password, thus allowing it to contact the proxy server and load the proxy certificate.
3. The portal server will hold the proxy in the user's session state for the amount of time specified at step 1. Within this period, the portal will be able to access the grid services on behalf of the user.

4.8. Application Domain Description

A PSE toolkit should be used for different application domains. Therefore it is necessary to distinguish what components are common and what are specific of an application domain.

Domain specific components help domain scientist or engineer in building the abstract workflow of their application (see Section 3.2 for more details). The choice of a particular domain hides the view of all components and shows only the components concerning that specified sphere of influence. For example, a geologist could have access to specialized software as GIS tools, to hardware resources as specialized printers and to limited access resources as computer maps. Note that these components can be in two different types: specialized for a particular type of task or general components for effective use concerning that a specified domain (can hide some general characteristic or acquire others).

Furthermore workflow templates are available for more common operations and they can also be added by the domain engineer, when needed. Obviously the latter operation can be executed in an abstract way without knowledge of the low level component design and of the dynamic characteristics of the grid environment. Graphical user interfaces can facilitate this task.

Crucial issues concerning specific components are:

- An ontology of goals, actors, resources specific of an application domain (see Section 4.3);
- The interface between the application domain and the PSE toolkit.

The last point is probably the more complex. There are two points of view:

1. from the point of view of the application domain, what are the services asked to the PSE toolkit (i.e. two fundamental services could be a graphical user interface or a user friendly domain-specific language.)
2. from the point of view of the PSE toolkit, what are the characteristics of the application domain that allow the effective use of the PSE toolkit? Can the toolkit be used only for scientific domains, i.e. for domain that have formal and well known characteristics?

Domain and task ontologies

A domain ontology describes a specification of basic categories as these are instantiated through the concrete concepts and relations arising within a specific application domain. Due to this, ways must be found to take into consideration different experts views on the domain concepts and relations, as well different goals. These concepts are used for all tasks that occur within that domain. Task Ontologies describe the reasoning concepts and their relationships occurring within a certain domain and for a specific task. Task ontologies link the reasoning process to domain factual knowledge [Grub93] Using domain ontologies together with task ontologies it

is necessary for the identification of mapping rules between the domain (including primitives) and its related tasks.

Domain specific languages and Graphical User Interfaces

The interface among the application domain and the PSE could be performed by means of a graphical user interface or a user friendly domain-specific language.

A possible scenario could be the following. A domain scientist could describe the domain-specific knowledge using the DSL or the GUI; both of them should be based on ontologies as those described in the previous paragraph.

Domain-specific languages (DSLs) [Deur00] are oriented towards a particular application domain. By making the notations and concepts of an application domain available to the programmer, DSLs allow application programs to be expressed more concisely and directly than in general purpose languages. Details of the underlying implementation platform are captured by knowledge built into the DSL compiler and associated ontologies.

The use of a DSL has many advantages. Since the language is close to the notation commonly used by domain-experts, it has a very high level of expression in the domain for which it was designed. Furthermore, the DSL compiler does not have to handle arbitrary programs, but can generate specialized code that is optimized toward domain-specific situations. Adopting a DSL approach for a PSE framework involves both risks and opportunities. The *benefits* of DSLs include that solutions may be expressed in the idiom and at the level of abstraction of the problem domain and consequently, domain experts themselves can understand, validate, modify, and often even develop DSL programs.

On the contrary, the use of a DSL involves the costs of designing, implementing and maintaining a DSL and the difficulty of balancing between domain-specificity and general-purpose programming language constructs.

Scenario of an application domain: Clustering of Human Proteins

A biologist can exploit the PSE application domain specific characteristics to develop its application. The description of the application domain can guide the user in the definition and composition of the application. The scientist can select its application domain (bioinformatics) by means of the GUI and he can see the data and the hardware and software resources related to its domain. In particular considering the bioinformatics domain it is possible to identify the following resources:

- biological data sources such as protein databases (e.g., SwissProt, PDB)
- software components such as bioinformatics tools and software for retrieving and managing biological data (Entrez, SRS)
- bioinformatics process/task (sequence alignment, protein structure prediction, similarity search)
- hardware components specific for bioinformatics applications such as the mass spectrometers

Moreover the biologist can use all the other resources related to PSE and the GUI will visualise suggestions if any general resources can be useful for the domain, i.e. a

storage device adapt to contain the database or a high performance computer for the processing of database.

A bioinformatics domain ontology will guide the user in the application definition describing the semantics of data sources, software components available and bioinformatics tasks and how they can be composed together.

For example a common bioinformatics applications is the clustering of proteins. Protein function predictions uses database searches to find proteins similar to a new protein, thus inferring the protein function. This method is generalized by protein clustering or classification, where databases of proteins are organized into groups or families in a manner that attempt to capture protein similarity. In Figure 8 is shown the workflow describing the clustering application. The domain specific PSE components used in the application are:

- EMBOSS suite (seqret)
- BLAST tool (blastall)
- TribleMCL tool
- SwissProt database

As it should be noted in Figure 8, first all the human proteins sequences are extracted from the Swiss-Prot database using the `seqret` program of the `EMBOSS` suite (`seqret` is a program for extracting sequences from databases). `TribeMCL` uses an all against all `BLAST` comparison (evaluates the similarity among proteins) as input to the clustering process, thus once the protein sequences have been extracted from the database a `BLAST` computation has to be performed. To speed up the similarity search activity the `seqret` output has been partitioned in three smaller files; in this way three `BLAST` computations can be run in parallel. The obtained raw `NCBI BLAST` outputs are converted in the format required to create the Markov Matrix used in the clustering phase and the parsing will be executed using tribe-parse program. Finally the clustering operation will be executed using `mcl` and the results will be displayed.

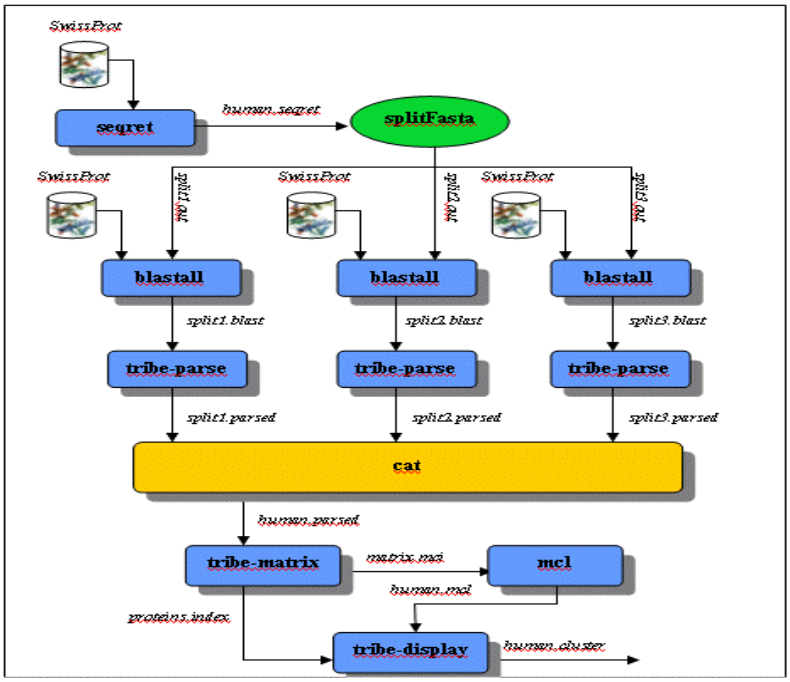

Figure 8. Workflow of a bioinformatics application: clustering of human proteins sequences

References

[Agui03] V. Aguilera, S. Cluet, T. Milo, D. Vodislav, P. Veltri. Views in a Large Scale XML Database. VLDB Journal, 11(3) 2002.

[Akar98] E. Akarsu, G. Fox, W. Furmanski, T. Haupt. Webflow - high-level programming environment and visual authoring toolkit for high performance distributed computing. Proceedings of Supercomputing'98, Florida, November 1998.

[Ant03] The Jakarta project, Ant – a Java-based Build Tool. *http://jakarta.apache.org/ant/*

[Arms99] 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. Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computation, 1999.

[Bala00] Z. Balaton, P. Kacsuk, N. Podhorszki, F. Vajda. Comparison of Representative Grid Monitoring Tools. Report of the Laboratory of Parallel and Distributed Systems, LPDS-2/2000.

[Bech01] S. Bechhofer, I. Horrocks, C. Goble, R. Stevens. OilEd: a Reason-able Ontology Editor for the Semantic Web. Proceedings of KI2001, Joint German/Austrian conference on Artificial Intelligence, September 19-21, Vienna. Springer-Verlag LNAI Vol. 2174, pp 396—408, 2001.

[Benj98] V. R. Benjamins, E. Plaza, E. Motta, D. Fensel, R. Studer, R. Wielinga, G. Screiber, Z. Zdrahal, S. Decker. An intelligent Brokering Service for Knowledge Component Reuse on the World-Wide Web. KAW'1998, 1998.

[Beyn00] M. D. Beynon, R. Ferreira, T. Kurc, A. Sussman, and J. Saltz. DataCutter: Middleware for filtering very large scientific datasets on archival storage systems. MASS2000, pages 119–133. National Aeronautics and Space Administration, Mar. 2000. NASA/CP 2000-209888.

[Bon99] E. Bonabeau, M. Dorigo, G. Theraulaz. Swarm Intelligence: From Natural to Artificial Systems, Oxford University Press, 1999.

[Cact03] Cactus Webmeister. The cactus code website. *http://www.CactusCode.org*, 2003.

[Cann03] M. Cannataro, A. Congiusta, C. Mastroianni, A. Pugliese, D. Talia, P. Trunfio. Grid-based data mining and knowledge discovery. To be published in: N. Zhong, J. Liu (Eds.), Intelligent technologies for information analysis, IOS Press, 2003.

[Cann03a] M. Cannataro, C. Comito. A Data Mining Ontology for Grid Programming. 1st Int. Woprkshop on Semantics in Peer-to Peer and Grid Computing (SemPGrid2003), pp. 113-134, 2003.

[Cann03b] M. Cannataro, D. Talia. KNOWLEDGE GRID: An Architecture for Distributed Knowledge Discovery. Communications of the ACM, January 2003.

[COG03] The Globus Project. Java Commodity Grid Kit. *http://www.globus.org/cog/java*.

[Corb03] Object Management Group. CORBA 3. *http://www.omg.org/technology/corba/corba3releaseinfo.htm*, September 10, 2002.

[Corp00] B.A. Bremdal, F. Johansen. CORPORUM Technology and Applications, CognIT a.s, June 2000. *http://www.ontoknowledge.org/downl/CorporumTechApp.pdf*

[Cunh03] J. C. Cunha. Future Trends in Distributed Applications and PSEs. Talk held at the Euresco Conference on Advanced Environments and Tools for High Performance Computing, Albufeira (Portugal), 2003.

[Czaj01] K. Czajkowski, S. Fitzgerald, I. Foster, C. Kesselman. Grid Information Services for Distributed Resource Sharing. Proc. of the 10 th IEEE Symp. on High Performance Distributed Computing, 2001.

[DAGM03] The Condor project. DAGMan Meta−Scheduler.

http://www.cs.wisc.edu/condor/manual/v6.2/2_10Inter_job_Dependencies.html

[DAML03] W3C. DAML+OIL. *http://www.w3.org/TR/daml+oil-reference*

[DAML03a] DAML API. *http://grcinet/grci.com/*

[Deel03] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, K. Blackburn, A. Lazzarini, A. Arbree, R. Cavanaugh, S. Koranda. Mapping Abstract Complex Workflows onto Grid Environments. Journal of Grid Computing, Vol.1, no. 1, 2003, pp. 25-39

[Deur00] A. van Deursen, P. Klint, J. Visser. Domain-Specific Languages: An Annotated Bibliography. SIGPLAN Notices 35(6): 26-36 (2000)

[Digr96] S. Di Gregorio, R. Rongo, W. Spataro, G. Spezzano, D. Talia. CAMEL: A Parallel Cellular Tool for Interactive Simulation and Modeling. IEEE Computational Science & Engineering, Vol. 3, No. 3, Fall 1996.

[Fost98] I. Foster, N. T. Karonis. A grid-enabled MPI: Message passing in heterogeneous distributed computing systems. In Supercomputing, November 1998. www.supercomp.org/sc98.

[Free99] E. Freeman, S. Hupfer, and K. Arnold. JavaSpaces: Principles, Patterns, and Practice. Addison-Wesley, 1999.

[Gall94] E. Gallopoulos, E. N. Houstis, J. Rice. Computer as Thinker/Doer: Problem-Solving Environments for Computational Science. IEEE Computational Science and Engineering, vol.1, n. 2, 1994.

[Gann03], D. Gannon. Grid Summer School 2003 slides. *http://www.extreme.indiana.edu/~gannon/Grid-school.ppt*

[GGF] The Global Grid Forum. *http://www.ggf.org*

[GGSI03] Globus Grid Security Infrastructure. *http://www.globus.org/security/overview.html*

[Govi03] M. Govindaraju, S. Krishnan, K. Chiu, A. Slominski, D. Gannon, R. Bramley. Merging the CCA Component Model with the OGSI Framework. Proceedings of CCGrid2003, 3rd International Symposium on Cluster Computing and the Grid. May 2003.

[Gpdk03] National Laboratory for Applied Network Research. Grid Portal Development Toolkit (GPDK). *http://doesciencegrid.org/projects/GPDK/*

[gPort03] The Grid Portal Toolkit. *http://gridport.npaci.edu*.

[Grub93] T. R. Gruber. A translation approach to portable ontologies. Knowledge Acquisition, 5(2):199-220, 1993. Available at *http://ksl-web.stanford.edu/KSL_Abstracts/KSL-92-71.html*.

[GSI-WG03] GGF Grid Security Infrastructure Working Group (GSI-WG). *https://forge.gridforum.org/projects/gsi-wg*

[SAAA-RG03] GGF Site Authentication, Authorization, and Accounting Requirements Research Group (SAAA-RG). *https://forge.gridforum.org/projects/saaa-rg*

[GSO-WG03] GGF Grid Scheduling Ontology Working Group. *http://www.gso-wg.org/*

[Iamn02] A. Iamnitchi, I. Foster, D. Nurmi. A peer-to-peer approach to resource discovery in grid environments. High Performance Distributed Computing, Edinbourgh, UK, July 2002.

[Jb03] JavaBeans. *http://java.sun.com/products/javabeans/* , September 19, 2003.

[Jena03] Jena – A Semantic Web Framework for Java. *http://jena.sourceforge.net/ index.html*

[Jets03] The Apache Jakarta project. *http://jakarta.apache.org/jetspeed/*

[jxta03] JXTA. *http://www.jxta.org/*. September 19, 2003.

[Kris01] S. Krishnan, R. Bramley, D. Gannon, M. Govindaraju, R. Indurkar, A. Slominski, B. Temko, R. Alkire, T. Drews, E. Webb, and J. *Alameda: The XCAT Science Portal*, Proceedings of SC2001, 2001.

[Lasz00] G. Von Laszewski, I. Foster, J. Gawor. CoG Kits: A bridge between commodity distributed computing and high-performancve Grids. Proceedings of the ACM Java Grande Conference, June 2000.

[Lasz00a] G. Von Laszewski. A Loosely Coupled Metacomputer: Cooperating Job Submissions across Multiple Supercomputing Sites. *Concurrency, Experience, and Practice*, 2000.

[Lasz01] G. Von Laszewski, I. Foster, J. Gawor, P. Lane, N. Rehn, M. Russell. Designing Grid-based Problem Solving Environments and Portals. 34th Annual Hawaii International Conference on System Sciences (HICSS-34) - January 2001, Hawaii.

[Lee01] C. Lee, S. Matsuoka, D. Talia, A. Sussman, N. Karonis, G. Allen, J. Saltz. A Grid Programming Primer. Global Grid Forum 2, Washington D.C., July 2001.

[Lenz02] M. Lenzerini et al. Data Integration. Proceedings of the ACM Symposium on Principles of Database Systems, 2002.

[Mala03] S. Malaika, A. Eisenberg, J. Melton. Standards for Databases on the Grid. ACM SIGMOD Record, Vol. 32, No. 3, September 2003.

[Mast03] C. Mastroianni, D. Talia, P. Trunfio. Managing Heterogeneous Resources in Data Mining Applications on Grids Using XML-Based Metadata. Heterogeneous Computing Workshop (HCW 2003), Nice, France, April 2003.

[Math03] Mathworks. MATLAB: the language of technical computing. *http://www.mathworks.com/*

[Mds03] The Globus Project. The Monitoring and Discovery Service (MDS-2). *http://www.globus.org/mds/mds2*

[Milo99] D. Milojicic. Mobile Agent Applications. IEEE Concurrency, 1999.

[Mol03] Molecular Science Software Suite. *http://www.emsl.pnl.gov:2080/mscf/ about/descr_ms3.html*

[Nabr03] J. Nabrzyski, J. Schopf, J. Weglarz (Eds.). Grid Resource Management. Kluwer, 2003.

[Nets03] The NetSolve project. *http://www.cs.utk.edu/netsolve*

[Ninf03] The Ninf Project. *http://ninf.apgrid.org*

[Novo01] J. Novotny, S. Tuecke, and V. Welch. An Online Credential Repository for the Grid: MyProxy. Proceedings of the Tenth International Symposium on High Performance Distributed Computing (HPDC-10), IEEE Press, August 2001.

[Ogsa03] The Globus Project. The Globus Toolkit 3, Towards the Open Grid Services Architecture. *http://www.globus.org/ogsa/*

[OGSA-DAI] Global Grid Forum, Open Grid Services Architecture - Data Access and Integration. *http://www.ogsa-dai.org.uk*

[Ope97] OpenMP Consortium. OpenMP C and C++ Application Program Interface, Version 1.0, 1997.

[OWL03] W3C - OWL Web Ontology Language, *http://www.w3.org/TR/owl-ref/*

[Paol02] M. Paolucci, T. Kawamura, T.R. Payne, K. Sycara. Semantic Matching of Web Services Capabilities. First International Semantic Web Conference (ISWC), June, 2002.

[Pell03] Purdue University. The PELLPACK PSE. *http://www.cs.purdue.edu/research/cse/pellpack/pellpack.html*

[Psew03] The PSEWare project. *www-extreme.indiana.edu/pseware*

[RDF] World Wide Web Consortium, Resource Description Framework (RDF), *http://www.w3.org/RDF*

[Rusi94] M. Rusinkiewicz, A. Sheth. Specification and Execution of Transaction Workflows. In: Modern Database Systems: the Object Model, Interoperability, and beyond, W. Kim (ed.), Addison-Wesley, 1994.

[Schu02] K. Schuchardt, B. Didier, G. Black. Ecce - a problem-solving environment's evolution toward Grid services and a Web architecture. Concurrency and computation: practice and experience, 14:1221–1239, 2002.

[Scir03] Scientific Computing and Imaging Institute. SCIRun PSE. *http://software.sci.utah.edu/scirun.html*

[SD-WG03] Global Grid Forum. Scheduling Dictionary Working Group. *http://www.fz-juelich.de/zam/RD/coop/ggf/sd-wg.html*

[SOAP03]. W3C. XML Protocol Working Group. *http://www.w3.org/2000/xp/Group/*

[Taha99] Y. Taha, A. Helal, K. Ahmed, J. Hammer. Managing Multi-Task Systems Using Workflow. International Journal of Computers and Applications (IJCA), 21:3, pages 69-78, September 1999.

[Tali02] D. Talia. The Open Grid Services Architecture: Where the Grid Meets the Web. IEEE Internet Computing 6(6): 67-71 (2002)

[Tang03] H. Tangmunarunkit, S. Decker, C. Kesselman. Ontology-Resource Matching in the Grid – The Grid meets the Semantic Web. 1st Int. Workshop on Semantics in Peer-toPeer and Grid Computing (SemPGrid2003), 85-101.

[Text03] The Text-To-Onto System. *http://ontoserver.aifb.uni-karlsruhe.de/texttoonto/*

[Thom01] M. Thomas, S. Mock, J. Boisseau, M. Dahan, K. Mueller, D. Sutton. The GridPort Toolkit Architecture for Building Grid Portals. Proceedings of the 10th IEEE Intl. Symp. on High Perf. Dist. Comp. Aug 2001

[Topc97] H. Topcuoglu, S. Hariri, W. Furmanski, J. Valente, I. Ra, D. Kim, Y. Kim, X. Bing, B. Ye. The software architecture of a virtual distributed computing environment. Proceedings of the High-Performance Distributed Computing Conference, 1997.

[UBOT03] UML Based Ontology Toolset, *http://ubot.lockheedmartin.com/ubot/intro/index.html*

[UDDI03] The OASIS Standards Consortium. UDDI: Universal Description, Discover and Integration of Business for the Web. *http://www.uddi.org*

[Vett97] J. Vetter, K. Schwan. High performance computational steering of physical simulations. In Proceedings of the 11th International Parallel Processing Symposium, IPPS 97, pages 128--132, 1997.

[W3C] The World Wide Web Consortium. *http://www.w3.org*

[Walk00] D. Walker, O. F. Rana, M. Li, M. S. Shields, and Y. Huang, The Software Architecture of a Distributed Problem-Solving Environment. Concurrency: Practice and Experience, Vol. 12,No. 15, pages 1455-1480, December 2000.

[Welc03] V. Welch, F. Siebenlist, I. Foster, J. Bresnahan, K. Czajkowski, J. Gawor, C. Kesselman, S. Meder, L. Pearlman, S. Tuecke. Security for Grid Services. T Twelfth International Symposium on High Performance Distributed Computing (HPDC-12), IEEE Press, 2003.

[WfMC03] The Workflow Management Coalition, *http://www.wfmc.org*.

[Wols99] R. Wolski, N. Spring, J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. Journal of Future Generation Computing Systems,Volume 15, Numbers 5-6, pp. 757-768, October, 1999.

[Wroe03] C. Wroe, R. Stevens, C. Goble, A. Roberts, M. Greenwood. A suite of DAML+OIL Ontologies to describe Bioinformatics Web Services and Data. Journal of Cooperative Information Systems special issue on Bioinformatics, March 2003.

[WSDL03] W3C. Web Services Description Language. *http://www.w3.org/TR/wsdl20/*

[WSFL03] IBM. Web Services Flow Language (WSFL). Version 1.0

[WSIL03] The OASIS Standards Consortium, WSIL: Web Services Inspection Language, *http://xml.coverpages.org/wsil.html*

[Xcat02] M. Govindaraju, S. Krishnan, K. Chiu, A. Slominski, D. Gannon, R. Bramley. XCAT 2.0: A Component-Based Programming Model for Grid Web Services. Technical Report-TR562, Department of Computer Science, Indiana University. Jun 2002.

[XML] W3C. Extensible Markup Language (XML). *http://www.w3.org/XML*

[Yang03] B. Yang, H. Garcia-Molina. Designing a Super-peer Network. IEEE International Conference on Data Engineering, 2003.