A Visual Programming Environment for Developing Complex Grid Applications

Antonio Congiusta[1,2], Domenico Talia[1,2], and Paolo Trunfio[2]

[1] ICAR-CNR, Institute of the Italian National Research Council,
Via P. Bucci, 41c, 87036 Rende, Italy
congiusta@icar.cnr.it

[2] DEIS - University of Calabria,
Via P. Bucci, 41c, 87036 Rende, Italy
{talia, trunfio}@deis.unical.it

1 Introduction

Grids are receiving even more attention by a significant number of scientific, industrial, and economical bodies, thanks to their capability to enable collaborations, even cross-organizational, based on large scale resource sharing and performance orientation.

In the latest years Grids researchers and professionals have been concerned with the development of a series of experiments and demonstrations aimed at showing basic Grid features and potentials. Large scale Grids were deployed to solve computational or data intensive problems as well as to perform complex simulations. Presently, Grids are widely recognized as the next generation computing architecture, the natural evolution of the Web towards the delivery of computing power, information, and knowledge.

Now, Grid community efforts are focused to make that technology robust, reliable, and available to those interested in adopting it. The recent involvement of companies like IBM, Sun, and Microsoft is a clear symptom of the relevance the matter is going to assume in the near future. Belong this trend, providing high-level environments and advanced instruments able to support end users and developers, is of main importance to explore many of the Grid related benefits not yet fully exploited.

In the Grid era people have not to worry about the acquisition of powerful computers or expensive instruments, but rather the key aspect is the capability of effectively exploit shared resources through high level environments providing the needed abstractions and facilities. Indeed, today this scenario is still not realistic due to research and technological challenges that must be faced, but it should lead scientists and professionals to provide more abstract techniques and tools for supporting Grid computing.

Most problems addressed by Grids are not simply solved through the execution of a specific ad hoc "program", but often require several software modules, most likely interacting each others, to run separately and/or concurrently over a given set of inputs. During a certain period of time Grid Portals have been the most advanced in-

strument for the solution of this kind of problems. But they represent still a limited programming approach and are often tailored to specific application domains.

Till today not much work has been done to build high-level design facilities for complex Grid applications in which many programs and data sets are involved. This class of applications are quite common in several domains, such as knowledge management, computational science, and e-business; in addition they share common traits with software component based applications.

Software component technology is now a standard part of many software design practices. Microsoft COM and much of .NET [1] are based on component concepts, as well as Enterprise Java Beans [2], that is another important technology for building large scale e-commerce applications. A software component model is a system for assembling applications from smaller units called components. The system defines a set of rules that specify the precise execution environment provided to each component, the rules of behavior, and special design features components may have. A component is then nothing more than an object (or collection of objects) that obey the rules of the component architecture. A component framework is the software environment that provides the mechanisms to instantiate components, compose and use them to build applications. The software component model can be effectively used in Grid applications integrating legacy code and new software modules.

In this chapter we present a high-level Grid programming environment that shares some common features with the software component paradigm. The system we discuss here is VEGA - *Visual Environment for Grid Applications*. VEGA provides a unified environment comprising services and functionalities ranging from information and discovery services to visual design and execution facilities. VEGA was designed and implemented to support users in the design of data-intensive Grid applications as part of the Knowledge Grid [6], a software infrastructure for developing knowledge discovery applications. However its high-level features make it useful in the development of a large class of Grid applications.

The remainder of the chapter is organized as follows. Section 2 presents the design aspects and the main features of VEGA. Section 3 introduces the visual language used to design an application in VEGA. Section 4 illustrates the architecture of the environment and Section 5 goes more deeply into some implementation aspects. Several enhancements and additional features under development are presented in Section 6, where "open issues" are discussed. Section 7 presents two case studies and Section 8 discusses some of the major related projects. Finally, Section 9 concludes the chapter.

2 Main Features and Requirements

The main goal of VEGA is to offer a set of visual functionalities that give the users the possibility to design complex software, such as complex solving environments and knowledge discovery applications, starting from a view of the present Grid status (i.e., available nodes and resources), and composing the different steps inside a structured environment, without having to write submission scripts or resource description files.

The high-level features offered by VEGA are intended to provide the user with easy access to Grid facilities with a high level of abstraction, in order to leave her/him free to concentrate on the application design process. To fulfill this aim VEGA builds a visual environment based on the component framework concept, by using and enhancing basic services offered by the Knowledge Grid and the underlying Grid middleware.

To date, a Grid user willing to perform a Grid application must know and handle a number of detailed information about involved resources (computing nodes, software, data, etc.), such as their names and locations, software invocation parameters, and other details. Thus, in the absence of high level tools the planning and submission of an application could result in a long and annoying work, exposed even to failures due to user mistakes in writing allocation scripts with a given syntax, wrong memory about resources details, etc.

As a first feature, VEGA overcomes these difficulties by interacting with the *Knowledge Directory Service* (KDS) of the Knowledge Grid to know available nodes in a Grid and retrieve additional information (metadata) about their published resources. Published resources are those made available for utilization by a Grid node owner by means of the insertion of specific entries in the G*rid Information Service*. Therefore, when a Grid user starts to design its application, she/he needs first of all to obtain from KDS *metadata* about available nodes and resources. After this step, she/he can select and use all found resources during the application design process (as described in the following). This first feature aims at making available useful information about Grid resources, showing the user their basic characteristics and permitting her/him to design an application.

The application design facility allows the user to build typical Grid applications in an easy, guided, and controlled style, having always a global view of the Grid status and the overall building application. VEGA offers the users a way to look at Grid resources as a collection of typed resources and a set of defined "relationships" between them. This can be identified as the core functionality of VEGA. To support structured applications, composed of multiple sequential stages, VEGA makes available the *workspace* concept, and the *virtual resource* abstraction. Thanks to these entities it is possible to compose applications working on data processed in previous phases even if the execution has not been performed yet (useful in many knowledge discovery applications).

Once the application design has been completed, resulting job requests are to be submitted to the proper Grid Resource Allocation Manager (GRAM). VEGA includes in its environment the execution service, which gives the designers the possibility to execute an application and to view its output.

Another important feature is the monitoring of the jobs execution, needed to allow the user to get information about the different jobs running on different machines as originated by the application execution.

A Grid-based application is often more complex with respect to a similar one based on classical computing systems. Issues like distribution of software, data, and computers themselves have to be addressed. The availability of computing nodes able to host a given computation is related to strict constraints about performance and

platform requirements, as well as specific policies about access to resources, as defined by the related *virtual organization* [4].

In addition, a Grid application seeks out to take advantage of all benefits coming from the distributed and potentially parallel environment. For that reason some other problems may arise. In complex applications, large simulations and expecially knowledge discovery applications, collaborations and interchanges between several concurrent or sequential stages of the overall computation are very common and able to boost up their capabilities and performance.

To conclude about the design properties and choices that guided us in the design of VEGA, here we list a set of properties that we identified to draw up the general requirements for a high level Grid programming environment:

- the environment has to provide a set of useful abstractions about Grid resources and basic and enhanced actions supported by them;
- an abstract model for defining relationships among Grid resources and constructs for composing an applications must be provided;
- an advanced use of information about resources availability and status is fundamental to allow for a dynamic adaptation of applications to the changing conditions of the Grid;
- the system architecture must be as much as possible independent from low level mechanisms used to gather information about resources, allocate jobs, etc.; but at the same time,
- more specialized and high level services must be compatible with low level basic Grid services, so as to facilitate the implementation and take advantage of the underlying middleware;
- when the planning of an application implies choices that may affect its performance, the system needs to implement the right combination between a user driven and a system driven policy in taking decisions.

3 A Visual Language to Specify Applications

A structured way to model and express the variety of constraints and implications of complex Grid applications is needed. We believe that a way to address discontinuities (in systems characteristics as well as domain policies) and problems related to the scattered nature of resources is to look at a Grid application as a collection of resources and a set of well defined relationships among them.

VEGA, rather than devise a set of customized syntactical rules, makes available a *visual language* to express “relationships” among “resources”, and to describe with a graphical representation the overall computation.

VEGA provides developers with a set of graphical objects representing different kinds of resources they can select and use to compose an application. In particular, there are three types of graphical objects:

- *hosts*,
- *software*, and
- *data*.

Each of these objects represents a physical resource in the Grid. The user can insert several instances of the same resource into a *workspace* of the current project if needed. A *workspace* is thus a working area of the VEGA environment, in which objects representing resources are hosted to form a particular stage of the application. When a resource is inserted in a workspace, a label containing the name of the related physical resource is added to the corresponding graphical object.

Several relationships, indicating interactions between resources, can be defined. Relationships are represented in VEGA as graphical links between the resources which they refer to. Through relationships it is possible to specify one or more desired actions on resources included in a workspace. In other words, it is possible to describe one or more jobs (see Fig. 1).

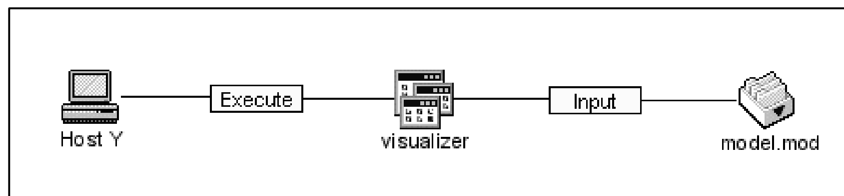

Fig. 1. Objects and links in VEGA

A common definition of "Grid job" states that it is *the execution of a given software on a specific Grid node*, with its input and output parameters/files specified as well. Relationships available to describe jobs are:

- *execution* of a given software on a given host,
- *file transfer*, of a certain software or dataset on a specified host
- *input*, (a given dataset as input for a software) and
- *output* (a given file as collector of a software output).

The file transfer relationship is a special kind of job, since it can be viewed as the execution of a file transfer program which parameters are the file to be moved, the destination host, etc. Thus, in VEGA *a job is a software execution or a file transfer*.

While introducing relationships between resources, some rules must be followed, in order to make the composition have sense. For instance, an execution relationship (link) cannot be inserted between a data object and a host object. Moreover, no link can be inserted between objects of the same type.

The set of admissible links is listed in Table 1, with enclosed the meaning each one takes in the specific context. Objects refer to specific resources offered by Grid nodes included in the deploying project; they can be linked with resources owned by the same node as well as with resources of a different node. In the latter case a *staging* operation is implicitly defined. If, for instance, a software component *SW* owned by the host *H1* is linked with an execute link with a host *H2*, then the executable associ-

ated to *SW* will be first transferred to *H2*, and will be deleted from there after the execution.

Table 1. Admissible links between resources

resource1	*resource2*	*link*	*Meaning*
Data	Software	Input	**indicates the software input**
Data	Software	Output	**indicates the software output**
Data	Host	File Transfer	**data transfer to the host**
Software	Host	File Transfer	**software transfer to the host**
Software	Host	Execute	**software execution on the host**

A computation is generally composed by a number of different sequential or parallel steps. In VEGA it is possible to distinguish sequential and parallel execution of jobs through the *workspace* concept. All jobs that can be executed concurrently are to be placed inside the same workspace, whereas different workspaces can be used to specify a priority relationship. It is worth to note that when different sets of jobs are placed into different workspaces often they share common data, on which they make different computations.

As an example, let us consider a typical execution of the data mining tool `DM-tool` on the specific Grid node `Host_X`, taking as input the `pre-processed.dat` file, and producing as result the file `classes.dat`. This job (say simple submission) is described in the VEGA *visual language* by linking the objects representing the resources as showed in Fig. 2-A. A multiple submission, that is the parallel execution of the same software component on more than a Grid node, is quite similar to a simple execution, the only exception is that the same software object is linked with more hosts (see Fig. 2-B). Similarly, different submissions, i.e. two different software to be executed on different hosts, may share the same dataset, like in Fig. 2-C.

The design of an application is obtained through the composition of a graphical model representing it. This is accomplished by using some graphical objects on which several actions (like insertion in a workspace, linking with other objects, specification of attributes and properties, etc.) can be invoked, given that they assure consistency and logic sense to the computation. All this happens giving always freedom to the user in choosing how to build its application and the ordering of the actions applied to graphical objects.

A fundamental characteristic is the possibility to model the designing application through a visual language offering a set of abstractions as flexible as possible and able to describe a significant part of the typical Grid scenarios and applications. In this approach lies the novelty and the power of an environment like VEGA, because till now the community was lacking of such a reference model and structured operational way on which to model a Grid application.

Moreover, it should be noted that although a set of predefined *links* are employed, there are several specific attributes and parameters that contribute to increase the expressiveness and the flexibility of the visual language.

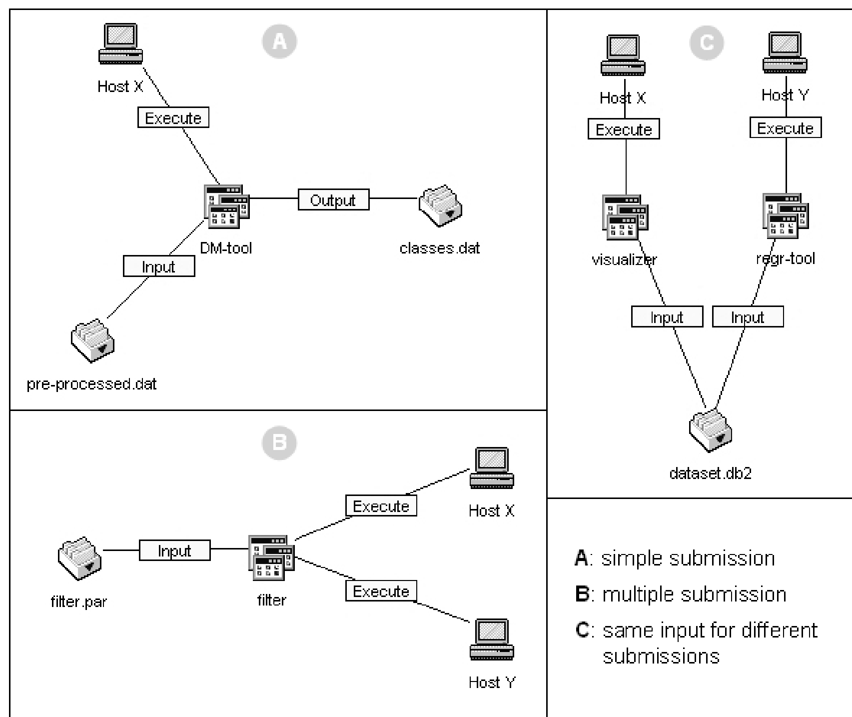

Fig. 2. Some jobs in the VEGA visual language

4 Architecture

This section outlines how the VEGA environment works on top of the Knowledge Grid and the underlying Grid middleware, basic relationships and interchanges among these entities are also explained.

The *Knowledge Grid* is a software infrastructure for distributed data mining and extraction of knowledge in Grid environments. The Knowledge Grid accomplishes its objectives through the implementation of a set of basic services and high level tools designed to support geographically distributed high-performance knowledge discovery applications [6].

The set of services and functionalities offered by VEGA is composed basically by two categories: design facilities and execution handling. The first ones are concerned with functions for designing and planning a Grid application, whereas the others make possible to execute the application. Fig. 3 shows hierarchies and some basic interac-

tions between them and the Knowledge Grid and Grid services. In particular, the design facilities make use of the *knowledge directory service*, implemented by the Knowledge Grid, to discover resources and their properties, whereas basic Grid services are used during the authentication and the execution phases.

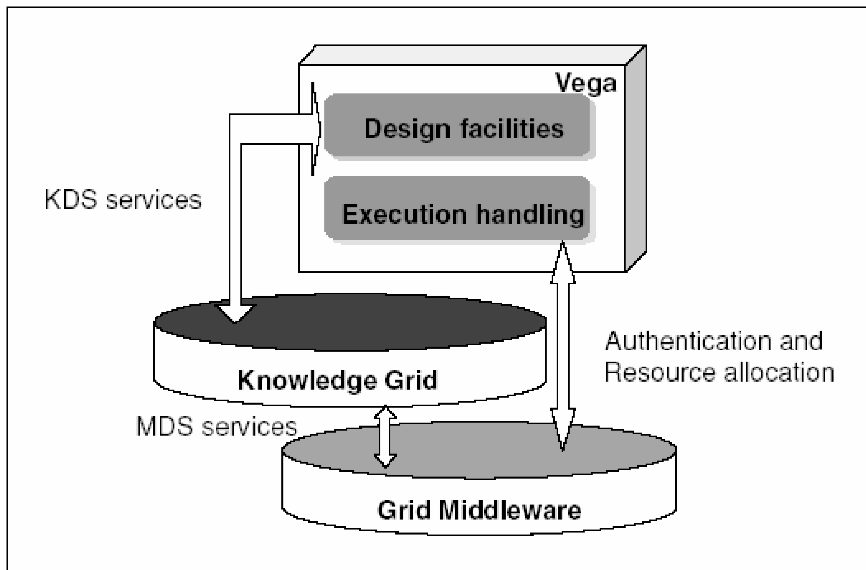

Fig. 3. VEGA overall architecture

There are at least four steps a user must follow to execute one or more jobs employing Grid resources:

- *definition* of involved resources and specifications of the relationships among them;
- *checking* of the planned actions consistency;
- *generation of the job set* to be submitted to one or more Grid resource allocation managers;
- *execution* of the jobs and *monitoring* of their life cycle.

The job submission procedure in VEGA can be divided in the previous four steps. Fig. 4 shows the VEGA software modules implementing them together with the needed data exchanges.

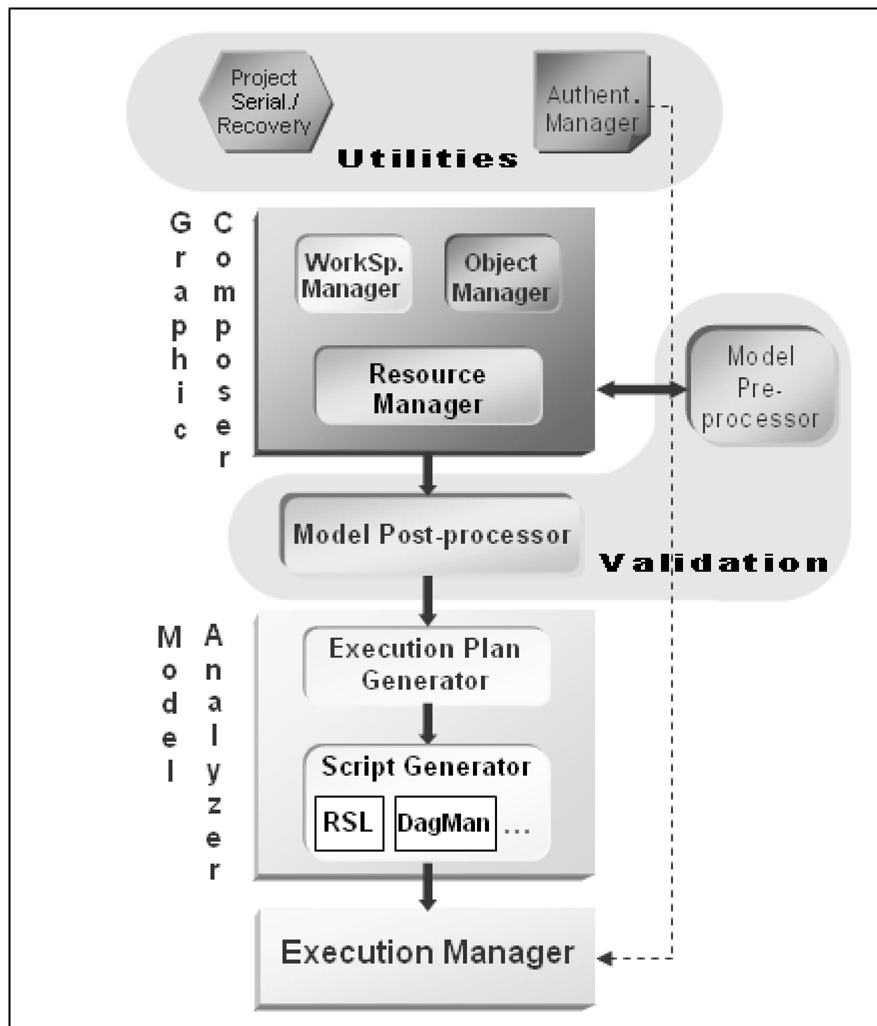

Fig. 4. VEGA software modules

The visual composition phase is useful to the user during the application design. It is accomplished by the **Graphic Composer** software module and its sub-modules. To design her/his application a user may compose graphical objects representing resources (datasets, software components, computing nodes). These objects are composed through visual facilities, aimed to specify existing relationships among them, to

form a graphical representation of each job of the entire computation, accordingly with the rules discussed in the previous section.

Namely, this task is accomplished by the following sub-modules: Workspace *Manager*, *Object Manager* and *Resource Manager*.

The **Resource Manager** (RM) is concerned with the browsing of a local cache called *Task Metadata Repository* (TMR), where metadata about retrieved resources are stored, in order to allow the selection of resources to include in the designing computation. The Resource Manager is divided into two sections, directly showed to the user: the first one contains the list of the chosen computing nodes (hosts); the second one contains the resources belonging to the currently selected host, divided into two categories, data and software, on the basis of the content of associated metadata (see Fig. 5). Such a structure gives an overall view of the available hosts and related resources, permitting, at the same time, to include them inside a workspace of the current project.

Additional information about these resources are retrieved through the *Knowledge Directory Service*, a system able to include customized metadata into the *Grid Information Service*, and to delivery them to each user of the Grid when requested. Metadata used in the Knowledge Grid are constituted by XML encoded information describing software and data resources. The analysis of the Knowledge Grid information system functioning is out the purposes of this chapter (details can be found in [6] and [7]). For what concerned with the operations of the RM, it is enough to know that XML documents specify some resources attributes among which: owner host, file name and related path.

The TMR is located on the file system of each node of the Knowledge Grid running VEGA and is organized as a set of directories, one for each host. Each of these directories contains XML documents about resources published by related hosts by using KDS services. A host can publish basically two types of resources, software and data.

The **Workspace Manager** (WM) is given the task of creating and managing workspaces. It maintains an internal representation of the entire computation designed by the user (internal model) and a priority relationship between workspaces, as they constitute a whole computation, in which the workspace sequence represents the planned sequence of tasks to be executed. In addition, it may be possible that some jobs in a given workspace need to operate on resources generated in a previous one; in this case the WM, since these resources aren't physically available at design time, generates the so called *virtual resources*, and make them available to subsequent workspaces through the Resource Manager.

The **Object Manager** (OM) copes with the management of the graphical objects associated with the resources chosen by the user to compose an application. Each object is associated with information regarding the resource which it refers to. This information is used during the creation of the internal model of the computation and to the end of generating the *execution plan* (see below). Objects, as already mentioned in the previous section, are data, software, and hosts.

During the composition phase, several operations can be applied to the objects such as: insertion in a given workspace, movement inside the owning workspace, selection, un-selection, linking with other objects and deletion from a workspace. All these operations are handled and supervised by the Object Manager. Moreover, it takes care of

the links labeling and the setting of their properties and/or attributes, like: file transfer destination path, parameters to be used in the software invocation, etc.

The **Consistency Checking** phase is needed to obtain a correct and consistent model of the computation. This process is accomplished not only as a checking of a set of requirements after the design phase is completed, but mainly through several interactions with the user while the design is in progress. Regarding the type of consistency checking and the time which it is performed on, two stages can be distinguished: *pre-processing* and *post-processing*. The pre-processing takes place simultaneously to the composition of the graphical representation made by the user. It operates in a context sensitive way, detecting situations that may lead to errors and undertaking actions to guide the user towards the right choices. The checking is completed with the post-processing, nonetheless needed to catch those error situations that are impossible to discover during the pre-processing phase.

When the consistency checking is carried out, it is time to parse the model of the computation to generate the *execution plan* and the specific *script* needed for the allocation. The generation of the execution plan is performed by the **Execution Plan Generator** (EPG) module. It parses the internal model of the computation and, on the basis of the properties of links and resources, produces an XML document describing the planned tasks, the so called execution plan. The execution plan describes the computation at a high level, without physical information about resources (identified by metadata references), and about the status and current availability of such resources. Specific information about the involved resources will be included during the translation into the script to be passed to the underlying middleware.

This script is constituted by a set of job requests expressed into a particular Resource Specification Language; it is produced by the **Script Generator** as a result of a subsequent elaboration on the execution plan.

The **Execution Manager** (EM) permits to start the execution of the application and handles communications between activated jobs and the user. The execution phase makes a direct use of specific GRAM services.

The EM includes as a sub-module the Job Monitor, whose purpose is to provide information about the executing jobs. In particular it shows to the user changes in the jobs status (pending, active, and so on) and whether they fail, giving her/him also the possibility to clean-up jobs from the queues where they are waiting in or to kill them while executing.

To start the execution, a valid proxy of the user authentication on the Grid must have been created. The Utilities section contains the *Authentication Manager* (see Fig. 4), which offers a visual facility to create and set up the proxy for accessing Grid resources.

Finally, the Utilities section comprises a functionality designed to serialize and restore the graphical model of the application. During the restoring, if the saved project contained virtual resources and the virtual resources were not materialized (because the saving was performed before the execution), these are restored as well.

5 Implementation

VEGA was implemented in Java, to guarantee the portability upon different platforms. Namely, the user interface components are all *Swing* based, whereas the access to specific Grid services is accomplished by using APIs of the Grid middleware.

The *Resource Manager* offers the so called resource pane, as showed in Fig. 5. It is divided into two subsections, one containing hosts and the other one the resources each host owns. Showed hosts are those chosen by the user, among ones available in the TMR, to be part of the current application. When a host is selected in the host sub-pane, the RM accesses the corresponding entry in the TMR, scans all metadata and shows the resources owned by this Grid node in the resource pane, each in the related category (software or data). To deal with XML documents, the RM makes use of the Xerces Java Parser [8], an open source implementation from Apache Software Foundation of the DOM (the W3C Document Object Model) [9]. Through the RM it is also possible to retrieve information about the status of the resources involved in the current project. For example, the user may query current CPU load and available memory, as well as static information like operating system name, etc.

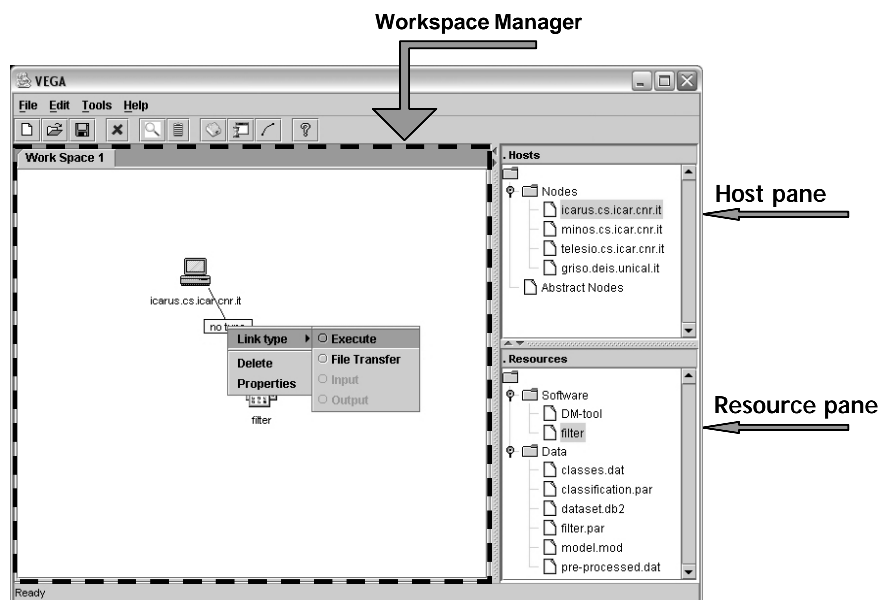

Fig. 5. VEGA user interface

To import a resource in a workspace the user can drag & drop it. When the RM detects the *drag-started* event, it provides the *Object Manager* with all the information found in the resource metadata document, so that the graphical object for that resource can be created.

The *Workspace Manager* performs a preliminary analysis of the composing computation, to discover conditions that may originate virtual resources: like the presence of a file transfer operation or the presence of an output resource generated by a job in the computation. When such a condition holds, the WM generates appropriate metadata and put them in the TMR, marking them as temporary entries (waiting for the real execution of the application). Metadata about virtual resources are homogeneous to ones about "real" resources and can be thus managed by the RM. Temporary entries, created during the design phase, will become permanent only when the execution is performed and it terminates successfully. Otherwise, if the user exits the working session without executing the application, they will be removed from the TMR. The user interface of the WM is basically constituted by a tabbed pane that allows for selecting the workspace on which to operate (see Fig. 5). The WM accomplishes also the task of building the *internal model* of the computation on the basis of the graphical composition made by the user and constituted by the workspaces sequence. The construction of the internal model takes place together with its graphical definition by the user.

The *Object Manager* can be thought as a "service module", in fact it hasn't a specific corresponding element in the user interface and most part of the code implementing it is distributed in several *listeners*. Java implements the *events driven programming paradigm*, in which listeners are classes notified of the occurrence of particular events (e.g. mouse and keyboard activities); these classes contain methods to handle such events.

The internal model consistency checking ensures that the model to be passed to the Model Analyzer is correct and without inconsistencies, that is, able to represent coherently a Grid application. The *Model pre-processor* operates during the application composition. Its main objective is to prevent the planning of jobs in a wrong or incomplete fashion. To this end, it supervises the links insertion, checking for the right association (see Table 1) between resource types and links.

When no type of link is admitted for a given couple of resources, each attempt to insert any link will fail. Whereas when one or more links are admitted, a generic link with a "no type" label is first traced, afterwards the user will specify the actual type by choosing among those available in a *popup menu*.

At the end of the design session, the user may have defined one or more workspaces. Although the workspaces composition has been guided by the Model pre-processor, there are some ambiguous situations that can be only recognized when the designing phase is over. Main constraints verified by the *Model post-processor* are:

- at least one host must be present in each workspace;
- all inserted links must have a specified type;
- each software component must be linked with at least one host;
- every resource must be linked with at least another one.

As previously mentioned, the *execution plan* is coded in XML and represents the application at a high level of abstraction; it is generated to make aware the Knowledge Grid Execution Plan Management Service (EPMS) of the computation structure (the formalism is well described in [10]). To generate the execution plan, the EPG analyzes the internal model of the computation to individuate all the jobs planned by the user.

Execution and file transfer jobs are described in the *execution plan*, accordingly with a set of well defined rules. In particular, they originate a set of `Computation` and `DataTransfer` elements, comprising attributes of the jobs and references to metadata of involved resources. To specify priority relationships among these jobs some `TaskLink` elements are used.

The *script* generation takes place on the basis of the execution plan and the information provided by metadata referred in it. All jobs in the execution plan are translated into requests expressed into the specific scripting language. For example, in the *RSL script generator* this is accomplished by assigning proper values to specific Globus RSL attributes. Details about Globus RSL tags can be found in [11]. Main attributes used by the VEGA *RSL script generator* are:

- `resourceManagerContact`, the node to which the request is to be submitted;
- `executable`, path and name of the program to run;
- `arguments`, a set of arguments to be passed to the program through the command line.
- `stdout`, the file to redirect standard output of the program to (if required by the user).

`Computation` elements in the execution plan are translated into job requests specifying the execution of a given executable with some inputs and outputs; as already mentioned, all details on executable inputs and outputs are retrieved in the referred XML files. `DataTransfer` elements are processed using as an executable the file transfer program provided by the Grid middleware, with source and destination parameters as indicated in the execution plan. If the middleware used is Globus all file transfer operations may be carried out using the GridFTP protocol [12] implemented by the file transfer program `globus-url-copy`. To reflect the workspace sequence in the jobs execution, job requests present in different workspaces are placed in separated script files and will be submitted to execution in strict sequence by the Execution Manager.

After the script files are generated, they will be executed through the proper submission command (i.e. `globusrun` in Globus). It is invoked by the VEGA Execution Manager taking into consideration each script file and allocating a new process for it on the machine which the user is working on. In addition, to provide the user with a feedback about the computation execution, standard output and error streams of that process are redirected to the EM and showed as well.

During this phase the **Job Monitor** gathers information about the activated jobs. The Job Monitor operates in strict relation with the middleware component in charge of the resource allocation, the GRAM. It periodically queries each node involved in the application about the status of the activated jobs, and update a set of information about the total execution time, the execution time of each job, its current status, the history of the statuses of each job, with enclosed the amount of time elapsed before a status change. This information is shown to the user in a table; moreover the user can clean up or kill each single job if needed, and eventually she/he can choose to save all this information into a report file.

6 Open Issues

In the current VEGA implementation workspaces can be connected in a pipelined fashion. This may represent a limitation of representing some computation patterns, even if complex patterns can be designed inside a single workspace. To make more flexible the way of composing workspaces, the sequential workspace composition is going to be replaced by an acyclic graph model.

In a varying and discontinuous environment such as a Grid, users' requests cannot be always deterministic in all details. It would be unacceptable, and flexibility loss leading, to pretend the user to specify all the details about the resources involved in a computation. Therefore, when the user doesn't worry about which will be the target machine for a given job, provided that it is able to satisfy a set of expressed requirements, it should be up to the system to find a suitable host and to assign it the job execution.

This could be also a powerful mechanism that can give the user the opportunity to design applications independent from the particular Grid on which they will be executed, hence, reusable upon different Grid systems and over time.

To this purpose the concept of "*abstract resources*" has been designed to be introduced in VEGA, This concept allows for specifying resources by means of constraints (i.e., required main memory, disk space, CPU speed, operating system version, etc.). In addition, a *meta-scheduler* will also be included to instantiate abstract resources. After the appropriate matching of abstract resources with physical ones and an optimization phase, the system can submit for execution all the jobs defined in the application design on the basis of the application structured layout.

VEGA defines in a particular way "file transfer" jobs, permitting also to specify and configure a particular protocol to be used in the transferring, this because the file transfer has been historically a fundamental mechanism of communications between team of scientists and the most used one in Grid applications. The existence of specific services and features such as those provided by the so called "Data Grids" confirms the particular importance of this matter.

Nevertheless, there are a lot of applications that need to operate on streams of data coming from a sensor or more generally from a network connection. Moreover, some other applications might achieve best performances, even if file-based, if it was possible for them to start their computations while the stream of data is reaching the Grid node to be stored on the file system (see Section 7.2 for an example).

7 Case Studies

This section presents two case studies through which the main features and potentials of VEGA will be better explored. At the same time a practical use of VEGA will be shown and some issues that may arise in this kind of applications will be analyzed. The first one consists of a Grid-enabled version of a knowledge discovery application, the second one is an example of a general purpose data intensive application.

7.1 Distributed Bank Scoring

This example takes into consideration the evaluation process done by banks when approving a loan. Loan officers must be able to identify potential credit risks and decide whether grant the money, and, in that case, the amount of the loan. Usually this is accomplished by evaluating information about people to whom the institution previously loaned money (such as debt level, income level, marital status, etc.).

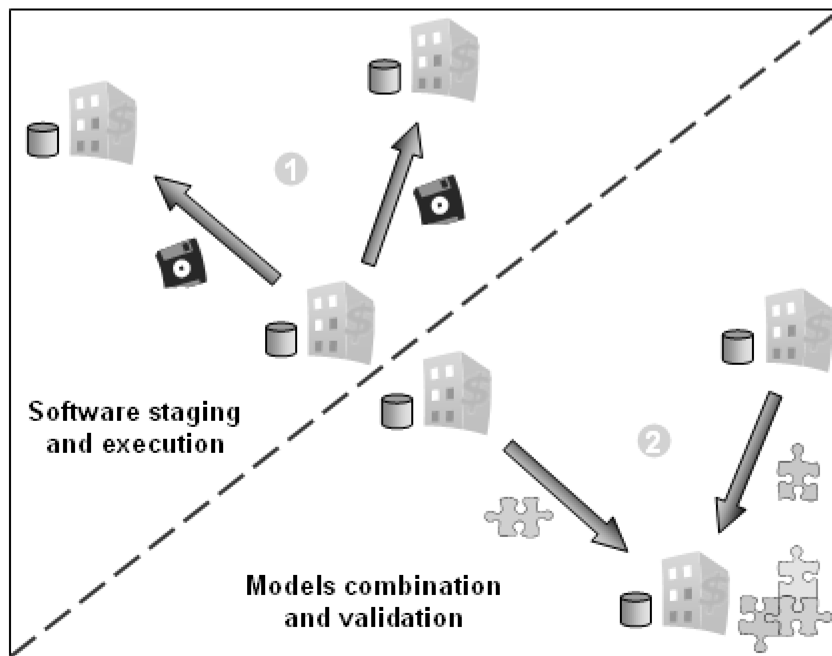

Fig. 6. Distributed bank scoring application

Let us consider the case of three banks whose purpose is to join their efforts and extract a loan-scoring prediction model, based on the information each of them own about their clients. Since preserving client's privacy is a must for credit institutes, the treatment of this information by third parts is often prohibited or subject to restrictions. To overcome this problem the banks decide to make the computations locally, at each of the three sites, and to transfer on a centralized location only the obtained models.

In this way, no sensitive information has to be accessed by unauthorized organizations, but only the data models have to be shared. Privacy commitments are thus assured, because the data models are constituted by coded information about aggregated data.

When a decision is based on several factors, a decision tree can help to identify which factors to consider and how each factor has historically been associated with different outcomes of the decision. A decision tree creates a model as either a graphical tree or a set of text rules that can predict (classify) each applicant as a good or bad credit risk. The training process that creates the decision tree is usually called *induction*. One important characteristic of the tree splitting algorithm is that it is greedy. Greedy algorithms make decisions locally rather than globally. When deciding on a split at a particular node, a greedy algorithm does not look forward in the tree to see if another decision would produce a better overall result. This allows for creating partial models from subsets of the data that can be then joined into a global model.

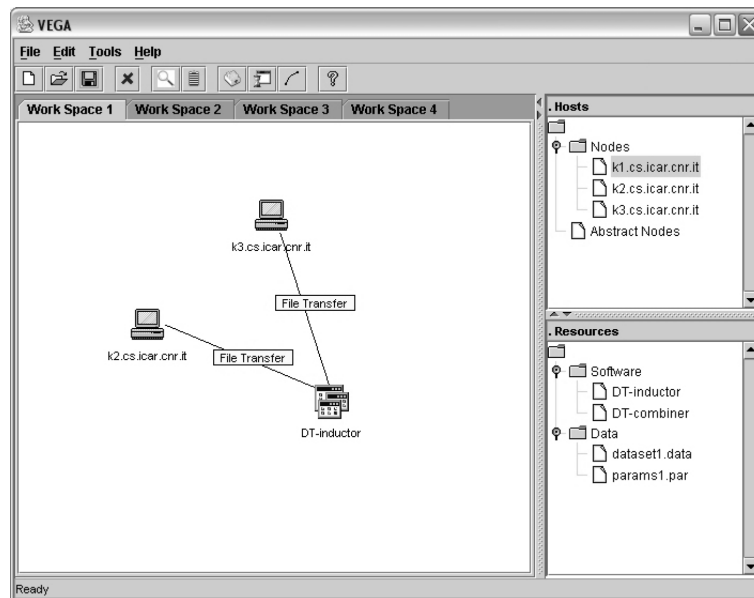

Fig. 7. Distributed bank scoring: workspace 1

Prior to integrating any decision tree into a business process as a predictor, a test and a validation of the model using an independent dataset is generally performed. Once accuracy has been measured on an independent dataset and is determined to be acceptable, the tree (or a set of production rules) is ready to be used as a predictor. The testing phase in this example is done after the combination of the three models.

To summarize, the entire application is composed of two main phases (see Fig. 6 for a graphical schema): the induction of the decision trees, performed locally at each of the three banks, and the models combination and validation operated at one of the three sites after the others models have been produced and moved there.

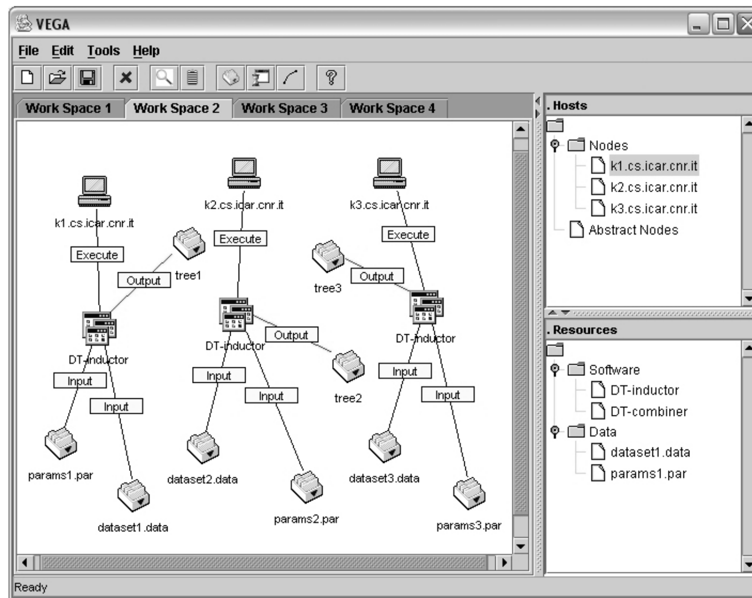

Fig. 8. Distributed bank scoring: workspace 2

Let the Grid nodes made available by the banks be: `k1.cs.icar.cnr.it`, `k2.cs.icar.cnr.it`, and `k3.cs.icar.cnr.it`. The application design and submission will be performed on the Knowledge Grid node `k1` that will be also the node on which the final model will be obtained from the partial ones. All nodes contain a dataset about the clients of the bank and a parameters file with a description of the structure of the dataset, this file is used by the inductor also to determine which the dependent variable is and which columns have to be considered as independent variables. The datasets are named respectively `dataset1.data`, `dataset2.data`, and `dataset3.data`; while the associated parameters files are `parameters1.par`, `parameters2.par`, and `parameters3.par`. On `k1` the software components `DT-inductor` and `DT-combiner` are also present.

The design of the application using VEGA produces four workspaces. First of all, it is necessary to transfer a copy of the software used for the induction to nodes `k2` and `k3` (where it is not available); this step is planned in workspace1 (see Fig. 7). Afterwards, the trees induction can take place at each of the three hosts by executing `DT-inductor` with the dataset and parameters file as inputs, see workspace 2 in Fig. 8. As a result of the computations in workspace 2, three files (`tree1`, `tree2`, `tree3`) containing the resulting partial trees will be obtained on each host. The subsequent stage performs the transferring of `tree2` and `tree3` to `k1`, so as to have all the trees

on the same node (see Fig. 9). The combination of the partial trees into a global one will next happen on k1 by means of the DT-combiner tool, as can be seen in Fig. 10

From workspaces 3 and 4, it is possible to note that as a direct consequence of the transfer of tree1 and tree2 to k1 in workspace 3, they are shown in workspace 4 as data resources of k1, even if the execution of the application has not been performed yet. This outcome is due to the intervention of the Workspace Manager that creates the needed virtual resources so as to allow for the use of this data in subsequent computations.

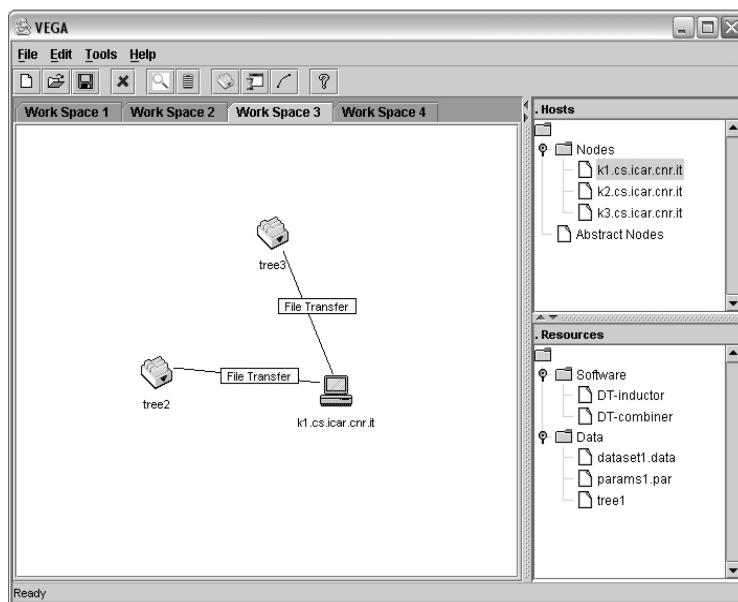

Fig. 9. Distributed bank scoring: workspace 3

VEGA generates four Globus RSL script files (bank_scoring0.rsl, bank_scoring1.rsl, bank_scoring2.rsl, bank_scoring3.rsl), one for each workspace, containing the formal description of the jobs to be executed. When requested by the user, the execution will be launched by the environment submitting the generated RSL files in sequence to the Globus GRAM.

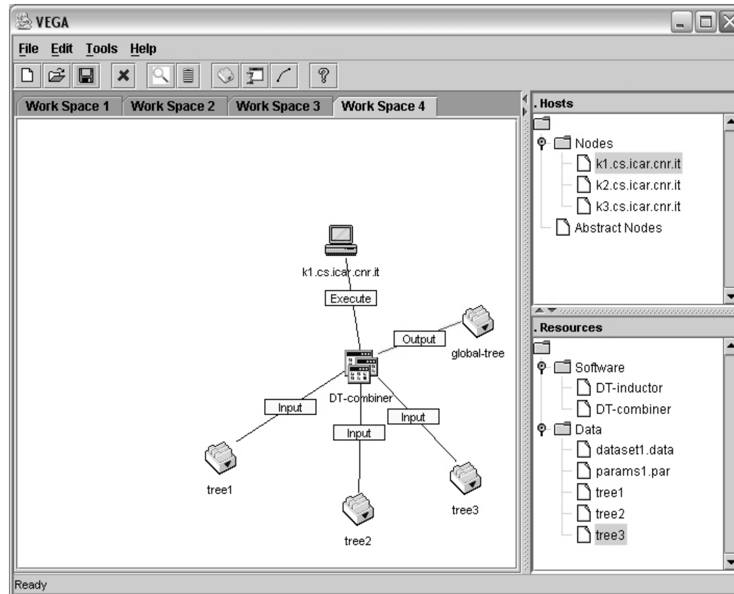

Fig. 10. Distributed bank scoring: workspace 3

The following figures present the content of the Globus RSL files. From a quick look is easy to understand the difference between the VEGA programming model and the low-level programming model offered by the Globus RSL. The high-level properties of the visual approach bring several benefits to developers both in terms of structured approach, easy programming, and code reuse.

```
+
( &(resourceManagerContact=k2.cs.icar.cnr.it)
   (label=subjob1)
   (executable=$(GLOBUS_LOCATION)/bin/globus-url-copy)
   (arguments=-vb -notpt gsiftp://k1.cs.icar.cnr.it/tools/bin/DT-inductor
               gsiftp://k2.cs.icar.cnr.it/tools/bin/DT-inductor
   )
)
( &(resourceManagerContact=k3.cs.icar.cnr.it)
   (label=subjob2)
   (executable=$(GLOBUS_LOCATION)/bin/globus-url-copy)
   (arguments=-vb -notpt gsiftp://k1.cs.icar.cnr.it/tools/bin/DT-inductor
               gsiftp://k3.cs.icar.cnr.it/tools/bin/DT-inductor
   )
)
```

Fig. 11. file bank_scoring0.rsl

```
+
( &(resourceManagerContact=k1.cs.icar.cnr.it)
   (label=subjob1)
   (executable=/tools/bin/DT-inductor)
   (arguments=  -d /data/dataset1.data
                -p /data/params1.par
                -o /data/tree1
   )
)
( &(resourceManagerContact=k2.cs.icar.cnr.it)
   (label=subjob2)
   (executable=/tools/bin/DT-inductor)
   (arguments=  -d /data/dataset2.data
                -p /data/params2.par
                -o /data/tree2
   )
)
( &(resourceManagerContact=k3.cs.icar.cnr.it)
   (label=subjob3)
   (executable=/tools/bin/DT-inductor)
   (arguments=  -d /data/dataset3.data
                -p /data/params3.par
                -o /data/tree3
   )
)
```

Fig. 12. file bank_scoring1.rsl

```
+
( &(resourceManagerContact=k1.cs.icar.cnr.it)
   (label=subjob1)
   (executable=$(GLOBUS_LOCATION)/bin/globus-url-copy)
   (arguments=-vb -notpt gsiftp://k2.cs.icar.cnr.it/home/data/tree2
              gsiftp://k1.cs.icar.cnr.it/home/data/tree2
   )
)
( &(resourceManagerContact=k1.cs.icar.cnr.it)
   (label=subjob2)
   (executable=$(GLOBUS_LOCATION)/bin/globus-url-copy)
   (arguments=-vb -notpt gsiftp://k3.cs.icar.cnr.it/home/data/tree3
              gsiftp://k1.cs.icar.cnr.it/home/data/tree3
   )
)
```

Fig. 13. file bank_scoring2.rsl

7.2 A Video Conversion Application

This example has been taken from a demo developed at IBM laboratories [17], and is aimed at showing main advantages coming from the use of a visual environment such as VEGA in respect of the classical approach, as well as the ability of VEGA to deal with general purpose Grid applications.

The application starts from a home video tape and converts it to a Video-CD that can be played on DVD players supporting this format. Depending on the quality level, a typical one hour tape can create over 10 GB of video data, which needs to be compressed to approximately 650 MB to fit on a Video-CD. The compression stage is CPU intensive, since it creates an MPEG data stream by encoding the frames after a matching process performed on all parts of adjacent video frames containing similar sub-pictures. The audio is compressed as well.

The compression process can take even more than one day, depending on the quality level and the speed of the system being used. For commercial DVD quality, conversions are typically done by a service company that has developed higher quality conversion algorithms. Such conversions may take weeks. Hence, Grid technology is ideal for improving the process of video conversion.

Sending many gigabytes of data from one computer to another takes a considerable amount of time, even with a 100 Mb Ethernet connection. Thus, for Grid applications processing large amounts of data, it is crucial to understand the network topology and to keep the data near the processing node that needs to use it.

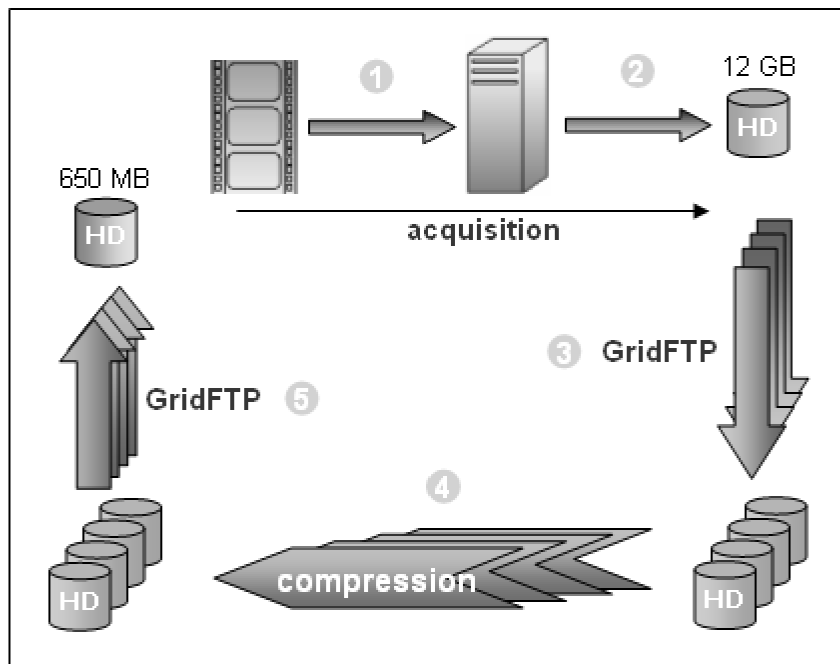

Fig. 14. The video conversion application

Let us assume that all packages and libraries required for performing the acquisition, the compression and the CD burning have been already installed and properly configured on related machines. Since, each phase of the application makes use of several calculations carried out by different software, to simplify the comprehension,

instead of executing them directly, some shell scripts invoking these tools will be used (namely, `videocapture.sh`, `videoconversion.sh` and `videocd.sh`).

```
#!/bin/sh
#First, capture video from DVcamera
./videocapture.sh
#set environment variables
target_dir=/home/globususer
curdir=`pwd`
#stage code and video to remote machines
ndx=1
# List machines to be used here and below (one conversion per machine):
for target_host in minos.cs.icar.cnr.it icarus.cs.icar.cnr.it
                     telesio.cs.icar.cnr.it
do
echo Setting up demo on host $target_host
globus-url-copy file:${curdir}/videoconversion.sh

gsiftp://${target_host}:2811${target_dir}/videoconversion.sh
echo sending video ${curdir}/videocap00${ndx}.avi
globus-url-copy file:${curdir}/videocap00${ndx}.avi \
gsiftp://${target_host}:2811${target_dir}/videocap00${ndx}.avi
globus-job-run ${target_host} /bin/chmod 755 videoconversion.sh
echo Building RSL for $target_host
echo +>demo_rsl${ndx}
echo "( &(resourceManagerContact="${target_host}")" >>demo_rsl${ndx}
echo " (subjobStartType=strict-barrier)" >> demo_rsl${ndx}
echo " (label="videocap00${ndx}")" >> demo_rsl${ndx}
echo " (executable= ${target_dir}/videoconversion.sh)" >> demo_rsl${ndx}
echo " (arguments = videocap00${ndx}.avi )" >> demo_rsl${ndx}
echo ' (stdout= $(GLOBUSRUN_GASS_URL) \
# "'$curdir/videocap00${ndx}.out'")' \>> demo_rsl${ndx}
echo ' (stderr= $(GLOBUSRUN_GASS_URL) \
# "'$curdir/videocap00${ndx}.err'")' >> demo_rsl${ndx}
echo ")" >> demo_rsl${ndx}
# Jobs submission
echo submiting job to $target_host
globusrun -w -f demo_rsl${ndx} &
ndx=`expr $ndx + 1`
done
echo waiting for all conversions to complete
wait
echo getting result files now
rm -f videocap.mpg
# Getting compressed files
ndx=1
for target_host in minos.cs.icar.cnr.it icarus.cs.icar.cnr.it
                    telesio.cs.icar.cnr.it
do
globus-url-copy
gsiftp://${target_host}:2811${target_dir}/videocap00${ndx}.avi.mpg \
file:${curdir}/videocap00${ndx}.avi.mpg
cat videocap00${ndx}.avi.mpg >> videocap.mpg
rm -f videocap00${ndx}.*
rm -f demo_rsl${ndx}
ndx=`expr $ndx + 1`
done
# Now create the video cd (VCD)
./videocd.sh
```

Fig. 15. Shell script for the video conversion application

In this example the acquisition node is the grid node `griso.deis.unical.it`, whereas the nodes used to execute the compression of the split video files are `minos.cs.icar.cnr.it`, `icarus.cs.icar.cnr.it` and `telesio.cs.icar.cnr.it`.

After the capture phase, the video file is then split into a number of smaller files. These files are sent via Globus to Linux based grid systems for compression. The compressed segments are then reassembled and a CD is written in the VCD format. Fig. 14 gives a conceptual schema of the entire process.

Following a classical approach to grid problems, a set of operations have to be executed to prepare and start the execution of the processing jobs on the grid nodes. These operations are shown in the shell script of Fig. 15, in which the different phases are marked in bold (video files creation, transfer of the video files and the conversion software, compression, retrieving of the compressed files, CD burning).

When the same application is designed using VEGA, it originates the workspace sequence reported by Fig. 16 and the following ones. Workspaces 1 and 2 are used to record the video files and to move them and the `videoconversion.sh` script to the hosts on which the conversion will take place. Workspace 3 executes concurrently the compression on the nodes `minos`, `icarus` and `telesio`; workspace 4 moves the resulting files to `griso`, the origin node. Finally, workspaces 5 and 6 create a unique file and write it on a recordable CD.

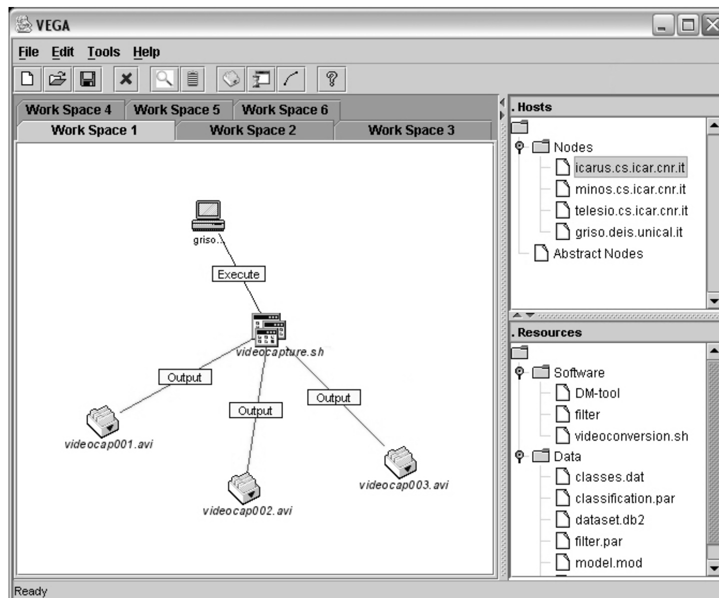

Fig. 16. Video conversion: workspace1

Making a comparison of the two approaches, it is possible to notice that while the logical phases that form the application are, obviously, the same, there is a fundamental difference in how the application will be executed. The shell script is replaced in

the VEGA version of the application by pure RSL scripts, since all the operations required to configure and run the video conversion application are performed using Globus services.

This means that such an application is really architecture independent, because based on Globus services. Hence it is reusable on different Grids and over time. In addition, thanks to this representation it would be simple, by using VEGA, to reconfigure the application to fit with a different Grid context or simply to change some parameters.

There are some changes that can be made to improve the performance of the video conversion application. One of these changes is to begin the video file transfer during the capture phase. Once the first video file is obtained, it can be staged to the remote machine and the conversion can begin. Furthermore, using MDS, it may be possible to locate on the Grid the machine with a low CPU load and send the video file to it. To this end, the considerations made in the open issues section are of primary importance and could provide support for application able to better exploit the Grid infrastructure.

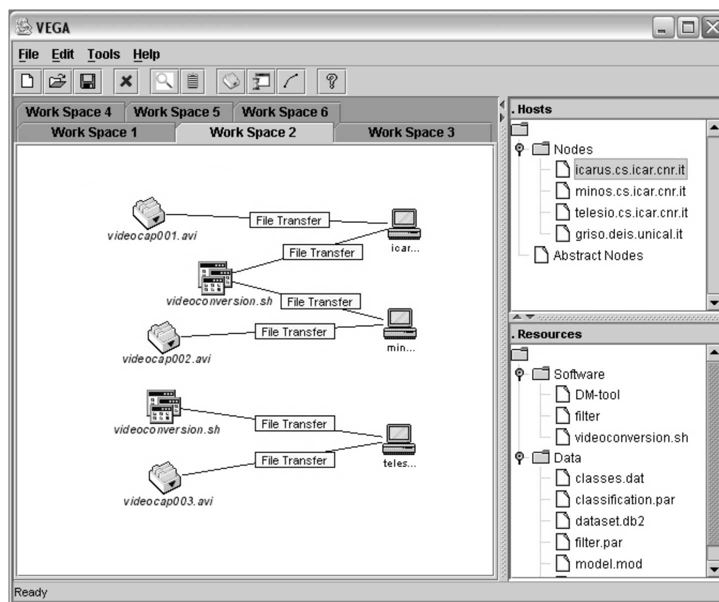

Fig. 17. Video conversion: workspace 2

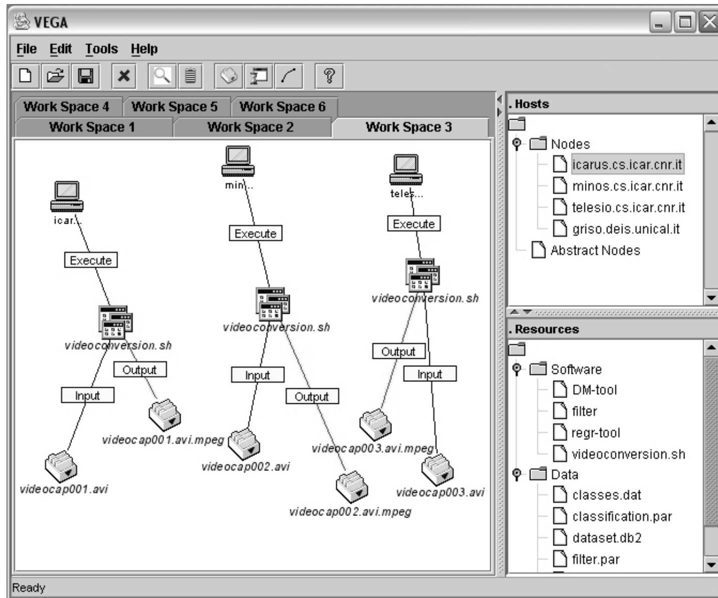

Fig. 18. Video conversion: workspace 3

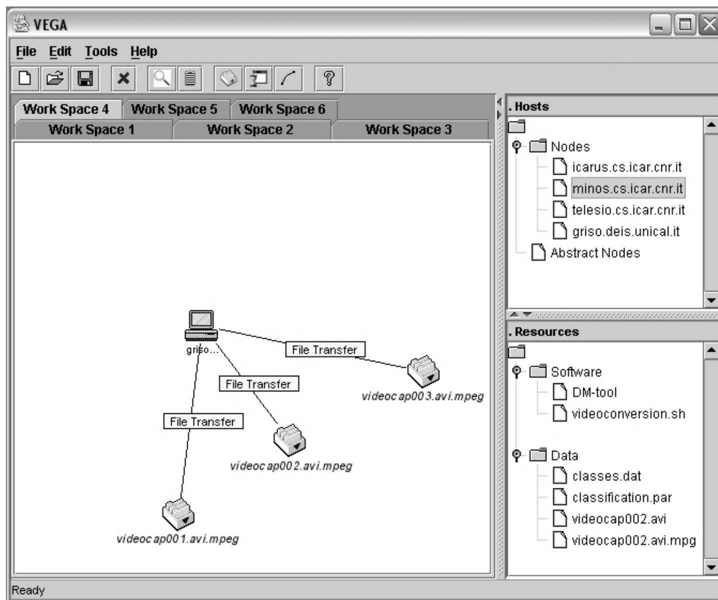

Fig. 19. Video conversion: workspace 4

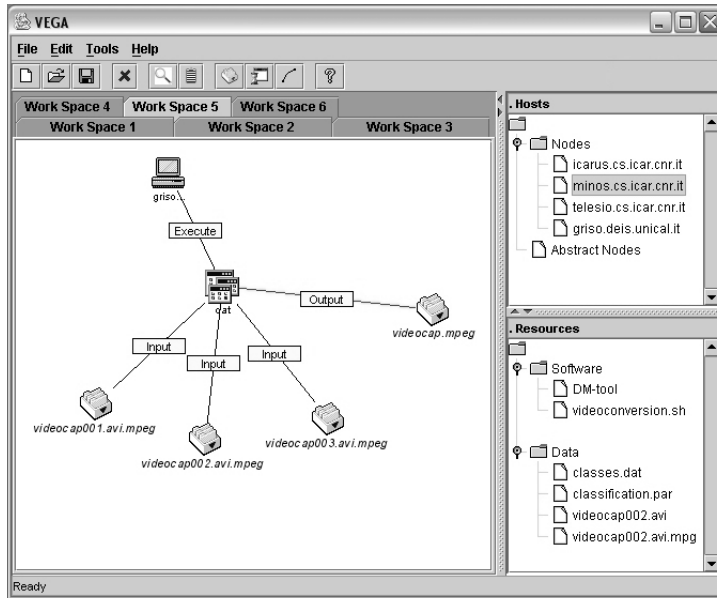

Fig. 20. Video conversion: workspace 5

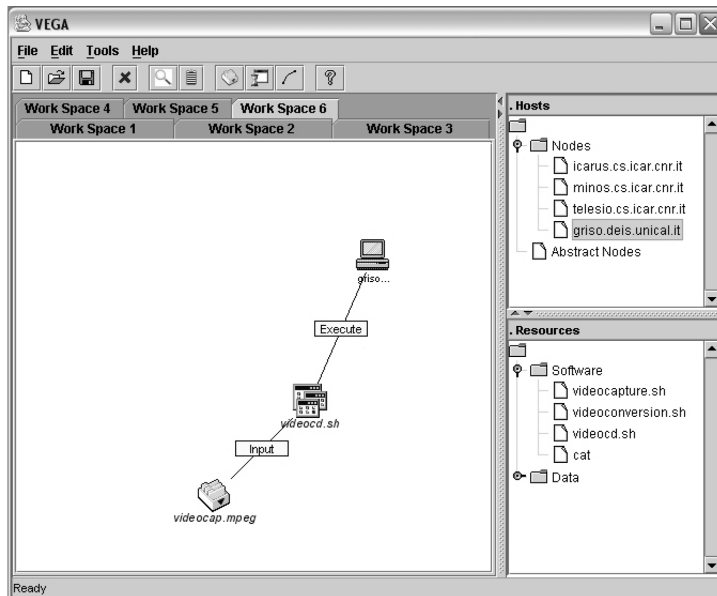

Fig. 21. Video conversion: workspace 6

8 Related Work

This section briefly describes some related projects and tools, giving also a short comparison of the common and distinctive features between them and the environment we presented here.

A Grid-based knowledge discovery environment that shares some goals with the Knowledge Grid is *Discovery Net* (D-Net) [13]. The D-Net main goal is to design, develop, and deploy an infrastructure to support real time processing, integration, visualization, and mining of massive amount of time critical data generated by high throughput devices. The building blocks in Discovery Net are the so-called *Knowledge Discovery Services* (*KDS*), distinguished in *Computation Services* and *Data Services*. The former typically comprise algorithms, e.g. data preparation and data mining, while the latter define relational tables (as queries) and other data sources. Both kinds of services are described (and registered) by means of *adapters*, providing information such as input and output types, parameters, location and/or platform/operating system constraints, *factories* (objects allowing to retrieve references to services and to download them), *keywords* and a *description*. KDS are used to compose moderately complex data-pipelined processes. The composition may be carried out by means of a GUI which provides access to a library of services. The XML-based language used to describe processes is called *Discovery Process Markup Language*. D-Net is based on an open architecture using common protocols and infrastructures such as the Globus Toolkit.

The *Parallel Application WorkSpace* (PAWS) [14] is a software infrastructure for connecting separate parallel applications within a component-like model. PAWS provides also for dynamically coupling of applications and supports efficient communication of distributed data structures. The PAWS Controller coordinates the coupling of applications, manages resources, and handles user authentication. Heterogeneity issues in PAWS are handled by the underlying Nexus library. Currently, PAWS is a C++ library (C and Fortran interfaces are under development). Applications written in any language that may incorporate such libraries can be interconnected with PAWS and may communicate exploiting the common PAWS layer. PAWS is designed to coordinate a parallel execution of multiple, interconnected programs, to this end multiple communication channels are exploited. For employing optimized communication schedules, PAWS requires information on the layout, the location, and the storage type of the data, all of which has to be provided by the user through appropriate PawsData objects.

Recently, a few general purpose grid programming tools have been developed or are going to be developed. *Graph Enabled Console COmponent* (GECCO) is a graphical tool developed at Argonne National Laboratory [15][16]. GECCO is based on the Globus CoG Kit [5] and provides facilities to specify and monitor the execution of sets of tasks with dependencies between them. Specifically it allows to specify the jobs dependencies graphically, or with the help of an XML-based configuration file, and execute the resulting application. Each job is represented as a node in a graph. A job is executed as soon as its predecessors are reported as having 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 RSL, the label, and other

parameters. Editing can also be performed at runtime (job execution), hence providing for simple computational steering.

These systems show how problems and issues of Grid-based generic, parallel and knowledge discovery applications are addressed and solved in various contexts. It can be noted that some approaches are similar to that defined into the Knowledge Grid architecture and used by VEGA, like the composition of tasks and the employment of a XML based formalism to represent the structure of the application. On the other hand, several differences are also present, above all the role and structure of the *execution plan* and the use in VEGA of a metadata based *information system* (KDS) from which extracting information about grid nodes and datasets characteristics.

VEGA, as part of the Knowledge Grid, provides access to a set of services for generic and knowledge discovery applications. An application running into the VEGA environment does not contain any limitation about the processing strategy to employ (i.e. move data, move model, etc.), neither about the number and the location of the grid nodes that will perform a mining process. The integration and use of new data access methods or processing algorithms, as well as entire commercial suite or software components coming from pre-existent sequential or parallel systems, is simple and does not require any customization. It is obtained by their publication in the KDS, which will provide the system with all needed information to use that component inside an application (i.e. invocation syntax, component requirements, etc.). The XML-based approach used in the Knowledge Grid and VEGA to define metadata is going to be the most used in several Grid-based environments and also the new version of the Globus Toolkit (GT3) exploits XML-based metadata for handling resource management.

9 Conclusion

A condition to bring Grid computing to a mature phase is the availability of high-level supporting tools and development environments that allows users and developers to effectively exploit Grid features in designing advanced applications. Here we presented VEGA, a high-level *Visual Environment for Grid Application* designed to support the design and execution of complex applications upon Grid environments.

VEGA offers the users a programming model that represents Grid resources as a collection of typed resources and a set of defined "relationships" between them. As part of the implementation of the Knowledge Grid, VEGA interacts with some of its services. In particular, the *knowledge directory service* is widely used to retrieve basic information about Grid resources. The key concepts in the VEGA approach to the design of a Grid application are the *visual language* used to describe the jobs constituting an application, and the methodology to group these jobs in *workspaces* to form a specific stage. These are also the features that make the environment provided by VEGA adhere to the *software component framework*, that is, a system for composing application from smaller software modules.

The software modules composing the VEGA architecture implement a set of functionalities able to simplify the planning and submission of complex applications, pro-

viding an easy access to Grid facilities with a high level of abstraction. These functionalities range from *design facilities* to *consistency checking*, *execution management*, *credentials management*, and *projects management*.

All these features have been developed specifically to support the design of data analysis and knowledge discovery applications, but are suitable to satisfy the requirements of most general purpose applications. The case studies presented in Section 7 are intended to show a practical use of VEGA, as well as to demonstrate how VEGA can handle a typical Grid application and to illustrate the main benefits in comparison with the still predominant low-level approach.

The open issues section discussed some improvements (part of which are already under development) that could be added to the system. In particular the *acyclic graph* hypothesis for the workspaces and the *abstract resources* concept are key features to open the way towards larger and more complex classes of applications.

References

1. Microsoft Corporation, ".NET", see http://www.microsoft.com.
2. A. Thomas, "Enterprise JavaBeans Technology: Server Component Model for the Java Platform", http://java.sun.com/products/ejb/white_paper.html, 1998.
3. I. Foster & C. Kesselman, "Globus: a metacomputing infrastructure toolkit", *Int. Journal of Supercomputing Applications*, 1997, vol. 11, pp. 115-128.
4. I. Foster, C. Kesselman, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations", *Int. Journal of Supercomputer Applications*, 2001, vol. 15, n. 3.
5. The Globus Project, "Java Commodity Grid Kit", see http://www.globus.org/cog/java.
6. M. Cannataro, D. Talia, "KNOWLEDGE GRID: An Architecture for Distributed Knowledge Discovery", *Communications of the ACM*, January 2003.
7. C. Mastroianni, D. Talia, P. Trunfio, "Managing Heterogeneous Resources in Data Mining Applications on Grids Using XML-Based Metadata", *Proc. IPDPS 12th Heterogeneous Computing Workshop*, Nice, France, April 2003.
8. The Apache Software Foundation, "Xerces Java Parser 2.0.0", available at http://xml.apache.org.
9. World Wide Web Consortium, "Document Object Model (DOM) Level 3 XPath Specification", see http://www.w3.org/TR/DOM-Level-3-XPath.
10. M. Cannataro, A. Congiusta, D. Talia, P. Trunfio, "A Data Mining Toolset for Distributed High-Performance Platforms", *Proc. 3rd Int. Conference Data Mining 2002*, WIT Press, Bologna, Italy, September 2002, pp. 41-50.
11. The Globus Project, "The Globus Resource Specification Language RSL v1.0", see http://www.globus.org/ gram/rsl_spec1.html.
12. W.Allcock, "GridFTP Update January 2002", available at http://www.globus.org/datagrid/deliverables/GridFTP-Overview-200201.pdf.
13. V. Curcin, M. Ghanem, Y. Guo, M. Kohler, A. Rowe, J. Syed, P.Wendel, "Discovery Net: Towards a Grid of Knowledge Discovery", *Proc. Eighth ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, Edmonton, Canada, 2002.
14. P. Beckman, P. Fasel, W. Humphrey, and S. Mniszewski, "Efficient Coupling of Parallel Applications Using PAWS", *Proceedings HPDC*, Chicago, IL, July 1998.
15. G. von Laszewski, "A Loosely Coupled Metacomputer: Cooperating Job Submissions across Multiple Supercomputing Sites", *Concurrency, Experience, and Practice*, Mar. 2000.
16. G. von Laszewski and I. Foster, "Grid Infrastructure to Support Science Portals for Large Scale Instruments", *Distributed Computing on the Web Workshop (DCW)*, University of Rostock, Germany, June 1999.
17. IBM Grid computing, see http://www.ibm.com/grid/.