Grid Interactive Service-Oriented Programming

Michael Sobolewski, Texas Tech University
sobol@cs.ttu.edu
Raymond Kolonay, Air Force Research Laboratory, WPAFB
raymond.kolonay@wpafb.af.mil

Abstract

Improvements in distributed computing, and the technologies that enable them, have led to significant improvements in middleware functionality and quality, mainly through networking and protocols. However, the distributed programming style is the same as ten, twenty, even thirty years ago. Most programs are still written line per line of code in languages like C, C++, and Java. These conventional programs that can provide grid operations and grid data can be considered as common grid resources and shared by research and education communities worldwide. However, there are no relevant programming methodologies to utilize efficiently these shared service providers as a potentially vast grid repository, except through the manual writing of code - just as it was done decades ago. Realization of the potential of grid computing requires significant improvements in grid programming methodologies. The Grid Interactive Service-Oriented (GISO) methodology is presented that provides a programming environment with development tools that permit interactive (point-and-click), true grid programming, thus permitting the different elements of programming to be stored, reused, aggregated, and executed with a level of concurrency and grid-level control strategy not achievable in the conventional programming languages.

1. Introduction

From the very beginning of networked computing, the desire has existed to develop protocols and methods that facilitate the ability of people and automatic processes across different computers to share information and knowledge across heterogeneous systems. As ARPANET [1] began through the involvement of the NSF [2,3] to evolve into the Internet for general use, the steady stream of ideas became a flood of techniques to submit, control, and schedule jobs across distributed systems [4]. The latest in these ideas is the *grid* [30-35], to be used by a wide variety of different users in a non-hierarchical manner to provide access to powerful aggregates of resources [5,6]. Grids, in the ideal, are intended to be accessed for computation, data storage and distribution, and visualization and display, among other applications without undue regard for the specific nature of the hardware and underlying operating systems on the resources on which these jobs are carried out [7,8].

The reality at present, however, is that grid resources are still very difficult for most users to access, and that detailed programming must be carried out by the user through command line and script execution to carefully tailor jobs on each end to the resources on which they will run, or for the data structure that they will access. This produces frustration on the part of the user, delays in adoption of grid techniques, and a multiplicity of specialized "grid-aware" tools that are not, in fact, aware of each other that defeat the basic purpose of the grid.

Most programs are still presently written line per line of code in compiled languages such as FORTRAN, C, C++, Java and or scripting languages such as Perl and Python. The current state of the art is that these scripts and programs can provide grid operations (by ***method providers***) and grid data (by ***context providers***) that can then be considered as common ***grid resources*** (the analog of conventional programming libraries) and shared by research and education communities worldwide. However, there are presently no relevant programming methodologies to deploy and access these ***service providers*** efficiently as a potential vast grid repository, except by writing code as it was done decades ago.

The need for further improvements in grid computing is clear, and requires significant further improvements in grid programming technology. By inspection of the above paradigm, it is clear that incremental improvements in the scripts and submission techniques will not suffice. A new ***grid interactive service-oriented (GISO)*** integrated development environment (IDE) that is based on evolution of the concepts and lessons learned in the FIPER project [9-14], a $21.5 million program founded by NIST, is presented. It provides an environment that will permit true interactive click-and-drag grid programming through the manipulation of graphical elements that

represent object-oriented grid resources, thus permitting the different elements of grid program to store, reuse, aggregate, and execute with a level of concurrency and grid-level control strategy not achievable in the conventional programming languages.

The presented GISO programming approach is characterized as follows:

1. Service-oriented grid programming is achieved by applying the object-oriented concepts directly to the grid as a repository of network objects (method and context providers)
2. Service-oriented execution infrastructure enabling dynamic federations of grid providers to execute service-oriented programs
3. Provisioning and deploying grid objects with an autonomic behavior, enabling grid objects to be instantiated and managed on compute resources available through the grid using an adaptive quality of service model
4. An open, web-based environment in which existing proprietary applications and analytical packages are integrated through Java-based wrappers that handle grid processes and data distributed across different locations.

The presented approach addresses a number of gaps which exist in the grid technology. The technology gaps and approach to solving these gaps are articulated in Table 1.

Technology Gap	GISO Solution
1. Protocol-based grid middleware is difficult to use	Develop object-oriented middleware components
2. Current grid middleware is transaction, data and host centric	Provide autonomic, dynamic, QoS, network centric middleware components
3. No grid-oriented programming methodologies to utilize grid resources and middleware services	Provide point-and-click interactive grid programming
4. Moving executable code and data over grid to compute resources is inefficient	Provide reusable method and data services as service-to-service grid providers
5. Access to grid resources is not user friendly	GISO easy-to-use web-based end-user-agents
6. No grid high-level programming and development tools	Develop interactive grid-programming and development tools

Table 1. Technology gaps vs. the GISO approach

2. GISO Conceptual Framework

The ***structured computing*** paradigm is a strategy based on a concept that a system has ***data*** and ***functionality*** (behavior) separated into two distinct parts. The ***object-oriented*** paradigm, on the other hand, defines a system as a collection of interacting ***active objects***. These objects ***do things*** and ***know things***, or stated equivalently they have functions and data that complement each other. Usually an object-oriented system creates its own ***object space*** instead of accessing a data repository. This object space constitutes an ***object-oriented program***.

Building on the object-oriented paradigm is the ***service-oriented*** paradigm, in which the objects are distributed, or more precisely they are ***network objects*** and play some predefined roles. A ***service provider*** is an object that accepts messages from ***service requestors*** to execute an item of work – a ***task***. The task object is a service request – a kind of elementary grid instruction executed by a service provider. A service ***jobber*** is a specialized service provider that executes a ***job*** – a compound request in terms of tasks and other jobs. The job object is a ***service-oriented program*** that is dynamically bound to all relevant and currently available service providers on the grid. This collection of grid providers dynamically identified by a jobber is called a ***job federation***. This federation is also called a ***job space***. While this sounds similar to the object-oriented paradigm, it really isn't. In the object-oriented paradigm the object space is a program itself; here the job space is the ***execution environment*** for the job itself and the job is a service-oriented program. This changes the game completely. In the former case the object space is a ***virtual computer***, but in the latter case the job space is the ***virtual network***. This virtual network or ***grid federation*** is the jobs' execution environment and the ***job object*** is a service-oriented program. In other words, we apply the object-oriented concepts directly to the grid in the service-oriented manner.

The GISO framework is built on the top of the FIPER Technology middleware. FIPER is a four year National Institute of Standards and Technology (NIST) Advanced Technology Program (ATP) which teams General

Electric, Goodrich, Parker Hannifin, Engineous Software, Ohio University, Stanford University, Texas Tech University and Ohio Aerospace Institute to develop a distributed large-scale engineering design system. Here and throughout the remainder of this paper FIPER refers to the FIPER Technology and proto-type implementation developed by the General Electric Global Research Center and it's sub-contractor Texas Tech University. The GISO environment provides the means to create interactive service-oriented programs and execute them without writing a line of source code. Jobs and tasks are created using web-based user interfaces. Also via web-based interfaces the user can execute and monitor the execution of jobs or tasks. The jobs and tasks are persisted for later reuse. This feature allows the user quickly to create new applications or programs on the fly in terms of existing tasks and jobs. Jobs created this way might be used later with their own custom user interfaces that collect input from the user and update a job accordingly before its execution as a service-oriented program.

FIPER supports three centricities and deploys three neutralities. FIPER's three centricities are network centricity, service centricity, and web centricity. A GISO federation is composed of various service providers any of these can come and go, and the system can respond to changes in its environment in a reliable way (network centricity). Services in GISO can discover lookup services and join the grid or lookup for relevant services in order to cooperate in a grid federation (service centricity). Users can request to use multiple services and check the status of their submissions in different locations through a FIPER HTTP portal with thin web clients (web centricity).

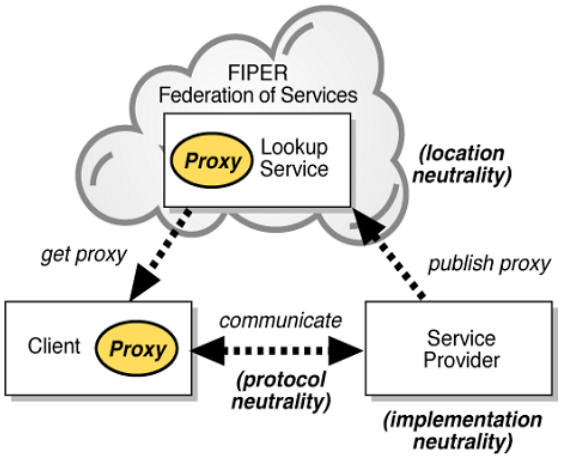

Figure 1. FIPER three neutralities, providing a, highly flexible grid environment.

The three neutralities FIPER deploys are location neutrality, protocol neutrality, and implementation neutrality as illustrated in Figure 1. With location neutrality, services need not be collocated; lookup services are discovered and used to find a particular service, which simplifies management of the entire grid environment. With protocol neutrality, the way in which clients communicate with a service provider is not important. Clients are not aware of what protocols are used or where the implementations reside. With implementation neutrality, the clients who use the FIPER services do not need to know what languages are used or how a service is implemented.

In all, GISO development tools provide (see Figure 2) accessibility through web-centric architecture; self-manageability using federated grids, scalability via network centricity, and adaptability with the power of mobile code inserted for execution through service providers.

Figure 2. GISO layered architecture.

3. FIPER Execution Environment

The peer-to-peer (P2P) service-oriented framework is proposed that targets multiparty grid transactions. A collection of all registered service providers (active and inactive) is called a service grid. A nested transaction is composed of a federation of providers that come together for completing a transaction. A transaction consists of a set of tasks with specific precedence relationships. The service providers do not have mutual associations prior to the transaction. They come together (federate) for a specific transaction. Each provider in the federation performs its services according to a job's control strategy that defines a transaction. Once the transaction is complete the federation dissolves and the providers disperse and seek other transactions to join. Different combinations of providers may come together for any given type of transaction at different times. The following characteristics define such transactions that are supported by the FIPER execution environment:

1. Multiple tasks/jobs need to be executed in order to complete the transaction
2. Service providers are interchangeable (i.e., any provider that implements the same interface for a service can be selected to perform the service)
3. Same service provider can perform multiple tasks in the transaction
4. Tasks in a transaction (federation) need to share data and resources with each other

The FIPER middleware is service-based in which a service is defined as an independent self-sustaining entity performing a specific task or job. Each service is defined by a public interface. The service grid is dynamic in which new services can enter the grid and existing services can leave the network at any instance. Services advertise themselves and can be found and selected based on the type (interface) and other attributes that they exhibit. FIPER defines all decentralized distributed components in the system to be equal. These components might be devices, repositories, processes or objects on the network. In the FIPER environment, peers are network objects of the same type. Each peer may implement multiple interfaces that are published when the peer joins the environment. All methods of these interfaces have the same format:

```
public ServiceContext operationName(ServiceContext)
```

Both arguments and return values of these methods are instances of type *ServiceContext* that represent service data. By its interface (type) and optional attributes (e.g., provider name), the network object can be dynamically found on the network without a host name and port required. These interfaces and their implementations might change, as they are specific to particular service providers. Thus peers should not expose their specific interface explicitly at the FIPER infrastructure level. The common, top-level peer interface called Servicer is defined as follows:

```
public interface Servicer {
// Put into action the specified exertion
public Exertion service(Exertion exertion)
throws RemoteException, ExertionException;
…
}
```

So, all peers implement this interface and their ***equality*** is defined as being service providers or servicers. A service is an act of requesting a service(Exertion) operation from a service provider as explained in Figure 4. The ***exertion*** is a distributed activity defined by the Exertion interface as follows:

```
public interface Exertion {
// Apply this exertion method to the specified context
public Exertion exert()
throws RemoteException, ExertionException;
…
}
```

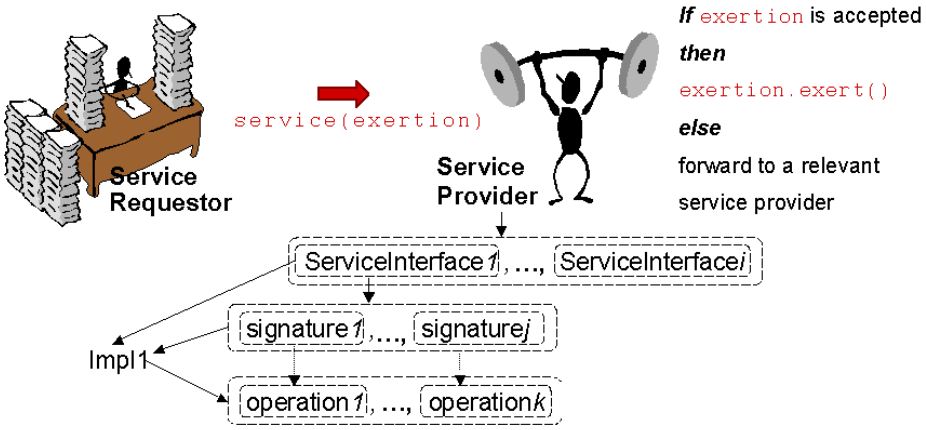

Figure 3. A service is an act of requesting a service(exertion) operation from a service provider. An *exertion* is accepted if its method matches one of provider's interface and one of the interface's methods.

In the FIPER environment two types of basic exertions are defined: ***tasks*** and ***jobs***. A task is the ***atomic exertion*** that is defined by its ***context model*** (data), and by its ***method***. An exertion method defines a service provider (grid object) to be bound to in runtime. This network object provides the business logic to be applied to the exertion context model. The computing framework based on concepts: context model, method and exertion is called for short the ***CME framework***.

A method is primarily defined by a ***provider type*** (interface) and ***selector*** (operation name) in the provider's interface. Optionally, additional attributes might be associated with the method, for example a provider's name or provider's identifier. The information included in the exertion method allows the GISO program to bind the exertion to the network object and process the exertion's context by one of its peer's operations, which is defined by its published interface. This type of service provider is called a ***method provider***. Another type of service provider is a ***context provider*** that provides shared data to the grid via the observer-observable paradigm. *Thus, both context and method providers represent grid data and operations to be used in the grid-oriented programs.* The FIPER middleware currently supports only method providers.

Service providers are equal since they define a common top-level interface, but the interface might be implemented differently by different groups of network objects. Rather than the currently prevalent client/server model, in which all communication passes through and is controlled by a central server (e.g., web, FTP, mail, and application servers), in service-toservice (S2S), the communication goes directly from one grid's object to another grid's object. Because accessing these decentralized object nodes means operating in an environment of unstable connectivity and unpredictable IP addresses, P2P nodes operate outside the domain name system (DNS) and have significant or total autonomy from central servers.

Sun's Jini™ Network Connection Technology [17-25] is used to implement the functionality specified above. The discovery and lookup protocols that Jini™ defines allow for a dynamic federation of services to be created.

A simplified UML-diagram showing the service-oriented framework of GISO is illustrated in Figure 4. The core of the framework consists of the four middleware services: ***Jobber*** (Service Broker), ***Cataloger*** (Service Catalog), ***Dropper*** (Exertion Spaces) and ***Provisioner***. The ***Jobber*** coordinates exertion execution within the GISO job. It interprets the job control context supplied by the end-user-client or any of service providers and coordinates the exertion execution accordingly as specified in the job.

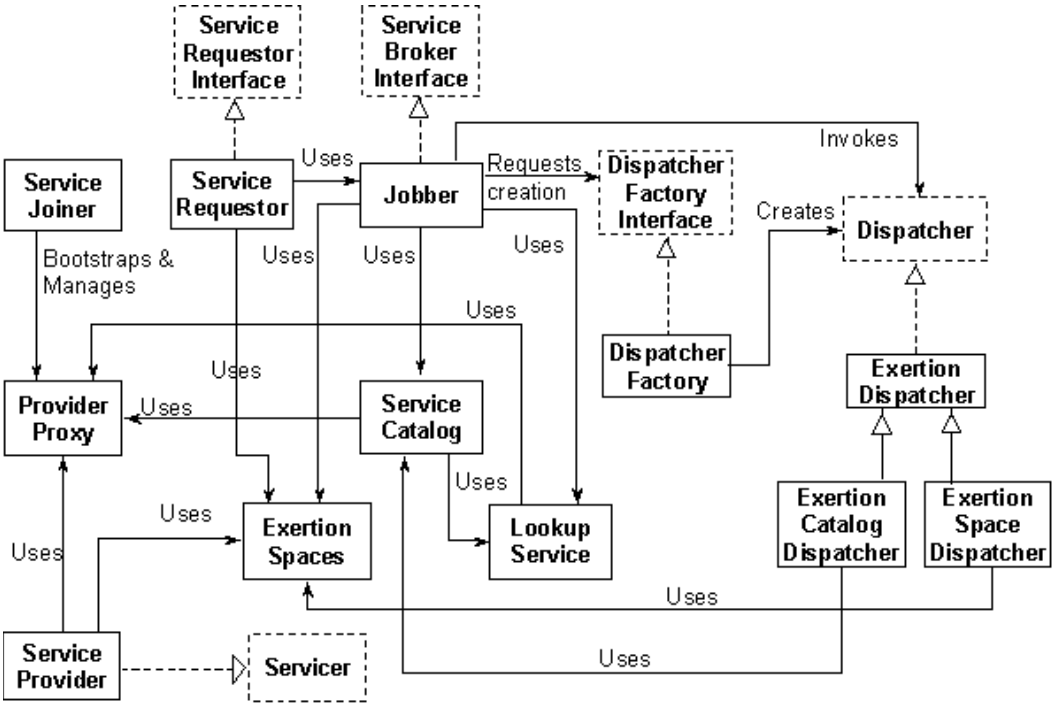

Figure 4. Job execution in FIPER.

There are two different ways in which the jobber can submit requests to the service providers. For an ***explicit access*** to the service provider the jobber can either use discovery to find a lookup service or use a service catalog (***Cataloger*** used by ExertionCatalogDispatcher) for selecting a service from dynamically formed peer groups. The lookup service caches all the proxies for services that have registered with it for a particular group(s) of services. A cataloger is a service-grid cache that periodically polls all relevant lookup services and maintains a cache of all the proxies that are registered with the lookup services for a particular peer group. The jobber has to discover lookup services each time it needs to use them where as it finds a cataloger then the cataloger continues service discovery for the jobber. In case the jobber finds an available service using a lookup registry or catalog, a proxy for the service is downloaded on to the jobber who invokes the service. Alternately the jobber submits the service request ***implicitly*** into an exertion space that is a shared GISO program repository for executing grid services. The exertion spaces provider, called ***Dropper*** (used by ExertionSpaceDispatcher), holds the request and waits for a relevant service provider to acquire the request from the exertion space. This is essential so that the job does not have to abort due to non-availability of a service.

Central to the execution of the GISO job is the *ExertionDispatcher* that can dispatch exertions either by the lookup service, the cataloger, or the dropper. A factory pattern is used whereby the right dispatcher is called based on the number of exertions, execution sequence (sequential or parallel) of the services and access type of services (*Cataloger* or *Dropper*). The framework supports autonomic provisioning whereby the jobber can provision or activate services on demand using a provisioning provider (***Provisioner***). Also GISO defines a common service provider interface and a set of utilities to create and register service providers with the GISO as service peers. A GISO service joiner is used for static bootstrapping service providers and also for maintaining leases on registered proxies.

The GISO programming environment consists of basic FIPER middleware providers: Jobber, Dropper, Cataloger, Provisioner, and Persister (as listed in the GISO functional architecture in Figure 5). Domain specific providers used by GISO programs (jobs) can be developed using GISO development tools (see Figure 2). All layers depend on the FIPER CME (Context, Method, Exertion) framework. Web clients and stand-alone requestors submit jobs to be executed by FIPER middleware services (infrastructure providers) in concert with domain specific providers that complement GISO programming.

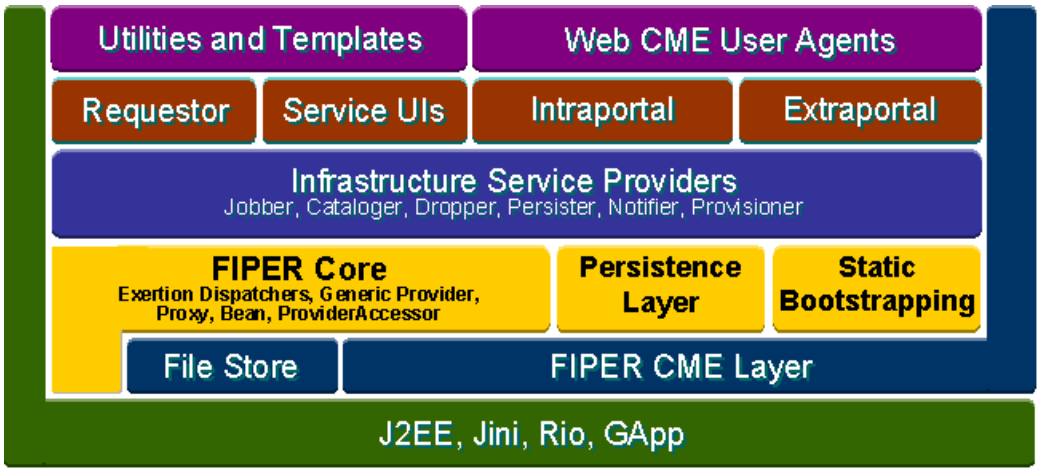

Figure 5. The GISO functional architecture.

In the case of web clients, exertions also can be submitted to a GISO interportal or extraportal. The extraportal is used for HTTP-based B2B secure communication over security firewalls.

4. GISO Programming and Development Tools

As defined earlier a nested transaction is carried out by a federation of providers that come together for completing a transaction. When performing a nested transaction, be it either a banking transaction or an engineering analysis, there are three basic components that can be identified. These are; the *process* or series of steps that must be executed to complete the transaction, a specification of the *action* to be taken in each step of the process, and the *information/data* associated with each step in the process (both input and output). Within FIPER the program objects that represent the components of a nested transaction are FiperExertion (FiperJob and FiperTask), FiperMethod, and FiperContext. That is the components of FIPER CME framework. The basic work unit or building block within the FIPER programming environment is an exertion. Each exertion (task or job) contains a FiperMethod and a FiperContext object. The FiperMethod specifies what *action*, or possibly the actual action in the case of mobile code, that is to be taken in a given step in the process. The FiperContext contains all the *data/information* the FiperMetod operates on or generates. The FiperContext also holds attributes for the data much like MIME types that identify the application(s) the data is associated with, its format (text, binary etc.), and other user defined modifiers. A FiperJob defines the *process*. A FiperJob consists of one or more exertions, the execution strategy for the process (sequential, parallel, looping and conditionals), and the mapping/relationship of data between exertions. The hierarchy of these classes is shown in Figure 6. It is worth nothing that recursion of FiperJobs is supported. That is any of the FiperTasks within a FiperJob can be a FiperJob itself.

The relationship between the FIPER program objects and the general description of a nested transaction is as follows; a FiperJob represents the process, the *FiperMethod represents the action*, and a FiperContext represents the data/information. The FiperTask acts as a container holding the FiperMethod and FiperContext creating the basic unit of work that is passed between various service providers.

As an example of a nested transaction in the FIPER Environment consider the following engineering application, the mechanical analysis of a gas turbine component. The component, a turbine blade is shown in figure 7. The *process* of performing a mechanical analysis consists of the following *actions*; generate solid geometry, descretize the geometry into a finite element model (FEM), apply boundary conditions to FEM, apply materials to FEM, solve the FEM for structural stresses. The necessary input data for each action and the resulting output data are shown in Figure 8. Also depicted in Figure 8 is the association between the three components of a nested transaction and the FIPER program objects.

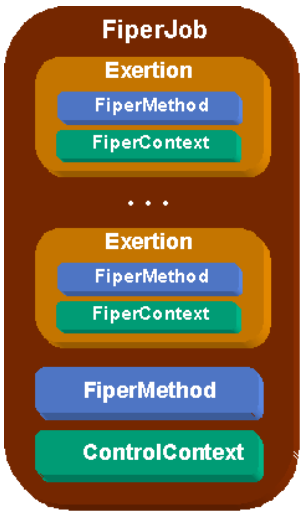

Figure 6. Fiper Program Object Hierarchy

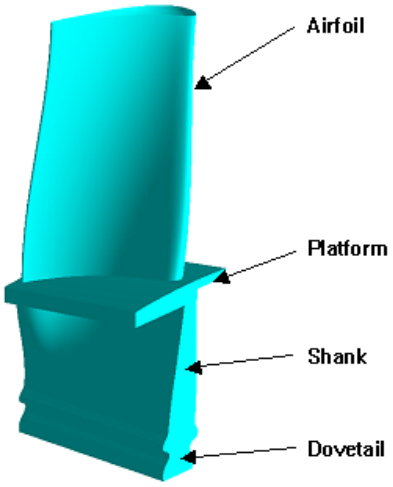

Figure 7. Turbine Blade Solid Geometry

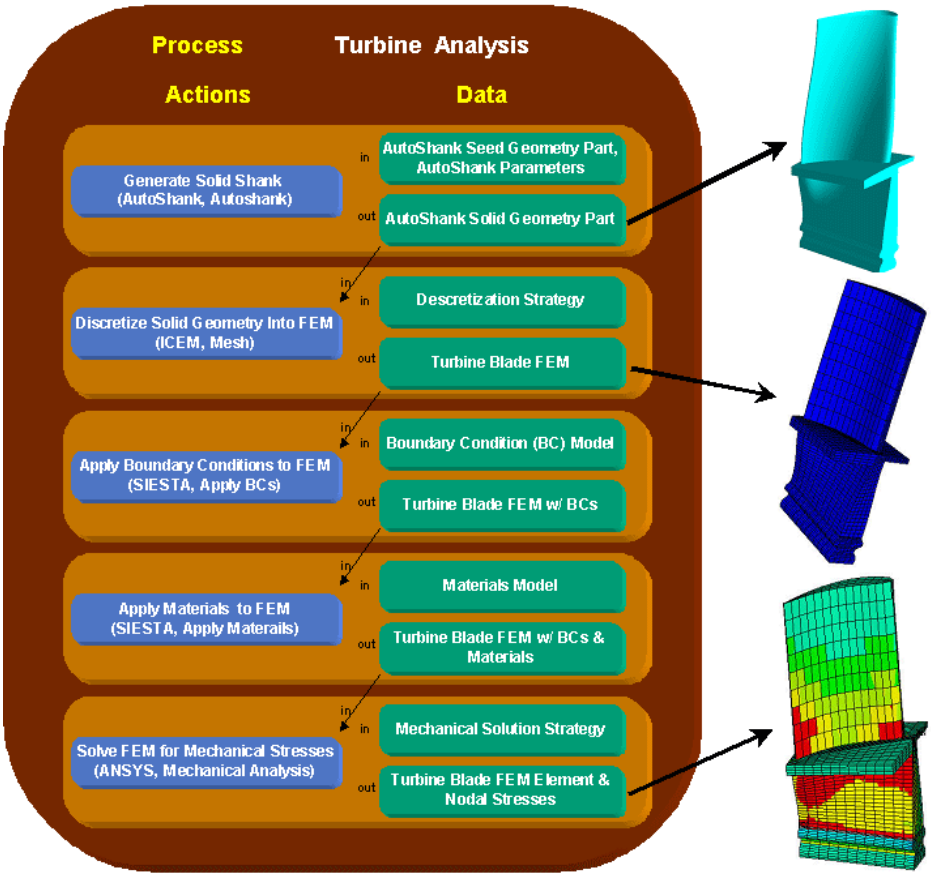

Figure 8. Process for the Mechanical Analysis of a Turbine Blade

To create the necessary program objects (FiperContext, FiperMethod, FiperTask, and FiperJob) for a nested transaction in the FIPER environment a collection of web browser user agents has been developed. It is not necessary to use these user agents for the development and execution of a FiperJob. Any standalone application can perform programmatically the same steps to create the necessary objects and act as a service requestor to submit the FiperJob for execution. Two of the goals of the FIPER project [9-14] are to allow access to the environment anywhere anytime (AWAT) and to evaluate if web browser technology can support large-scale engineering analysis

and design. Web browser technology clearly supports the AWAT requirement with the ability to access the environment from any device that can execute a web browser such as laptops, PDAs, and cell phones. This gives the end-user almost continuous access to the FIPER environment for creating, editing, submitting, and monitoring FiperJobs. The ability of web browser technology to support large-scale engineering has not proven to be so obvious. The performance of the web client and more importantly the robustness of web browsers raises serious questions to weather browser technology is mature enough to support large-scale engineering analysis and design. With the current implementation, which is based on JDK1.1 due to the requirement that it should run on many devices, many robustness issues have been encountered. For example, when the application is launched all the necessary classes will not download properly requiring the termination of the browser and a restart. Also, the browser will periodically freeze or lock for no apparent reason, once again requiring the restarting of the browser. Engineers will not tolerate these problems when performing analysis and design tasks.

The following sections illustrate the usage of the web user agents to create and execute the necessary FIPER program objects to perform the mechanical analysis of the turbine blade. Figure 9 shows the Fiper launcher page once logged into the Fiper environment. Here it can be seen that there are separate selections for the above described program objects, FiperContext, FiperTask, and FiperJob. The FiperMethod object is created within the FiperTask menu selection.

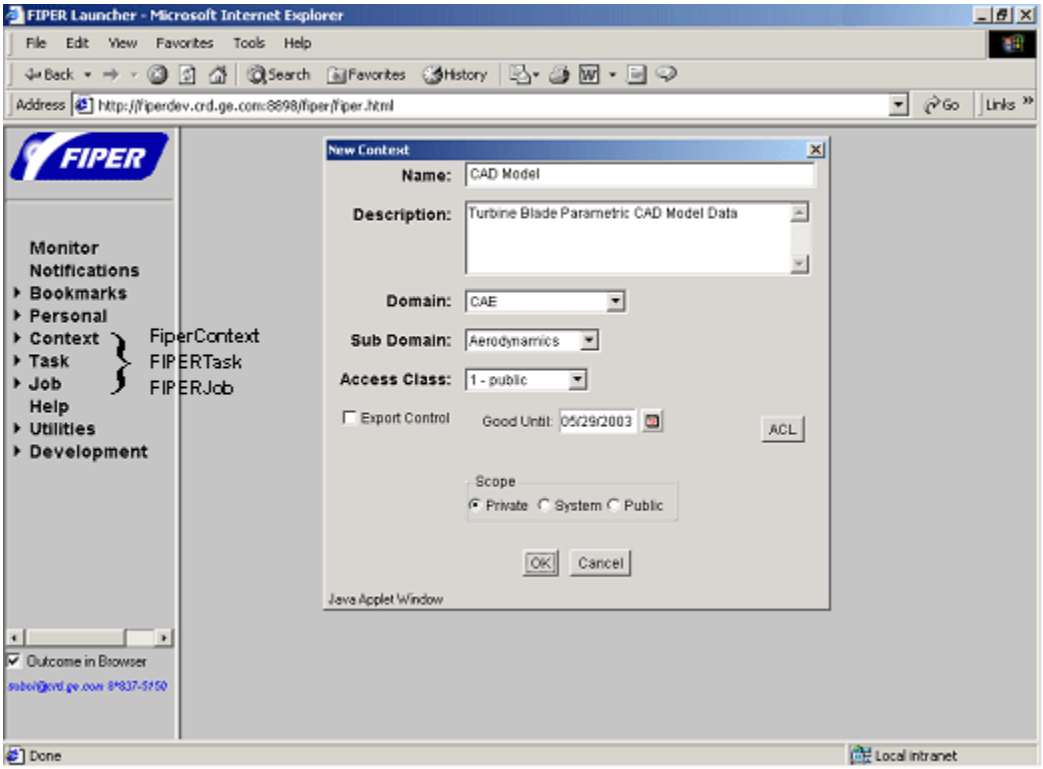

Figure 9. Fiper Launcher and New Context Dialogue

4.1 Context Editor

The Context Editor allows the end-user to specify the data or references to the data along with attributes associated with the data. When creating a new context the end-user is presented with the dialog presented in Figure 9.

The Name and Description fields are user defined, the Domain and Subdomain are selected from a drop down menu. At this point Domain and Sub Domain are used solely for sorting and searching purposes. The Access field is a company internal access classification and the Export Control box indicates if the data is export controlled. The ACL button produces an Access Control List (ACL) dialogue that allows the end-user to assign read, write, and

execute permissions on this program object based on userid or role. The scope indicates weather the context is publicly available (anyone who has access to the FIPER Environment can view and read), private (only the owner can see the object) or available to only the system administrator. Once the end-user completes the New Context Dialogue and selects OK the Context Editor then appears. Figure 10 shows the Context Editor along with the context for the first *action* or task in the Turbine Mechanical Analysis Job represented in Figure 8.

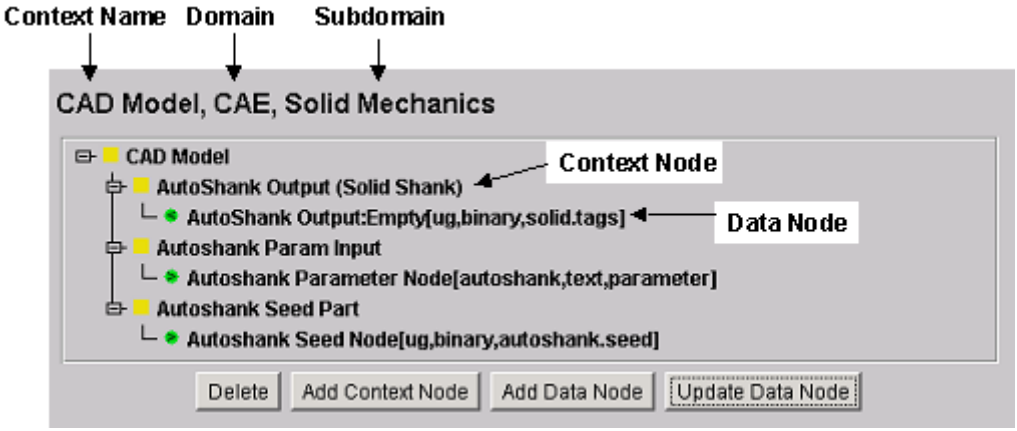

Figure 10. Context Editor

Figure 10 also illustrates that the FiperContext is a tree structure with Context Nodes and Data Nodes. The Data Nodes are further identified as either input ">" or output "<". The editor allows the end-user the ability to create, edit, or delete Context Nodes and Data Nodes in the FiperContext. Specifically if the Autoshank Parameter Node is highlighted and "Update Data Node" is selected from the editor the dialog in Figure 11 appears.

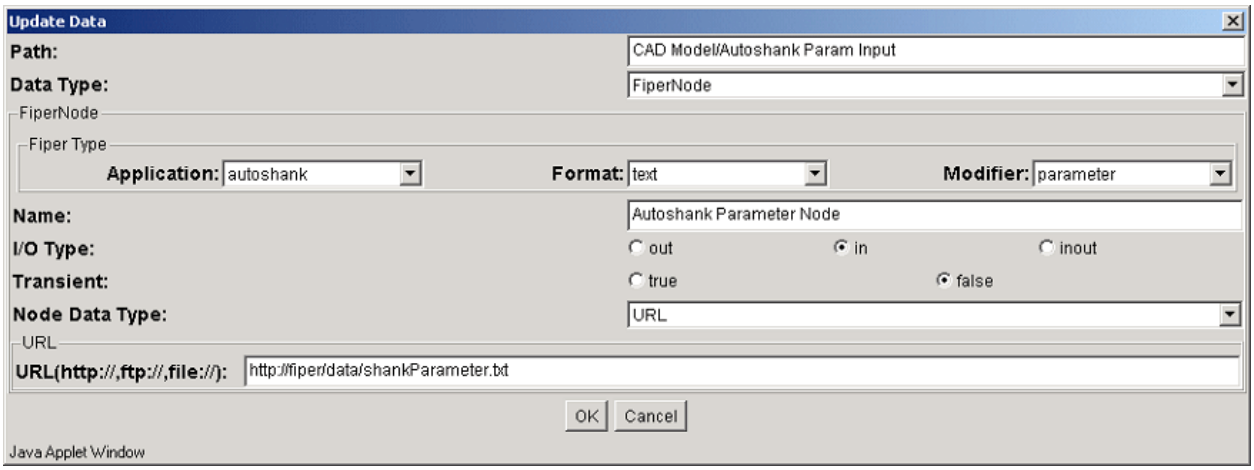

Figure 11. Update Data Node Dialog

This dialog shows the pertinent information associated with this Context Data Node. In this dialog the Data Type of the object that resides in the Context Data Node is seen to be of type FiperNode. The contents of a Context Data Node can be any Java primitive data type or object. For an object type of FiperNode it is further noticed that a Fiper Type can be associated with the FiperNode. Here three fields represent the Fiper Type: Application, Format, and Modifier. These are attributes describing the data that resides within or is referenced by the Node Data Type field in the Dialog. In this case the Node Data Type is URL. The URL is essentially a reference to a file. Based on the Fiper Type one can see that the file at the given URL is for the Application "autoshank," Format is "text", and has a Modifier of "parameter". The service, which receives this Context, will recognize this Fiper Type and take appropriate action based on this information.

Upon close examination of Figure 10 it can be seen that the Autoshank Output DataNode is Empty. That is it contains no data or reference at this time. When the autoshank provider finds this node at run time it recognizes that it must take the appropriate actions to generate and fill this empty DataNode. Once all the necessary FiperContexts are created the end-user can now create the FiperTasks.

4.2 FiperTask Editor

From the Fiper launcher in Figure 9 the end-user selects Task, New and completes the New Task Dialog (similar to the New Context Dialog in Figure 9) to gain access to the Task Editor shown in Figure 12. Figure 12 represents the creation of the first task (Generate Solid Shank) associated with the FiperJob illustrated in Figure 8.

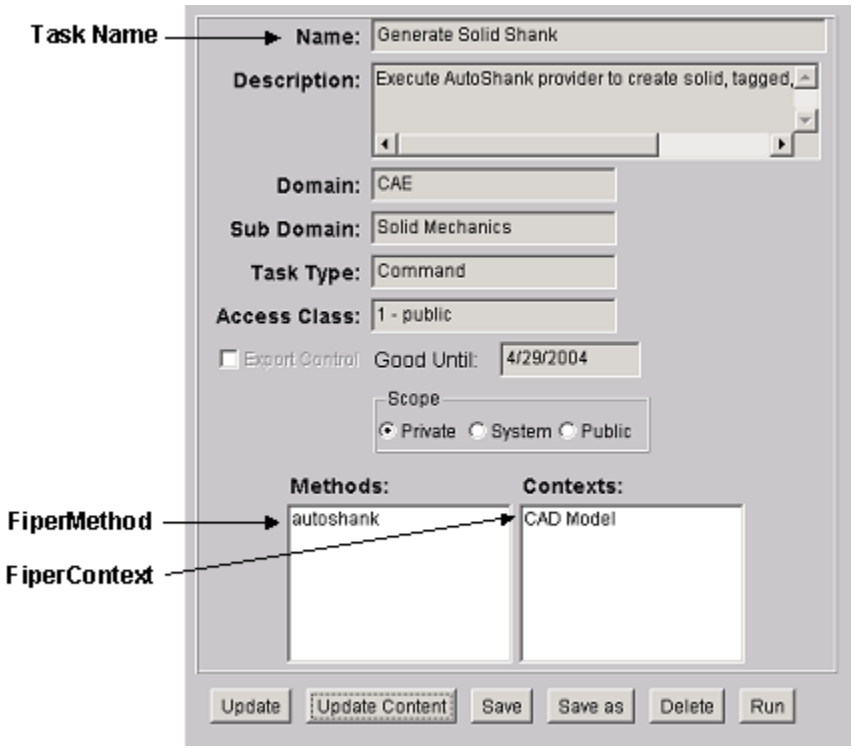

Figure 12. FiperTask Editor

Recalling that the FiperTask is the fundamental building block or work unit in the FIPER Environment which contains the *action* and *data* for a nested transaction (reference Figure 6), one can see that the Methods field represents the *action* and the Context field in the FiperTask Editor in Figure 12 represents the *data*. To view/edit more detail on theses fields the end user selects “Update Content” which produces an editor (see Figure 13).

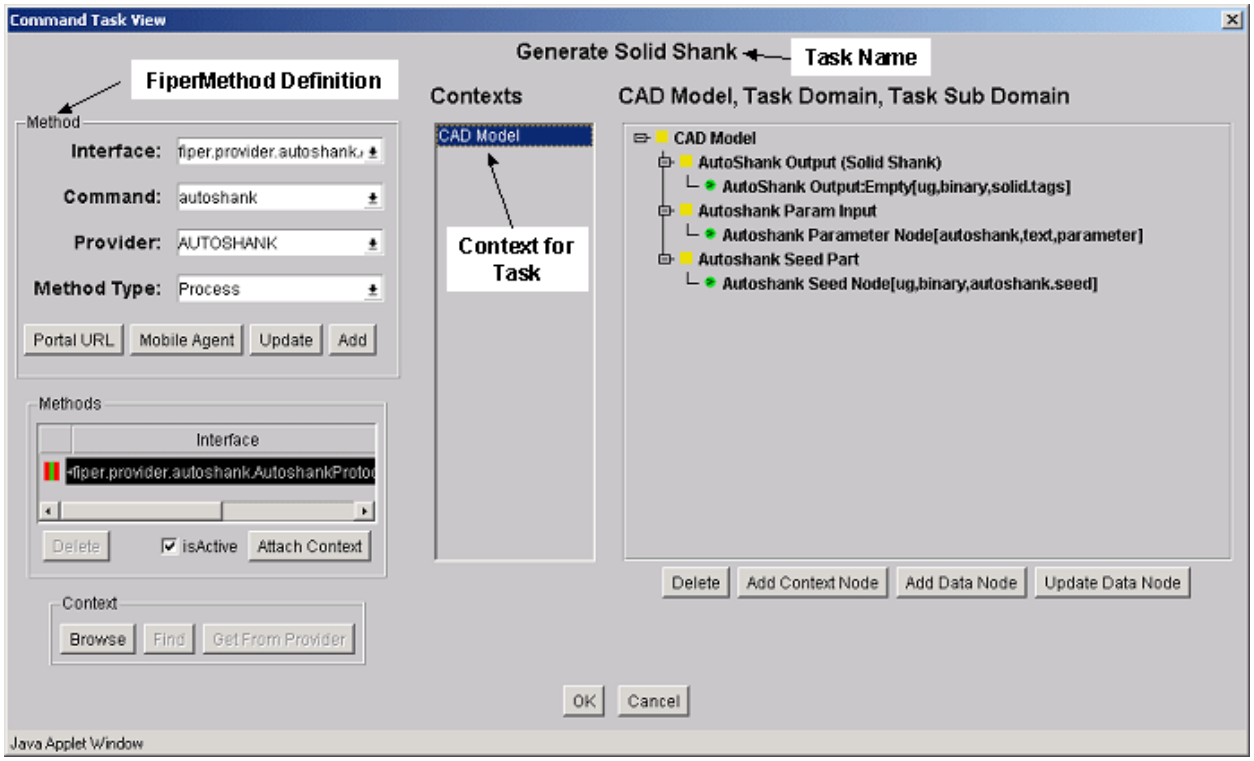

Figure 13. FiperTask, FiperMethod and FiperContext Editor

Figure 13 shows the definition of the FiperMethod and the Context that is used for the selected task, Generate Solid Shank. The fields Interface, Command, Provider, and Method Type define the Method. The interface is the Java Interface that is implemented by a given service provider, the Command is the java Method within that interface that will be called, and the Provider is the Name that the provider is published under. Method Type can be pre-process, process, or post-process. Currently only process is implemented for Method Type. The Interface and the Provider are used as the attributes to locate a service within the environment with the current implementation. This Method Definition should be replaced with a much more user-friendly way to search and select the available methods in the Fiper Environment. In addition, a Quality of Service (QoS) should be implemented as well to expand the number and types of attributes that are used to resolve or select a given service. The context for this task is the CAD Model Context presented in Figure 10. This is inserted into the task by browsing the contexts that the end-user has created or has access to. Once the context has been added to the task the end-user has the full context editor available to them to modify the context in the task. Once all the *actions*/FiperTasks have been defined for a given *process*/FiperJob the FiperJob itself can then be constructed.

4.3 FiperJob Editor

Once again from the Fiper launcher in Figure 9 the end-user selects Job, New and completes the New Job Dialog (similar to the New Context Dialog in Figure 10) to gain access to the Job Editor shown in Figure 14. Figure 14 illustrates the creation of the FiperJob represented in Figure 8. It contains all the tasks Generate Solid Shank, Mesh Shank, Apply Boundary Conditions, Apply Materials, and Perform Stress Analysis.

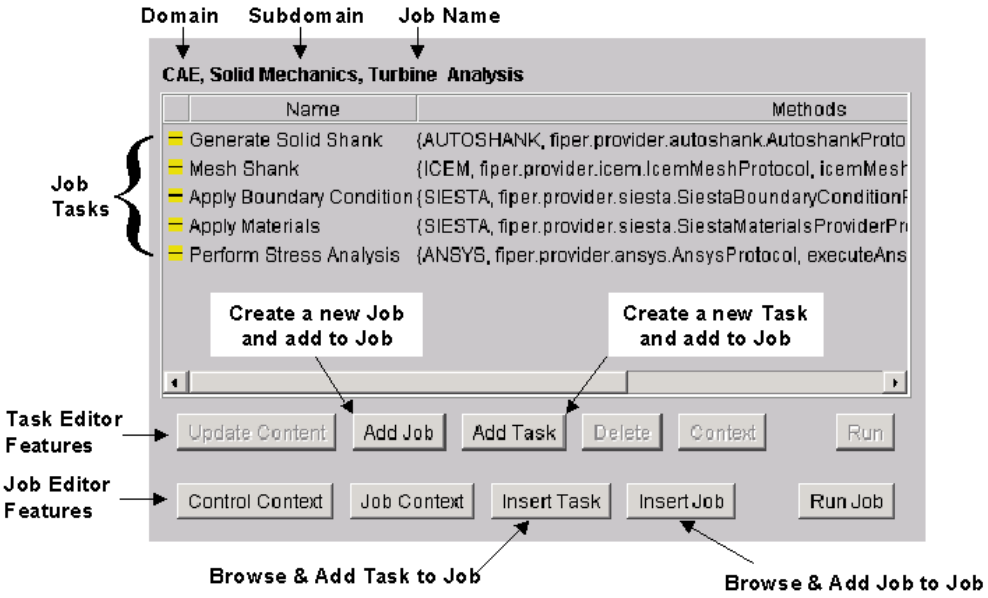

Figure 14. FiperJob Editor

The Job Editor lists all FiperTasks associated with the job along with the FiperTask's Name and FiperMethod Attribute information (Provider Name and requested provider's type - interface). The Task & Job Editor features allow the end user to add additional FiperTasks or FiperJobs by either browsing existing program objects or creating new objects on the fly. The Job Editor features also enable the specification of the Control Context and the JobContext. The ControlContext specifies the flow and method of execution of the FiperJob. Figure 15 shows the Control Context for the Turbine Analysis Job.

Figure 15. FiperJob Control Context Editor

The Control Context dialog is divided into two regions, Job Tasks and Job Settings. Under the Job Tasks region the end-user can specify if a task is executed or not and the order in which the tasks are executed (for sequential execution strategy). Currently the Control Context does not support the capability of looping and conditionals. But these are considered to be critical features in a production GISO computing environment. Finally, the end-user has the ability to specify that the job suspend during execution so that results can be reviewed before continuing. This is a very powerful and desirable feature that will be discussed in further detail in the Job Monitor section. Within the Job Settings region of the editor one can select the Task dispatching method (catalog, dropper, Jini Lookup), the Execution strategy (sequential or parallel), and the Job broker type.

The final step before a FiperJob can be executed is to define the flow of data between tasks in the job. This is done using the JobContext dialog, which can be invoked from the Job Editor features on the Job Editor Dialog in Figure 14.

The FiperJob Context dialog for the Turbine Analysis Job is shown in Figure 16. Here the Job is shown with each task and the context for each task in a hierarchical tree structure. The data flow from one task to the other is defined by dragging one Fiper DataNode onto another Fiper DataNode. In Figure 16 this has occurred by dragging the AutoShank Output Solid Shank Node contained in Task0 onto the Solid Shank unnamed Fiper DataNode in Task1.

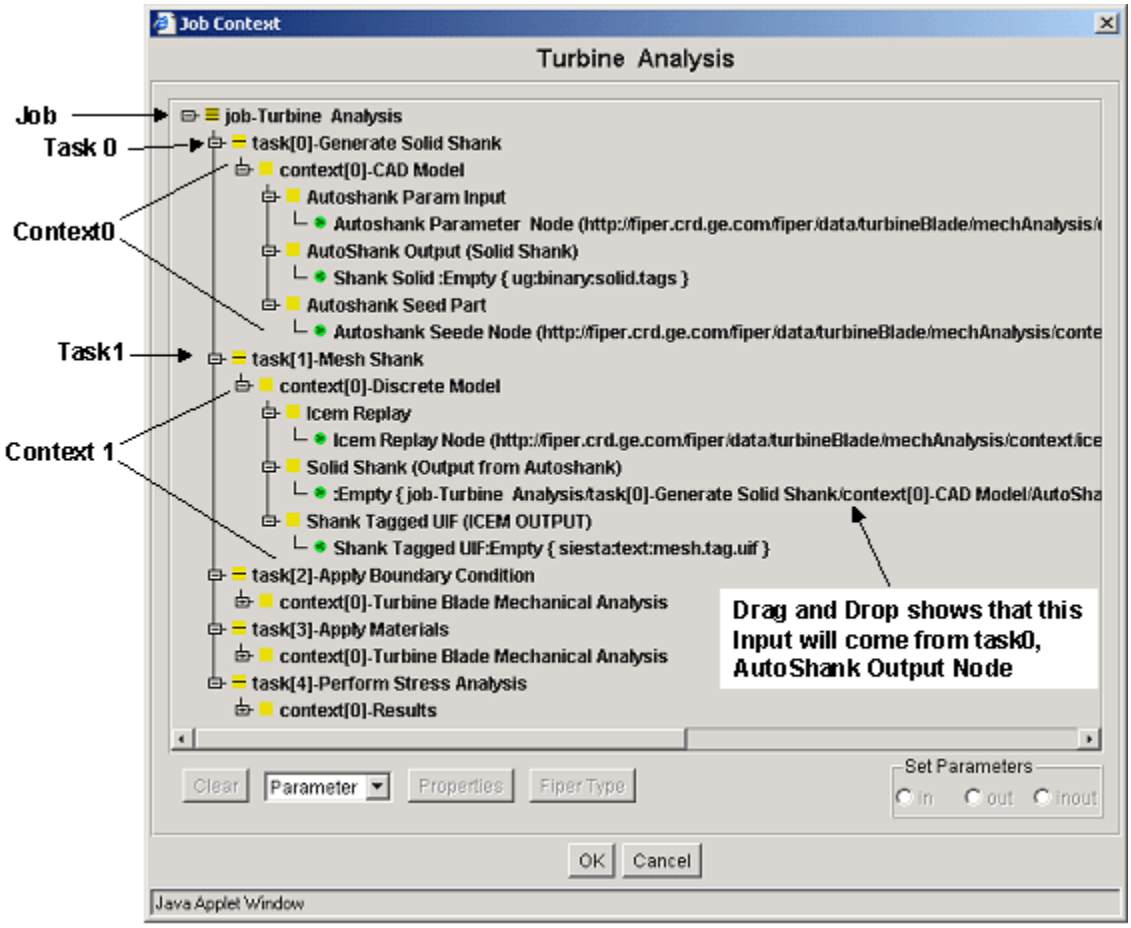

Figure 16. JobContext Dialog

Once the data flow has been defined in the JobContext the FiperJob is now ready for execution. To submit the job to the Fiper Environment the Run Job button is selected in the Job Editor (Figure 14). A typical engineering analysis or design job could take anywhere from a few hours up to several days or even weeks. With jobs running this long it is critical that the end-user have access to the status of the job and control over the job as it executes. This is the function of the Job Monitor. The Job Monitor can be accessed from the Fiper Launcher, Figure 9, by selecting the Monitor option.

4.4 Job Monitor

Probably the most critical capability that GISO programming will need from an end-users perspective is the ability to interact with the *process*/FiperJob once it has been submitted to the environment. Using a GISO IDE will require a cultural change within the end-user community. Today's state of practice is that typical designers and analysts execute single standalone applications either on their desktop or submit the runs to a major shared resource (MSR) computing environment. In either case the end-user is executing applications individually and if a failure occurs they know at least which application the failure occurred within. Also, when running locally or in a MSR the end-user usually has some or all control over the running application and can closely monitor the progress of the execution by monitoring log files and or output files from the application. In the GISO IDE the end-user is now combining many application to perform a nested transaction and submitting the execution of the nested transaction to the network, which could easily take days or weeks to complete. In the GISO IDE the end-user may have no idea where the execution is taking place and worse will have no feedback as to the state of progress of the process. In the

GISO IDE the end-user surrenders all control to the environment, a precarious proposition for a designer who is accustomed to having complete control of the applications they are running. With these facts in mind a few essential functionalities are identified for GISO programming that are necessary for end-user to accept such a working environment. The end-user must be able to monitor the progress of the process and obtain intermediate results from a given task. The end-user must be able to control the process once it is submitted to the environment by stopping, suspending, or terminating the process. For a suspended GISO program the end-user must be able to edit not only the data within the process but also the process itself by adding or deleting tasks. After any edits to the data or process the end-user must be able to resume the process from any task within the process not necessarily the task the process was suspended at. If the process fails the end-user must obtain meaningful information that specifies where the failure occurred and what action needs to be taken to correct the problem. This last requirement puts a significant burden on the service provider developers to properly trap exceptions and translate them into meaningful information for the end-user.

As an example of the needs for end-user interaction with a running process consider the Turbine Analysis Job presented in Figure 8. The first task is to generate the geometric solid shank for the turbine blade with a new set of shank parameters. The second task is to discretize the shank with a given meshing strategy. It is not known if the meshing strategy will be adequate with the parametric changes made to the shank. Thus, when the mesh task is taking place the end-user needs to obtain intermediate information from the meshing application to evaluate if the quality of the mesh is satisfactory in terms of aspect ratios of the finite elements. If not, the task will need to be terminated and the process suspended. The end-user makes the necessary edits to the meshing strategy and the process resumed from the meshing task. There is no need to re-run the generate solid shank task. In this case this saves only a few minutes of run time but not re-running satisfactorily completed tasks could save hours or days. Once the process is resumed the end-user soon decides that they would like to review the entire FEM before performing the stress analysis. The end-user must be able to select a task in the process and state that the process should suspend after execution of the task. That is suspending after the Apply Materials task in the current example. Once a review of the FEM is completed the process is resumed and the stress analysis initiated. If the stresses are requested for several loading conditions the client would like to review the stresses for each load case as they are completed. Again, the need for intermediate information from a task is desired. As the client reviews this information it is determined that due to the parametric changes in the shank geometry the location of the critical stresses have moved and the mesh density in that region is not sufficient to properly resolve the stresses. At this point the user terminates the task, and once again edits the meshing strategy to create a denser mesh in the new critical stress region. Once the edits are complete the user now resumes execution, not from where the job was suspended(Perform Stress Analysis) but, from the meshing task. Once the job completes the client reviews the stress results obtained and recognizes that the stresses are not in the proper format. The user desires to add an additional task to the job after the stress analysis. This is not an edit to the data but now an edit to the process. The client adds a post-processing task after the stress analysis task. The output from the stress analysis is passed as input to the post-processing task. The job is then resumed starting with the prost-processing task. Although this is a simple example with only five tasks it demonstrates the critical need for end-user control and interaction of the executing process/FiperJob. In a large scale multi-disciplinary analysis and design applications the process could easily have one hundred tasks and could take weeks to execute. In such a scenario the user control and interaction is the only way that GISO programming could be successfully employed.

In the FIPER Environment the monitoring/client process interaction is done using the Job Monitor. The Job Monitor is accessed from the Fiper Launcher page as the first menu pick. Figure 17 shows the Turbine Analysis Job running in the Job Monitor. The Job Monitor can be viewed as an “interactive debugger for program objects or services on the network”. The Job Monitor shows the progress of the process (green complete, green/yellow running, red failed, yellow suspended). It also displays intermediate information from a task (by viewing the job context) if the provider returns such information. The client is also able to stop, suspend, step and resume a running job. In addition, for a given suspended or completed job, the client has access to a drop down menu that allows full edit capability of the data in the job or the job/process itself. Data can be changed, tasks can be edited/added/deleted and the job resumed from any task. The only functionality not supported by the job monitor in the current implementation is the suspension or termination within a task. Currently suspension/termination can occur only between tasks.

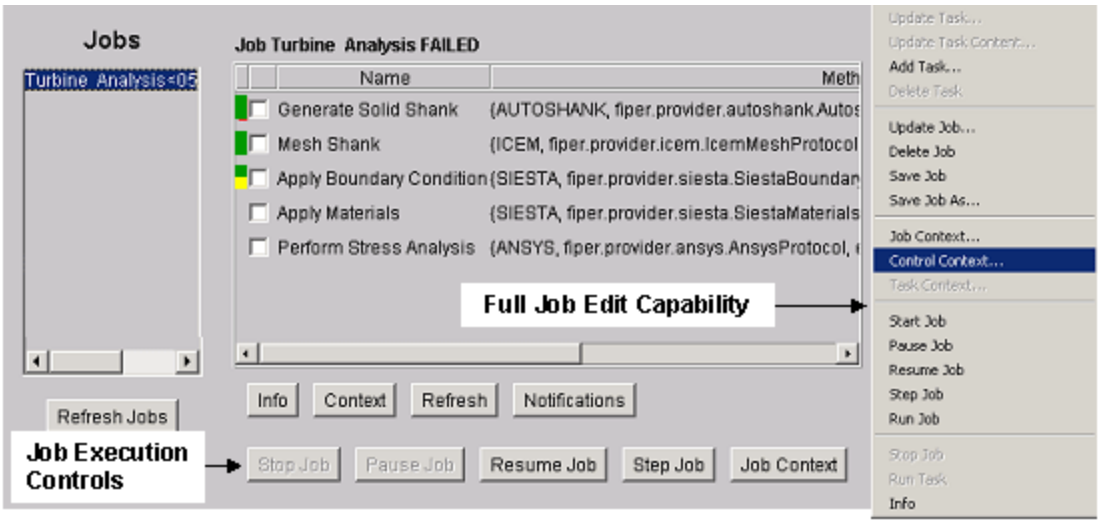

Figure 19. FiperJob Monitor

Conclusions

In the GISO approach object-oriented concepts are applied to the network and grid-oriented programs. A job (a collection of exertions) is a service-oriented program executed in a federated service-oriented environment across multiple virtual organizations. Jobs are crated using friendly, interactive web-based graphical interfaces. Jini connection technology from Sun Microsystems enables federated, platform independent, real world grids. It allows us to create GISO programs that process a whole aircraft engine as a virtual object-oriented product control structure that can be manipulated by multidisciplinary teams as network-centric, active, evolving product. New shared programs and engineering applications can be assembled as needed on the fly by integrating new capabilities into existing workflows, systems, devices and applications. The presented web-centric GISO IDE reduces the costs of solving business problems as well as of establishing and maintaining online business relationships. Services are provided by shared low cost, easy to develop service providers and are easily integrated into the core business of an enterprise. An experimental version of presented approach was successfully deployed at GE Aircraft Engines. The further extensions of the presented approach, including distributed observer/observable model [26] for context (data) provider, algorithmic and logic programming control contexts, integration with JXTA [28], and web services are investigated in the SORCER Laboratory, Texas Tech University [16] to be tested and deployed on Texas Tech University TechGrid [15].

References

1. J. Postel, C. Sunshine, D. Cohen, “The ARPA Internet Protocol,” Computer Networks 5:261-271 (1981).

2. J. Postel and J. Reynolds, “Request for Comments Reference Guide (RFC1000),” Internet Engineering Task Force (1987).

3. D. Lynch and M.T. Rose, eds., “Internet System handbook,” Addison-Wesley, reading, MA (1992).

4. See for example J. Lee, ed., “Time-Sharing and Interactive Computing at MIT,” IEEE Annals of the History of Computing 14:1 (1992); K. Hafner and M. Lyon, “Where Wizards Stay Up Late,” (a history of Internet development), Simon and Schuster, New York (1996).

5. I. Foster and C. Kesselman, eds., “The Grid: Blueprint for a New Computing Infrastructure,” Morgan Kaufmann Publishers, San Francisco CA (1999).

6. A. S. Grimshaw, W. A. Wulf, et al., “The Legion vision of a worldwide virtual computer”, Communications of the ACM, 40(1), 39-45, 1997.

7. L. Smarr, “Computational infrastructure: Toward the 21st century,” Special issue on plans for a National Technology Grid, Communications of the ACM 40, 11 (1997)

8. National Research Council reports relevant to early grid research include the following: “National Collaboratories: Applying Information Technology for Scientific Research,” National Academy Press, Washington D.C. (1993); “Evolving the High Performance Computing and Communications Initiative to Support the Nation’s Information Infrastructure,” National Academy Press, Washington D.C. (1995); and “More Than Screen Deep: Toward Every-Citizen Interfaces to the Nation’s Information Infrastructure,” National Academy Press, Washington D.C. (1997).

9. Sobolewski,. Federated P2P Services in CE Environments, Advances in Concurrent Engineering, A.A. Balkema Publishers, 2002, ISBN 90 5809 502 9, pp. 13-22, keynote paper (2002).

10. Michael Lapinski and Michael Sobolewski, “Managing Notifications in a Federated S2S Environment,” International Journal of Concurrent Engineering: Research & Applications, December, 2002.

11. Sobolewski, 2002. FIPER: The Federated S2S Environment, JavaOne, Sun’s 2002 Worldwide Java Developer Conference, San Francisco (2002),
(http://servlet.java.sun.com/javaone/sf2002/conf/sessions/display-2420.en.jsp)

12. Zhao, Shuo, and Michael Sobolewski, 2001, Context Model Sharing in the FIPER Environment, Proc. of the 8th Int. Conference on Concurrent Engineering: Research and Applications, Anaheim, CA.

13. Michael Lapinski, and Michael Sobolewski, August 2001, Notification Manager in the FIPER Environment, Proc. of the 8th Int. Conference on Concurrent Engineering: Research and Applications, Anaheim, CA.

14. Peter J. Röhl, Raymond M. Kolonay, Rohinton K. Irani, Michael Sobolewski, Kevin Kao, A Federated Intelligent Product Environment, AIAA-2000-4902, 8th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, Long Beach, CA, September 6-8, 2000.

15. Texas Tech University TechGrid Web site, http://www.hpcc.ttu.edu

16. SORCER Web site, http://www.sorcer.cs.ttu.edu

17. Jini.org Web site, http://www.jini.org/

18. Jini Architecture Specification. [online] 2000 [cited 2002 March 22]. Available at URL: http://www.sun.com/jini/specs/jini1_1.pdf.

19. Edwards, W.K. (2000). Core Jini, 2nd ed., Prentice Hall, ISBN: 0-13-089408.

20. Oaks, S. & Wong, H. (2000). Jini in a Nutshell, O'Reilly, ISBN: 1-56592-759-1.

21. Newmarch, J., "A Programmers Guide to Jini Technology", Apress, Berkeley, California, ISBN: 1893115801, 2000.

22. Freeman, E., Hopfer, S., & Arnold, K.(1999), JavaSpaces™ Principles, Patterns, and Practice, Addison-Wesley, ISBN: 0-201-30955-6.

23. Halter Steven L., JavaSpaces Example-by-Example, Prentice Hall; ISBN: 0130619167.

24. Davis Project. http://davis.jini.org/

25. Rio Web site, http://rio.jini.org/

26. Grand, M. (1999). Patterns in Java, Volume 1, Wiley, ISBN: 0-471-25841-5

27. IBM Autonomic Computing Web site, http://www.research.ibm.com/autonomic

28. Sing Li, JXTA Peer-toPeer Computing with Java, Wrox Press Ltd., ISBN 1-861006-35-7, 2001.

29. D. Brookshier, D, Govoni, N. Krishnan, JXTA: Java P2P Programming, SAMS, ISBN 0-672-32366-4, 2002.

30. The Globus Project Web site, http://www.globus.org

31. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. I. Foster, C. Kesselman, J. Nick, S. Tuecke, Open Grid Service Infrastructure WG, Global Grid Forum, June 22, 2002. (extended version of Grid Services for Distributed System Integration).

32. Grid Service Specification. S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman; Open Grid Service Infrastructure WG, Global Grid Forum, Draft 2, 7/17/2002.

33. Grid Services for Distributed System Integration. I. Foster, C. Kesselman, J. Nick, S. Tuecke. Computer, 35(6), 2002.

34. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. I. Foster, C. Kesselman, S. Tuecke. International J. Supercomputer Applications, 15(3), 2001. Defines Grid computing and the associated research field, proposes a Grid architecture, and discusses the relationships between Grid technologies and other contemporary technologies.

35. The Grid: A New Infrastructure for 21st Century Science. I. Foster. Physics Today, 55(2):42-47, 2002.