A Component Based Services Architecture for Building Distributed Applications

Randall Bramley, Kenneth Chiu, Shridhar Diwan, Dennis Gannon,
Madhusudhan Govindaraju, Nirmal Mukhi, Benjamin Temko, Madhuri Yechuri
Department of Computer Science
Indiana University - Bloomington, IN

Abstract— **This paper describes an approach to building a distributed software component system for scientific and engineering applications that is based on representing GRID services as application-level software components. These GRID services provide tools such as registry and directory services, event services, and remote component creation. While a services-based architecture for Grids and other distributed systems is not new, this framework provides several unique features. First, the public interfaces to each software component are described as XML documents. This allows many adaptors and user interfaces to be generated from the specification dynamically. Second, this system is designed to exploit the resources of existing Grid infrastructures like Globus[7], [15], Legion[17], [7], and commercial Internet frameworks like e-speak[11]. Third, and most important, the component-based design extends throughout the system. Hence tools such as application builders which allow users to select components, start them on remote resources, and connect and execute them, are also interchangeable software components. Consequently, it is possible to build distributed applications using a graphical "drag-and-drop " interface, a web-based interface, a scripting language like Python, or an existing tool such as Matlab.**

I. Introduction

Software component architectures have emerged as a standard design paradigm in many areas of application development. Java Beans [22], [5] is a component standard for building Java based desktop applications. COM [19], [4] is Microsoft's ubiquitous component model that is central to their application interoperability. And in July, 1999 the Object Management Group approved the specification of the CORBA component model (CCM) [27] which extends Enterprise Java Beans [23] for applications in the e-commerce sector. In all cases, the goal of a component architecture is to simplify the application design process and to speed application development. This shifts the traditional focus of scientific and engineering computing infrastructure from scaling the performance of individual applications to exploring new ways to increase programmer productivity.

Several component systems designed for wide-area, scientific computation have been built and are now in use. SciRun from Utah [28], [26] provides a powerful extension to scalable, parallel computation and visualization of a component paradigm used with AVS and IRIS Explorer [10]. Webflow from Syracuse [7], [25] uses CORBA as part of a multi-tiered architecture for a scientific problem solving workbench. High performance implementations of CORBA exist and powerful toolkits are also available [29], [1], [8]. Recently, representatives from several DOE national laboratories, the University of Utah, Indiana University and NCSA have released a preliminary specification for a component model for distributed and parallel scientific computing called the Common Component Architecture (CCA). Over its two-year development, the CCA [2], [16] has closely followed the evolving design of the OMG CCM and so both share many design features. There are also significant differences. The current version of CCA is specified from the perspective of the required behavior of software components. There is no detailed specification of the component framework or containment mechanisms. In particular, the CCA specification only provides standard mechanisms for software components to name and export their public interfaces. It does not provide a standard model for how component instances are discovered, created and connected. In addition, it does not identify a set of standard services that are made available by the component framework to each component instance. The reason is that CCA components may run on massively parallel supercomputers or over wide area networks, and any of several framework services and component containment policies may be appropriate.

As an experiment, we have designed and built an implementation of the CCA specification that runs over Globus [7], [15] based wide "grids" such as NASA IPG [13]. The purpose of the experiment is to test one approach to the design of a service model for the component framework. As the OMG CCM is also largely untested, this experiment also validates several aspects of that design. We have three working hypotheses:

- The existing CCA architecture is significantly rich enough that the entire framework and

all of the services, containers and component "builders" can themselves be built as components.
- With this "federation of components" design of the basic framework it should be possible to layer this architecture on top of several existing distributed systems including Globus [15], Sun's Jini [24], HP's e-speak [11], OMG CCM [27] and Legion [17]
- The resulting component framework achieves the goal of providing an efficient, practical foundation for the design of scientific problem solving environments for grid computation, as has been proposed by the Computing Portals [9] project of the Java Grande effort.

This paper only addresses the first hypothesis and a single instance of the second. In the months ahead we will research layering the system on top of other systems. As to the third hypothesis, we are currently working with two application groups to design Science Portals based on our implementation. One group is part of the DOE NGI program working on X-ray crystallography and the other group is the NCSA chemical engineering team working on a computational workbench for the simulation of the chip fabrication processes for copper deposition.

The primary contributions of this paper can be summarized as follows:
- We illustrate how XML can be used as powerful interface design language. XML allows greater flexibility in the content of a specification than does a static CORBA-style IDL solution.
- We demonstrate how the CCA component specification can be used to build a service architecture in which the framework and grid services act as and are treated identically to other software components.
- We briefly describe the way in which four different "application builder" components are constructed.

II. A Brief Introduction to the CCA Specification

The Common Component Architecture consists of two type of entities: components and frameworks. Components are the basic units of software that are composed together to form applications. Instances of components are created and managed within a framework which also provides the basic services that components use to operate and communicate with other components. A component might be a computational "engine", such as an equation solver or a graphics package, or it might be an encapsulated application such a large SPMD simulation or a database system. It could also be a desktop application such as Matlab, or the Python interpreter, or some other more abstract component framework service.

The philosophy of CCA is to precisely define the rules for constructing components (or, in the case of existing applications, the software wrapping that makes them into components) and the specification of the required behavior that a component must exhibit for it to coexist with other components within a CCA framework. A CCA framework is a software environment that allows components to be dynamically instantiated, coupled together, and have methods invoked on them.

Currently, the CCA does not specify how the framework is constructed or how the user interacts with the framework to connect components together. This allows many different frameworks that can be used in different situations. Some frameworks will be designed to optimize the use of components that are distributed across a wide-area Grid. Others will be designed to optimize the composition of components that run on a single, massively parallel supercomputer.

A. Ports

The *port* is a fundamental CCA concept. A port is a public communication interface of a component. A port can be either a *Provides-port* or a *Uses-port.*

A *Provides-port* is an interface of functions that the component implements. A component can have zero or more Provides-Ports. A Provides-port can also be thought of as functionality that is "provided" to other components or to the framework. The member functions of a Provides-port interface may be thought of as "handler" functions that are executed by the component on behalf of the component's "users".

Provides-ports may be connected to Uses-ports. A Uses-port can be viewed as a connection point on the surface of the component where the framework can connect references to Provides-ports supplied by other components or the framework. Viewed from the inside the component, a Uses-port is an object that implements functionality the component requires. The component makes calls on a Uses-port reference to "use" the "provided" functionality. A component may have zero or more Uses-ports.

Depending on the framework, one or more Provides-ports may be connected to a single Uses-port and a Provides-port may be "provided" to one or more Uses-ports. In general, if a port interface has a member function that returns a value, the number of providers will be restricted to be one. In these cases, there must be a connected provider for the component to operate correctly.

In the case where two components exist in the same address space, a "direct" connection between a Provides-port and a Uses-port can be made. This means that the object that is "used" is the same as the one "provided". In this case there is only one function call between a user and a provider. In the case of two components that are in different address spaces, the Uses-port is a container that holds proxies to the remote Provides-ports that it is connected to.

CCA components may be written in Java, Fortran, C or C++. It is up to the framework to ensure that the appropriate infrastructure is in place to allow components from different languages to interoperate. In the system described here, this is accomplished by wrapping Fortran and C in C++ and using an implementation of Java RMI over Globus Nexus[6], [3] which interoperates with a C++ distributed computing library called HPC++.

Additional information about CCA may be found at the web site[16] and in the HPDC99 proceedings[2].

III. The Service Architecture of the CCAT Framework

Our implementation of CCA is called the CCA Toolkit (CCAT). It has been designed to operate over a number of different Grid middleware systems including Globus[15], Legion[17] and other distributed computing and e-commerce technologies such as Jini[24] and e-speak[11]. This is accomplished by abstracting the core services needed to authenticate, discover, launch, connect and monitor user applications and encapsulating them as components. We have designed and built five CCA framework service components.

- *A Directory Service.* The Directory Service provides the ability for any other component connected to it to search databases and remote repositories for other components with needed behaviors and required interfaces, and to add their own descriptions to these repositories.
- *A Registry Service.* This Registry Service can be used by any other component instance to advertise its existence and to search for other advertised instances. Designed primarily as means to advertise well-known service components, the registry is also a core part of our collaboration framework.
- *A Creation service.* This component can be used by another component to create a running instance of a third. This is the essential "boot strap" component in the framework.
- *A Connection service.* In both the CCA and the CORBA component models, components have two primary types of interfaces: those that a component "provides" to others and those that a component "uses" from others. In this model, if a component needs functionality provided by another component, and the uses-port and provides-port interfaces are type-compatible, the two may be connected, i.e. "uses" may call the interface "provided" by the other. The connection service enables that linkage either by a directly connecting them or establishing a remote invocation pathway.
- *An Event Service.* The Event Service acts as a "channel" for a publish-subscribe event model. This is used by components that wish to subscribe to event streams such as those generated by component creation, component connection, registry updates and application events.

The first version of the CCAT is based on an implementation over Globus, but work is underway to add additional functionality.

The advantage to encapsulating the framework services as application level components is two-fold. First, it makes it possible to easily swap in a new version of a service without changing applications or the basic framework. For example, changing the directory service to support LDAP discovery instead of our current WEBDAV [12] based protocol can be done by either extending the current service or replacing the directory component with one based wholly on LDAP. The key is that the information service port that the service provides to other components does not need to change. The second important advantage of this design is that other, higher level framework activities can now be built as application level components that use these services. We will return to this topic in the last section of this paper.

IV. An XML-based Information Directory Service.

One of the main contributions of this project is to show how XML can be used to extend the concept of an Interface Definition Language (IDL) as an extensible mechanism for describing software components.

Prior to instantiating components a user must locate and gather information about the components to be used. The user may be interested in choosing from a range of solvers for a particular problem, or in a solver from a particular author or a combination of the two. The user may even be interested in the locations of the installations of the components to be used. The Information Directory Service (IDS) fulfills the need for identifying the components that matches the interests of the user.

The Information Directory is an abstract concept; the IDS component implements an interface which includes the specified functionality. A storage-specific implementation of an IDS is called a *context*. The CCAT implements two Directory Contexts, one which uses the local file system and another which uses a WEBDAV server for storage. This model closely follows the JNDI [20] scheme of storing and looking up information, wherein any number of service providers can be plugged in as a backend, and the actual storage mechanism and protocols used are hidden from the user of the service.

The component descriptions themselves are XML documents that follow a prescribed but extensible XML schema definition language. The description includes such information as the component's name, author, purpose, available ports and the methods that can be invoked on them, etc. Another vital section of the description is installation information. Once a component has been written, the component developer then makes available at least one publicly accessible installation, so that collaborators can then also instantiate the component. The installation information includes such things as the location of the executable and the environment needed to start up an instantiation. It is important to note that while the information is in XML text format, CCAT does not expect the application to parse the text information. Instead, the IDS makes the information available to the developer through a hierarchical data structure accessible using keys. There is a one-one correspondence between an XML document and this data structure.

Appendix A shows part of the XML specification for a sparse linear system solver called SuperLU. In addition to the obvious fields like name, authors and description, it describes the ports this component has: a Provides-port used to load a linear system into the component and a Uses-port that is called when the solution is ready. The Port type interface details are also provided in the specification, but not listed here. Including these interface type details in the specification was a design error. That is because these interfaces are shared by many components ports. In the next version of CCAT, the port-type fields will be links to the XML files that contain these interface specifications.

In addition to standard ports, one additional Provides-port that is often included with a component is called the parameter port. This port is a CCA Provides-Port which can be used to change internal parameter values of the component. The specification of a parameter port also includes default values and the path for the class which encapsulates these parameters.

Finally, the XML specification contains information about how the component can be launched. In this case we show the information about how Globus GRAM can launch the component. Often a component has many different installations and launch mechanisms. Hence this information should probably be available via an XML X-link from the description rather than directly embedded here.

The use of XML as a specification medium is not limited to components. Registry instances, which contain the remote reference proxies to instantiated components are represented as XML documents. This allows them to be passed easily from one application to another. A user can even email a proxy representing a live component to another user. Also, in the next version of the CCAT system, XML will be used to define and encode the event streams.

V. Building Builders as Components

Most complete component architecture systems provide a tool that allows a user to select components, connect them together and test or execute them. Often, as with most Java Bean systems, this "builder" is a graphical programming tool. CCAT also has such a graphical builder as shown in figure 2.

In the CCAT a builder is actually another CCA component which is connected to the Provides-ports of the core Services (information, creation, connection and events), and is used to present some sort of interface to the user for invoking these services. Consequently, its design is made relatively simple by the component nature of the architecture. In scientific computing, one often needs to run a distributed computation multiple times with minor variations. This makes a script language interface for building such applications invaluable. To accomplish this, we developed a simple component that wrapped JPython to allow it to operate as a component builder. To illustrate how the script code looks to a Python [14], [18] programmer, Appendix B shows a script that builds a simple distributed PDE solver. (This is the same application as constructed in the graphical builder figure 2.)

To extend this scripted composition and execution sequence one step further we turned to another favorite tool of the computational scientist, Matlab. Once again, it was not hard to use the java plugin for Matlab to turn it into a component. The actual Matlab script is similar to the python script.

It should be noted that the example above only utilized the core Service components. Python and Matlab can also be used to write components, for example, the "assemble" component is a wrapped instance of Matlab.

One last example illustrates another kind of builder interface. A simple Java CCAT component can also be a Servlet[21] in an http web server. Figure 3 illustrates a web interface that has the same access to the framework services as do the other builders described above.

VI. Conclusion

This paper has described an implementation of the DOE Common Component Architecture that has been implemented on top of the Globus Grid framework. Two primary contributions of this work have been described here. First we have illustrated how XML can be used as an extensible and powerful tool to describe software components and running instances of these components. Our information directory service is capable of parsing the XML and delivering details about component functionality, parameter settings, interfaces and execution environments to any client including other components.

The second contribution of the work has been to show how one may build the component framework as a collection of "service components". These service components can easily be replaced or extended and they provide a common gateway by which an application component can easily access a wide variety of services. To illustrate this we described four different builder components: a Graphical User Interface, a Python interface, a Matlab interface, and a web-based interface.

References

[1] Aniruddha Gokhale and Douglas Schmidt. Principles for Optimizing CORBA Internet Inter-ORB Protocol Performance. In *31th Hawaii International Conference on System Sciences*, Jan 1998.

[2] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a Common Component Architecture for High-Performance Scientific Computing. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computation*, August 1999.

[3] Fabian Breg and Dennis Gannon. Compiler support for an rmi implementation using nexusjava. Technical Report 500, Extreme Computing Lab, Indiana University, Bloomington, Indiana, 1997.

[4] David Chappell. *Understanding ActiveX and OLE*. Microsoft Press, 1997.

[5] Robert Englander. *Developing Java Beans*. O'Reilly, 1997.

[6] I. Foster, C. Kesselman, and S. Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *J. Parallel and Distributed Computing*, 37:70–82, 1996.

[7] Ian Foster and Carl Kesselman. *The GRID: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1998.

[8] A. Gokhale, D.C. Schmidt, T. Harrison, and G. Parulkar. Towards real-time CORBA. *IEEE Communications Magazine*, 14(2), Feb 1997.

[9] Java Grande. Computing Portals, visited 02-19-00. http://www.computingportals.org.

[10] Numerical Algorithms Group. IRIS Explorer, visited 8-20-97. http://www.nag.co.uk/Welcome_IEC.html.

[11] Hewlett-Packard. E-Speak, visited 1-10-2000. http://www.e-speak.hp.com/.

[12] IETF. WebDav, visited 8-20-99. http://www.ics.uci.edu/ ejw/authoring/.

[13] William E. Johnston, Dennis Gannon, and Bill Nitzberg. Grids as production computing environments: The engineering aspects of nasa's information power grid. In *Proceedings of Eight IEEE International Symposium on High Performance Distributed Computing Conference, Redondo Beach, California*, August 3-6 1999.

[14] jpython.org. JPython, visited 3-1-00. http://www.jpython.org.

[15] Argonne National Lab. Globus, visited 7-15-99. http://www.globus.org.

[16] Argonne National Laboratory, Indiana Univeristy, The Advanced Computing Laboratory at Los Alamos National Laboratory, Lawrence Livermore National Lab, and Univeristy of Utah. Common Component Architecture, visited 1-10-2000. http://z.ca.sandia.gov/ cca-forum see also http://www.extreme.indiana.edu/ccat.

[17] Michael J. Lewis and Andrew Grimshaw. The Core Legion Object Model. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society Press, August 1996.

[18] M. Lutz. *Programming Python*. O'Reilly and Associates, 1996.

[19] Microsoft. COM, visited 7-22-99. http://www.microsoft.com/com.

[20] SUN Microsystems. JNDI, visited 3-7-2000. http://java.sun.com/products/jndi/.

[21] Sun Microsystems. Servlets, visited 3-7-2000. http://java.sun.com/products/servlet.

[22] SUN Microsystems. Java Beans, visited 7-11-99. http://java.sun.com/beans/.

[23] Sun Microsystems. EJB, visited 7-15-99. http://java.sun.com/products/ejb/index.html.

[24] Sun Microsystems. Jini, visited 7-15-99. http://www.sun.com/jini.

[25] NPCA. WebFlow, visited 8-20-99. http://osprey7.npac.syr.edu:1998/iwt98/products/webflow/.

[26] University of Utah. SCIRun, visited 7-10-99. http://www.cs.utah.edu/sci/scirun/.

[27] OMG. Corba Component Model, visited 1-11-2000. http://www.omg.org/cgi-bin/doc?orbos/97-06-12.

[28] S.G. Parker and C.R. Johnson. SCIRun: A scientific programming environment for computational steering. In *Supercomputing '95*. IEEE Press, 1995.

[29] Douglas C. Schmidt. Lessons learned building reusable oo frameworks for distributed software. *Communications of the ACM Special Issue on OO Application Frameworks*, 40(10), October 1997.

Appendix

I. Sample XML Specification for a Sparse Solver

```
<!-- XML specification for a sparse linear system solver:SuperLU -->
<component-info>
  <component-name>/ccat/XML/new/SuperLU</component-name>
  <author-list>
     <author>Xiaoye S. Li</author>
     <author>Bramley (mods)</author>
  </author-list>
  <component-description> Solves sparse linear ... </component-description>
  ...
  <port-list>
     <port>
        <port-name>inputSLS</port-name>
        <my-port-type>SparseLinearSystem_idl</my-port-type>
        <port-dir>provides</port-dir>
        <port-description>
            This port receives the linear system
        </port-description>
     </port>
     <port>
        <port-name>outputSV</port-name>
        <my-port-type>SolutionVector_idl</my-port-type>
        <port-dir>uses</port-dir>
        <port-description>
             This port sends the solution vector
        </port-description>
     </port>
  </port-list>
  <parameter-block>
     <parameter-port-name>inputSuperLUParms</parameter-port-name>
     <parameter-class>idl.superLUParms.SuperLUParms</parameter-class>
     <parameter-port-class>SuperLUParms_idl</parameter-port-class>
     <method-name>sendSuperLUParms</method-name>
        <parameters>
            <name>FactorizationJob</name>
            <default-value>3</default-value>
        </parameters>
        <parameters>
            <name>panelSize</name>
            <default-value>8</default-value>
        </parameters>
        <parameters>
            <name>MarkovitzParameter</name>
            <default-value>0.01</default-value>
        </parameters>
        ...
  </parameter-block>
  <installation>
        <host-name>caledonia.cs.indiana.edu</host-name>
        <creation-info>
             <creation-method>gram</creation-method>
             <creation-env>
                <name-value-pair>
```

```
                <name>globus-run-script-path</name>
                <value>/u/lib/bin/caledoniaGlobusRun.sh</value>
            </name-value-pair>
            ...
         </creation-env>
      </creation-info>
   </installation>
</component-info>
<!-- End of XML specification for a sparse linear system solver:SuperLU -->
```

II. Sample Python Script for Distributed PDE Solver

```
import ccat

# Create component models for the components
# TimeGen, Discretizer, BasicInfo, Splib, and Assemble.
timegen = ccat.createComponent ('TimeGen')
discretizer = ccat.createComponent ('Discretizer')
splib = ccat.createComponent ('Splib')
assemble = ccat.createComponent ('Assemble')

ccat.setMachineName (timegen, 'baldy.extreme.indiana.edu')
ccat.setCreationMechanism (timegen, 'gram')
ccat.setMachineName (discretizer, 'baldy.extreme.indiana.edu')
ccat.setCreationMechanism (discretizer, 'gram')
ccat.setMachineName (assemble, 'baldy.extreme.indiana.edu')
ccat.setCreationMechanism (assemble, 'gram')
ccat.setMachineName (splib, 'baldy.extreme.indiana.edu')
ccat.setCreationMechanism (splib, 'gram')

ccat.createInstance (timegen)
ccat.createInstance (discretizer)
ccat.createInstance (assemble)
ccat.createInstance (splib)

ccat.connectPorts (timegen, 'outputTimeStep', discretizer, 'inputTimeStep')
ccat.connectPorts (discretizer, 'outputGeometry', assemble, 'inputGeometry')
ccat.connectPorts (discretizer, 'outputSLS', splib, 'inputSLS')
ccat.connectPorts (splib, 'outputSV', assemble, 'inputSV')
ccat.connectPorts (assemble, 'outputRunItAgain', timegen, 'inputRunItAgain')

ccat.setParams (timegen)  # uses defaults
ccat.execute (timegen)    # timegen calls the others

# End of Python script
```

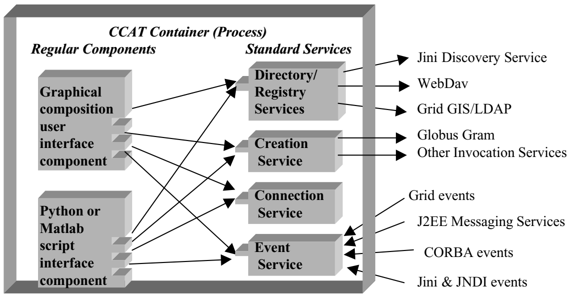

Fig. 1. Each CCAT Container process contains both user components and one instance of each of the Service Components. Many of the Service Components act as gateways to external wide-area services.

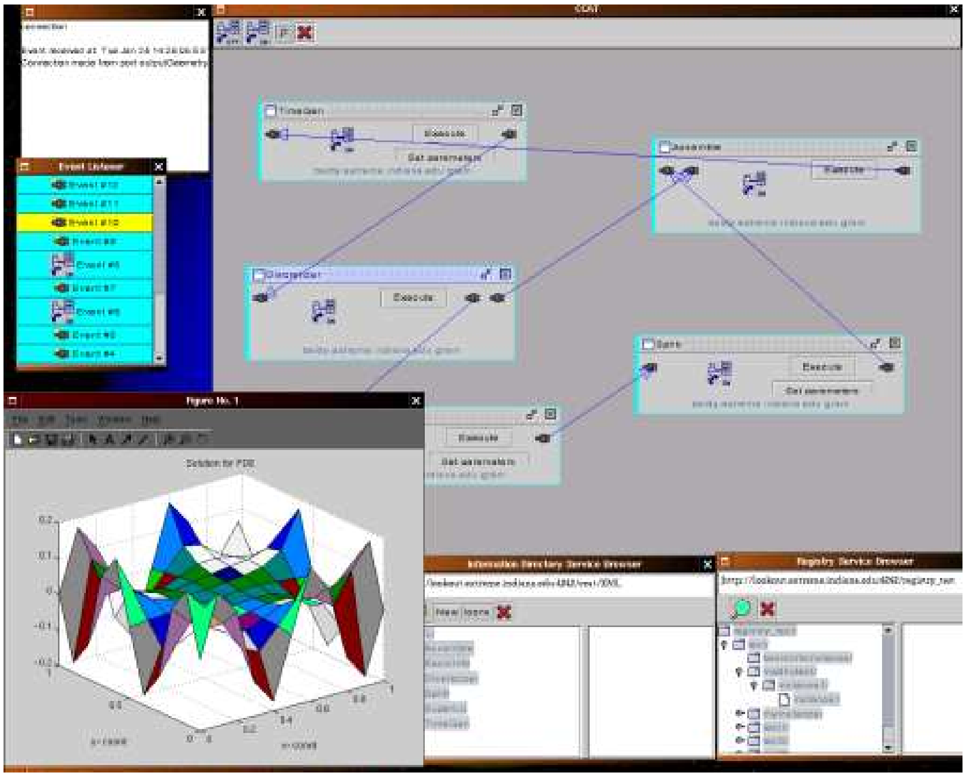

Fig. 2. CCAT session with java GUI

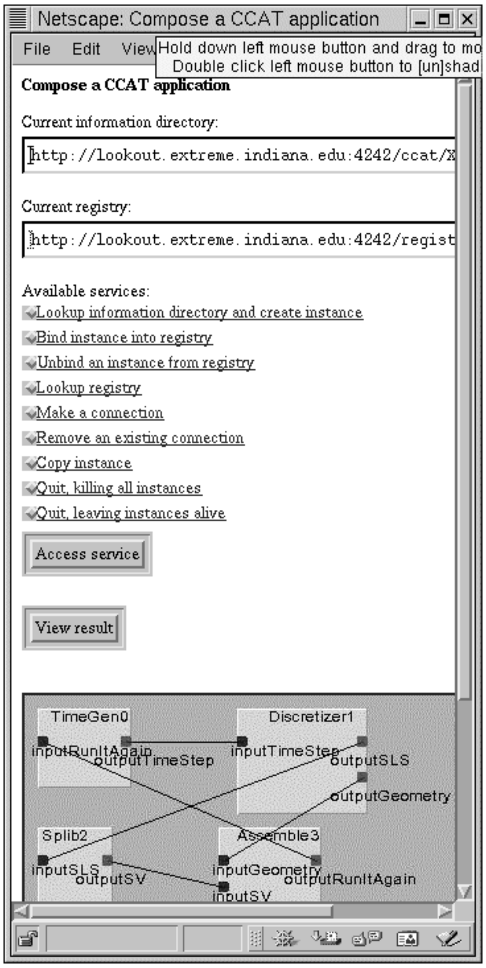

Fig. 3. CCAT session with servlet interface