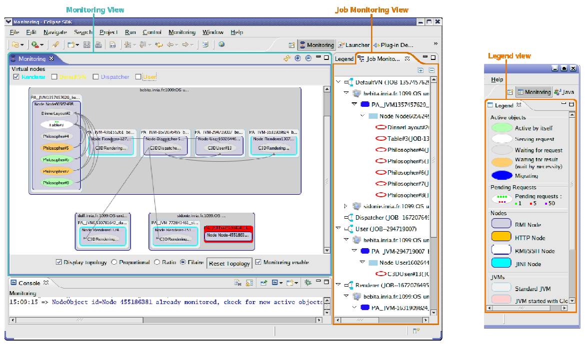
ProActive

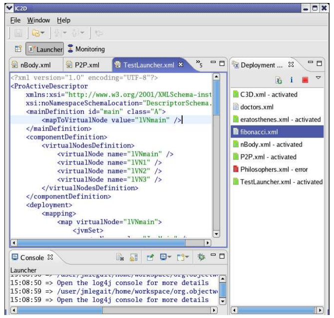
Programming, Composing, Deploying on the Grid

Version 3.9 – April 2008

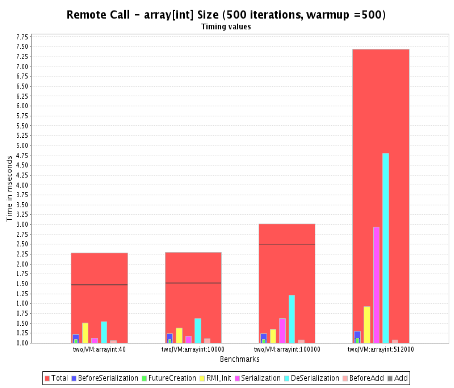
Programming Reference Booklet



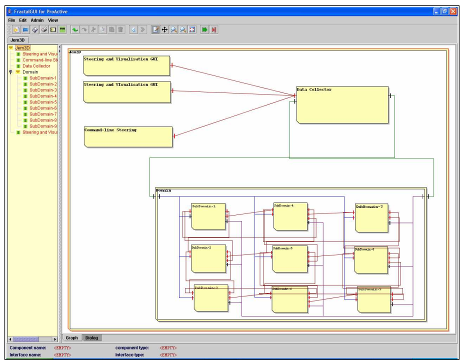
Graphical and Textual Visualization



Application Launcher



Application Performance Evaluation



Components GUI

ObjectWeb Open Source Middleware

CoreGRID

GridCOMP Effective Components for the Grids

ProActive is a Java library for **parallel**, **distributed**, and **concurrent** computing, also featuring **mobility** and **security** in a uniform framework. **ProActive** provides a comprehensive API. The library is based on an Active Object pattern that is a uniform way to encapsulate:

- a **remotely** accessible object,
- a **thread** as an asynchronous activity,
- an **actor** with its own script,
- a **server** of incoming requests,
- a **mobile** and potentially secure entity,
- a **component** with server and client interfaces.

ProActive is only made of standard Java classes, and requires **no changes to the Java Virtual Machine**. Overall, it simplifies the programming of applications distributed over Local Area Network (LAN), Clusters, Intranet or Internet GRIDs.

ProActive interoperates with the following official or de facto standards:

- Web Service Exportation
- JMX Connector
- HTTP Transport, OSGi
- SSH, RSH, RMI/SSH Tunneling
- LSF, PBS, OAR, Sun Grid Engine
- Globus GT2, GT3, GT4
- sshGSI
- NorduGrid
- Unicore, EGEE gLite

Table of Contents

Main Concepts and Definitions

Programming	Active Objects (AO)	A remote object, with its own thread, receiving calls on its public methods
	FIFO Activity	By default an AO executes the request it receives one after the other, in the order they were received
	No-sharing	Standard Java objects cannot be referenced from 2 AOs, ensured by deep-copy of constructor params, method parameters, and results
	Asynchrony	Method calls towards AOs are asynchronous
	Future	The result of a non-void asynchronous method call
	Request	The occurrence of a method call towards an AO
	Service	The execution by an AO of a request
	Reply	After a service, the result is sent back to the caller
	Wait-by-necessity	Automatic wait upon the use of a still awaited future
	Automatic Continuation	Automatic transmission of futures and replies between AO and JVMs
	Migration	An AO moving from one JVM to another. The AO have weak computational mobilitybecause if the AO migrates the stack is lost
	Group	A typed group of objects or AOs. Methods are called in parallel on all group members

Composing	Component	Made of AOs, a component defines server and client interfaces
	Primitive Component	Directly made of Java code and AOs
	Composite Component	Contains other components (primitives or composites)
	Parallel Component	A composite that is using groups to multicast calls to inner components

Deploying	Virtual Node (VN)	An abstraction (a string) representing the location for AOs at creation
	Deployment Descriptor	An XML file where a mapping VN → JVMs → Machine is specified
	Node	The result of a mapping VN → JVMs. After activation, a VN contains a set of nodes living in a set of JVMs
	P2P	A P2P network of self-organized JVMs, on which applications are deployed
	Fault tolerance	Applications can be turned fault-tolerant simply by modifying the deployment descriptor
	Security	X.509 Authentication, Integrity, and Confidentiality defined at deployment in an XML
	IC2D	Interactive Control and Debugging of Distribution: a Graphical environment for monitoring and steering

Main Principles: Asynchronous Method Calls and Implicit Futures

```
A a = (A) PAActiveObject("A", params, node);
   // Create an active Object of type A in the JVM specified by Node
a.foo (param);
   // A one way typed asynchronous communication towards the (remote)
   // AO a. A request is sent to a.
v = a.bar (param);
   // A typed asynchronous communication with result.
   // v is first an awaited Future, to be transparently filled up
   // after service of the request, and the reception of a reply
v.gee (param);
   // Use of the result of an asynchronous call.
   // If v is still an awaited future, it triggers an automatic
   // wait: Wait-by-necessity
```

Explicit Synchronization

```
boolean PAFuture.isAwaited(Object);
   // Returns True if the object is still an awaited Future
void  PAFuture.waitFor(Object);
   // Blocks until the object is no longer awaited
   // The object is a future
void  PAFuture.waitForAll(Vector);
   // Blocks until all the objects in Vector are no longer awaited
int   PAFuture.waitForAny(Vector);
   // Blocks until one of the objects in Vector is no longer awaited.
   // Returns the index of the available future.
```

Programming AO Activity and Services

When an AO must implement an activity that is not FIFO, the `RunActive` interface has to be implemented: it specifies the AO behaviour in the method named `runActivity(Body body)`.

Example:

```
public class  A implements  RunActive {
   // Implements RunActive for programming a specific behaviour;
   // runActivity is automatically called when such an AO is created
public void  runActivity(Body body) {
Service service = new Service(body);
while ( !terminate ) {
          ...    // Do some activity on its own
          ...    // Do some services, e.g. a FIFO service on method
                //  named foo. See Service section
          service.serveOldest("foo");
```

```
        ...
      }...
```

Two other interfaces can also be specified:

```
Interface InitActive
void  initActivity(Body body)
   // Initializes the activity of the active object.
   // Not called in case of restart after migration.
   // Called before runActivity() method, and only once.

Interface EndActive
void  endActivity(Body body)
   // Finalizes the active object after the activity stops by itself.
   // Called after the execution of runActivity() method, and only
   // once. Not called before a migration.
```

Reactive Active Object

Even when an AO is busy doing its own work, it can remain reactive to external events (method calls). One just has to program non-blocking services to take into account external inputs.

```
public class  BusyButReactive  implements  RunActive {

public void  runActivity(Body body) {
Service service = new Service(body);
while ( ! hasToTerminate ) {
         ...    // Do some activity on its own
         ...    // Non blocking service
         service.serveOldest("changeParameters", "terminate");
         ...
      }
    }
public void  changeParameters () {// change computation parameters}
public void   terminate (){ hasToTerminate=true;}
}
```

It also allows one to specify explicit termination of AOs Of course, the reactivity is up to the length of going around the loop. Similar techniques can be used to start, suspend, restart, and stop AOs.

Service Methods

A service method selects a given request amongst the pending calls, and executes it. Those methods are to be used by modifying the `RunActivity` method when a FIFO service is not appropriate.

Non-blocking services: returns immediately if no matching request is pending. These methods are all available in the `Service` class.


```
void serveOldest();
   // Serves the oldest request in the request queue
void  serveOldest(String methodName)
   // Serves the oldest request aimed at a method of name methodName
void serveOldest(RequestFilter requestFilter)
   // Serves the oldest request matching the criteria given by the
   // filter
```

Blocking services: waits until a matching request can be served

```
void blockingServeOldest();
   // Serves the oldest request in the request queue
void  blockingServeOldest(String methodName)
   // Serves the oldest request aimed at a method of name methodName
void blockingServeOldest(RequestFilter requestFilter)
   // Serves the oldest request matching the criteria given by the
   // filter
```

Blocking timed services: wait a matching request at most a time given in ms

```
void blockingServeOldest (long timeout)
   // Serves the oldest request in the request queue.
   // Returns after timeout (in ms) if no request is available
void  blockingServeOldest(String methodName, long timeout)
   // Serves the oldest request aimed at a method of name methodName
   // Returns after timeout (in ms) if no request is available
void blockingServeOldest(RequestFilter requestFilter)
   // Serves the oldest request matching the criteria given by the
   // filter
```

Waiting primitives:

```
void waitForRequest();
   // Wait until a request is available or until the body terminates
void  waitForRequest(String methodName);
   // Wait until a request is available on the given method name,
   // or until the body terminates
```

Others:

```
void fifoServing();
   // Start a FIFO service policy. Call does not return. In case of
   // a migration, a new runActivity() is started on the new site.
void   lifoServing()
   // Invoke a LIFO policy. Call does not return. In case of
   // a migration, a new runActivity() will be started on the new
   // site.
void  serveYoungest()
   // Serves the youngest request in the request queue
```

```
void  flushAll()
   // Removes all requests in the pending queue
```

Active Object Creation

```
Object PAActiveObject.newActive(String classname, Object[]
         constructorParameters,Node node);
   // Creates a new AO of type classname. The AO is located on the
   // given node, or on a default node in the local JVM if the given
   // node is null
Object[]  PAActiveObject.newActive(String classname,Object[]
         constructorParameters,VirtualNode virtualnode);
   // Creates a new set of AO of type classname.
   // The AOs are located on each JVM the Virtual Node is mapped onto
Object  PAActiveObject.turnActive(Object, Node node);
   // Copy an existing Java object and turns it into an AO.
   // The AO is located on the given node, or on a default node
```

Groups

A typed collection of active objects on which calls are performed in parallel (hiding latency), and asynchronously (returning a group of futures).

```
A ga = (A) PAGroup.newGroup( "A", params, nodes);
   // Creates at once a group of AO of type "A" in the JVMs specified
   // by nodes. ga is a Typed Group of type "A".
   // The number of AO created matches the number of param arrays.
   // Nodes can be a Virtual Node defined in an XML descriptor
ga.foo(...);
   // A general group communication without result.
   // A request to foo is sent in parallel to AOs in group ga
V gv = ga.bar(...);
   // A general group communication with a result.
   // gv is a typed group of "V", which is first a group
   // of awaited Futures, to be filled up asynchronously
gv.gee (...);
   // Use of the result of an asynchronous group call. It is also a
   // collective operation: gee method is called in parallel on each
   // object in group.
   // Wait-by-necessity occurs when results are awaited
Group ag = PAGroup.getGroup(ga);
   // Get the group representation of a typed group
ag.add(o);
   // Add o in the group ag. o can be a standard Java object or
   // an AO, and in any case must be of a compatible type
ag.remove(index)
   // Removes the object at the specified index
A ga2 = (A) ag.getGroupByType();
   // Returns the typed view of a group
void PAGroup.setScatterGroup(g);
   // By default, a group used as a parameter of a group
   // communication is sent to all as it is (deep copy of the group).
   // When set to scatter, upon a group call (ga.foo(g)) such a
```



```
   // scatter parameter is dispatched in a round robing fashion to
   // AOs in the target group, e.g. upon ga.foo(g)
void  PAGroup.unsetScatterGroup(g);
   // Get back to the default: entire group transmission in all group
   // communications, e.g. upon ga.foo(g)
```

Explicit Group Synchronizations

Methods to wait for the availability of all results of a group call, or the first one(s) to be available. Methods are both in interface `Group`, and static in class `org.objectweb.proactive.api.PAGroup`

```
boolean  PAGroup.allAwaited (Object);
   // Returns true if object is a group and all members are still
   // awaited
boolean PAGroup.allArrived (Object);
   // Returns False only if at least one member is still awaited
void   PAGroup.waitAll (Object);
   // Wait for all the members in group to arrive (all no longer
   // awaited)
void  PAGroup.waitN (Object, int nb);
   // Wait for at least nb members in group to arrive
int  PAGroup.waitOneAndGetIndex (Object);
   // Waits for at least one member to arrived, and returns its index
```

OO SPMD

A group in which each group member has a proxy to the group it belongs to. Typically used in applications with sub-domain decomposition, numerical SPMD (Simple Program, Multiple Data), etc.

```
A spmdGroup  =  (A) PASPMD.newSPMDGroup("A", params, nodes);
   // Creates an SPMD group and creates all members with params on
   // the nodes.
   // An SPMD group is a typed group in which every member has a
   // reference to the others (the SPMD group itself).
A mySpmdGroup = (A) PASPMD.getSPMDGroup();
   // Returns the SPMD group of the activity.
int rank = PASPMD.getMyRank();
   // Returns the rank of the activity in its SPMD group.
PASPMD.barrier("barrierID");
   // Blocks the activity (after the end of the current service)
   // until all other members of the SPMD group invoke the same
   // barrier. Three barriers are available : total barrier,
   // neighbors based barrier and method based barrier.
```

Migration

Computational mobility: an active object changes of JVM at execution.When an active object migrates the stack is lost – weak migration. The `migrateTo` primitive is a static method.

```
void  PAMobileAgent.migrateTo(Object o);
   // Migrate the current AO to the same JVM as the AO
void  PAMobileAgent.migrateTo(String nodeURL);
   // Migrate the current AO to JVM given by the node URL
int   PAMobileAgent.migrateTo(Node node);
   // Migrate the current AO to JVM given by the node
```

To initiate the migration of an object from outside, define a public method, that upon service will call `migrateTo` primitive:

```
public void moveTo(Object o) {
     try{
       PAMobileAgent.migrateTo(o);
     } catch (Exception e) {
       e.printStackTrace();
       logger.info("Cannot migrate.");
     }
   }
```

The following methods are in the `MigrationStrategyManager` interface and serve to set various migration actions:

```
void onDeparture(String MethodName);
   // Specification of a method to execute before migration
void  onArrival(String MethodName);
   // Specification of a method to execute after migration, upon the
   // arrival in a new JVM
void setMigrationStrategy(MigrationStrategy ms);
   // Specifies a migration itinerary
void migrationStrategy.add(Destination d);
   // Adds a JVM destination to an itinerary
void migrationStrategy.remove(Destination d) ;
   // Remove a JVM destination in an itinerary
```

Exceptions

ProActive has an exception mechanism that handles functional exceptions. Functional exceptions permit asynchronous calls with exceptions:

```
PAException.tryWithCatch(MyException.class);// Just before the try
try { // Some asynchronous calls with potential MyException
    PAException.endTryWithCatch();// At the end of the try
} catch (MyException e) {
    // ...
} finally {
    PAException.removeTryWithCatch();// At the beginning of the
finally
```



```
}
```

The behaviour of the default handler (if no one could handle the exception) is to throw the exception if it's on the proxy side, or log it if it's on the body side.

Components

Components are formed from AOs, a component is linked and communicates with other remote components. A component can be composite, made of other components, and as such itself distributed over several machines. Component systems are defined in XML files (ADL: Architecture Description Language); these files describe the definition, the assembly, and the bindings of components. Components follow the Fractal hierarchical component model specification and API(see http://fractal.objectweb.org). The methods needed for working with components are in `org.objectweb.proactive.core.component.Fractive` To create a component use:

```
Component     newFcInstance()

Component     newFcInstance(Type type, ControllerDescription
                      controllerDesc, ContentDescription contentDesc)
```

Web Services

ProActive allows active objects exportation as web services. The service is deployed onto a Jakarta Tomcat web server with a given URL. It is identified by its URN, a unique id of the service. It is also possible to choose the exported methods of the object. The WSDL file matching the service will be accessible at http://localhost:8080/servlet/wsdl?id=a for a service whose name is "a" and whose id is deployed on a web server which location is http://localhost:8080.

```
A a = (A) PAActiveObject("A", new Object []{});
   // Constructs an active object
String [] methods = new String [] {"foo", "bar"};
   // A String array containing the exported methods
WebServices.exposeAsWebService(a,"http://localhost:8080","a",methods);
   // Export the active object as a web service
WebServices.unExposeAsWebService("a", "http://localhost:8080");
   // Undeploy the service "a" on the web server located at
   // http://localhost:8080
```

Deployment

Virtual Nodes (VN) allow one to specify the location where to create AOs. A VN is uniquely identified as a String, is defined in an XML Deployment Descriptor where it is mapped onto JVMs. JVMs are themselves mapped onto physical machines: VN --> JVMs --> Machine. Various protocols can be specified to create JVMs onto machines (ssh, Globus, LSF, PBS, rsh, rlogin, Web Services, etc.).

After activation, a VN contains a set of nodes, living in a set of JVMs. Overall, VNs and deployment descriptors allow to abstract away from source code: machines, creation, lookup and registry protocols.

Descriptor example: creates one JVM on the local machine

```
...
  <virtualNodesDefinition>
   <virtualNode name="Dispatcher"/><!-- Name of the Virtual Node
       that will be used in program source -->
  </virtualNodesDefinition>
...

<mapping>
  <!-- This part contains the mapping VNs -- JVMs -->
   <map virtualNode="Dispatcher">
    <jvmSet>
     <vmName value="Jvm1"/>
     <!-- Virtual Node Dispatcher is mapped onto Jvm1 -->
    </jvmSet>
   </map>
  </mapping>
  <jvms>
   <jvm name="Jvm1">
   <!-- This part defines how the JVM will be obtained: creation or
       acquisition: creation in this example -->
    <creation>
     <processReference refid="creationProcess"/><!-- Jvm1 will be
       created using creationProcess defined below -->
    </creation>
   </jvm>
  </jvms>
 </deployment>
 <infrastructure>
  <processes>
   <processDefinition id="creationProcess">
   <!-- Definition of creationProcess referenced above -->
    <jvmProcess
       class="org.objectweb.proactive.core.process.JVMNodeProcess"/>
     <!-- creationProcess is a jvmProcess. The JVM will be created
       on the local machine using default settings (classpath, java
       path,...) -->
   </processDefinition>
  </processes>
...
```

Deployment API

```
ProActiveDescriptor pad = PADeployment.getProActiveDescriptor(String
         File);
   // Returns a ProActiveDescriptor object from the xml
   // descriptor file name
pad.activateMapping(String VN);
```



```
   // Activates the given Virtual Node: launches or acquires
   // all the JVMs the VN is mapped onto
pad.activateMappings();
   // Activates all VNs defined in the ProActiveDescriptor
VirtualNode vn = pad.getVirtualNode(String)
   // Creates at once a group of AO of type "A" in the JVMs specified
   // by the given vn. The Virtual Node is automatically activated if
   // not explicitly done before
Node[] n = vn.getNodes();
   // Returns all nodes mapped to the target Virtual Node
Object[] n[0].getActiveObjects();
   // Returns a reference to all AOs deployed on the target Node
ProActiveRuntime part = n[0].getProActiveRuntime();
   // Returns a reference to the ProActive Runtime (the JVM) where
   // the node has been created
pad.killall(boolean softly);
   // Kills all the JVMs deployed with the descriptor
   // not softly: all JVMs are killed abruptely
   // softly: all JVMs that originated the creation of a RMI registry
   // waits until registry is empty before dying
```

File Transfer Deployment

File Transfer Deployment is a tool for transfering files at deployment time. This files are specified using the ProActive XML Deployment Descriptor in the following way:

```
<VirtualNode name="exampleVNode" FileTransferDeploy="example"/>
....
</deployment>
<FileTransferDefinitions>
   <FileTransfer id="example">
      <file src="hello.dat" dest="world.dat"/>
      <dir src="exampledir" dest="exampledir"/>
  </FileTransfer>
  ...
</FileTransferDefinitions>
<infrastructure>
....
<processDefinition id="xyz">
  <sshProcess>...
    <FileTransferDeploy="implicit"> <!-- referenceID or keyword
         "implicit" (inherit)-->
      <copyProtocol>processDefault, scp, rcp</copyProtocol>
      <sourceInfo prefix="/home/user"/>
      <destinationInfo prefix="/tmp" hostname="foo.org"
         username="smith" />
    </FileTransferDeploy>
  </sshProcess>
</processDefinition>
```

Peer-to-Peer Infrastructure

ProActive offers the ability to deploy a P2P infrastructure of JVMs over desktop machines. The infrastructure is self-organized and configurable and maintains a dynamic network of JVMs for deploying computational applications.

Deploying the infrastructure

```
$ cd ProActive/scripts/unix/p2p
$ ./startP2PService [-acq acquisitionMethod] [-port portNumber] [-s Peer ...]
```

A simple example

```
first.peer.host$ ./startP2PService.sh
second.peer.host$ ./startP2PService.sh -s //first.peer.host
third.peer.host$ ./startP2PService.sh -s //second.peer.host
```

Acquiring Nodes

```
<services>
  <serviceDefinition id="p2plookup">
    <P2PService nodesAsked="50" acq="rmi" port="6666">
     <peerSet>
       <peer>//second.peer.host</peer>
     </peerSet>
    </P2PService>
  </serviceDefinition>
  ...
</services>
```

Fault Tolerance

ProActive can provide fault-tolerance capabilities through two different protocols: a Communication-Induced Checkpointing protocol (CIC) or a Pessimistic Message Logging protocol (PML). Making a ProActive application fault-tolerant is fully transparent; active objects are turned fault-tolerant using Java properties that can be set in the deployment descriptor. The programmer can select at deployment time the most adapted protocol regarding the application and the execution environment.

A fault-tolerant deployment descriptor

```
<virtualNodesDefinition>
 <virtualNode name="NonFT-Workers" property="multiple"/>
 <virtualNode name="FT-Workers" property="multiple"
     ftServiceId="appli"/>
</virtualNodesDefinition>
...
 <serviceDefinition id="appli">
```



```
   <faultTolerance>
    <!-- Protocol selection : cic or pml -->
    <protocol type="cic"></protocol>
    <!-- URL of the fault-tolerance server -->
    <globalServer
      url="rmi://localhost:1100/FTServer"></globalServer>
    <!-- URL of the resource server; all the nodes mapped on the
      FT-Workers virtual node will be registered in as resource
      nodes for recovery -->
    <resourceServer
      url="rmi://localhost:1100/FTServer"></resourceServer>
    <!-- Average time between two consecutive checkpoints for each
      object -->
    <ttc value="5"></ttc><!-- in seconds -->
   </faultTolerance>
  </serviceDefinition>
 </services>
```

Starting the fault-tolerance server

The global fault-tolerance server can be launched using the `ProActive/scripts/[unix|windows]/FT/startGlobalFTServer.[sh|bat]` script, with 5 optional parameters:

- **the protocol**: `-proto [cic|pml]`. Default value is cic.
- **the server name**: `-name [serverName]`. Default name is FTServer.
- **the port number**: `-port [portNumber]`. Default port number is 1100.
- **the fault detection period**: `-fdperiod [periodInSec]`, the time between two consecutive fault detection scanning. Default value is 10 sec.
- **the URL of a p2p service** that can be used by the resource server: `-p2p [serviceURL]`. No default value.

Security

An X.509 Public Key Infrastructure (PKI) allowing communication Authentication, Integrity, and Confidentiality (AIC) to be configured in an XML security file, at deployment, outside any source code. Security is compatible with mobility, allows for hierarchical domain specification and dynamically negotiated policies.

An example of specification

```
<Rule>
   <From><Entity type="VN" name="VN1"/> </From>
   <To> <Entity type="VN" name="VN2"/> </To>
   <Communication>
    <Request value="authorized">
      <Attributes authentication="required"
                  integrity="required"
                  confidentiality="optional"/>
    </Request>
  </Communication>
  <Migration>denied</Migration>
```

```
  <AOCreation>denied</AOCreation>
</Rule>
```

This rule specifies that: from Virtual Node "VN1" to the VN "VN2", the communications (requests) are authorized, provided authentication and integrity are being used, while confidentiality is optional. Migration and AO creation are not authorized.

Branch and Bound API

This is a simple API for solving parallel problems using a Branch-And-Bound infrastructure.

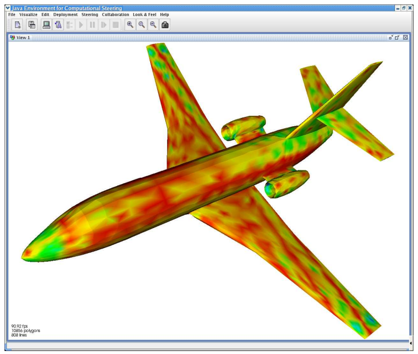
```
public class YourTask extends Task {

  public Result execute() // For computing a solution
  public Vector split() // For  generating sub-tasks
  public Result gather(Result[] results) // Gathering all results
  public void initLowerBound() // For computing a lower bound
  public void initUpperBound() // For computing a upper bound
```

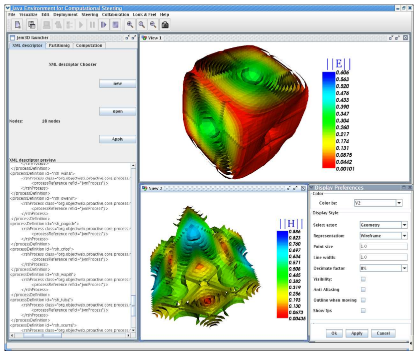
Start the computation:

```
Task task = new YourTask(someArguments);
Manager manager =  ProActiveBranchNBound.newBnB(task,
                        nodes,LargerQueueImpl.class.getName());
Result futureResult = manager.start(); // this call is asynchronous
```

JEM3D: Java Electro-Magnetism with

ProActive
Programming, Composing, Deploying on the Grid

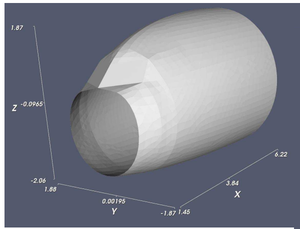




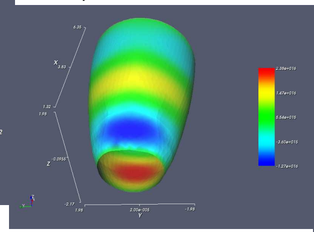
Vibro-Acoustic Code Coupling of MPI Legacy with Components

Courtesy of EADS CCR

Airbus A319 Cockpit



Cockpit solution for 100 Hz



Cockpit solution for 300 Hz