



Project no. FP6-034442

GridCOMP

**Grid programming with COMPONENTS : an advanced component platform
for an effective invisible grid**

STREP Project

Advanced Grid Technologies, Systems and Services

**D.GIDE.04 – Grid IDE Tuned Prototype and Final Documentation
(Manual and Detailed Architectural Design)**

Due date of deliverable: 30 November 2008

Actual submission date: 19 January 2009

Start date of project: 1 June 2006

Duration: 33 months

Organisation name of lead contractor for this deliverable: UoW

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
P	PUBLIC	PU

Keyword List: Integrated Development Environment, Component Composition, Deployment,
Component Monitoring and Steering
Responsible Partner: Vladimir Getov, UoW

MODIFICATION CONTROL			
Version	Date	Status	Modifications made by
0	DD-MM-YYYY	Template	Patricia Ho-Hune
1	10-12-2008	Draft	Artie Basukoski
2	12-12-2008	Draft	Stavros Isaiadis
3	15-12-2008	Draft	Alessandro Basso
4	18-12-2008	Draft	Vladimir Getov
5	6-1-2009	Draft	Stavros Isaiadis
6	8-1-2008	Draft	Artie Basukoski
7	10-1-2008	Final	Vladimir Getov

Deliverable manager

- Vladimir Getov, UoW

List of Contributors

- Stavros, Isaiadis, UoW
- Artie, Basukoski, UoW
- Alessandro, Basso, UoW

List of Evaluators

- Gaston Freire Amoedo, GridSystems
- Marco Aldinucci, UNIPI

Summary

Component-oriented development is a software design methodology which enables users to build large scale Grid systems by integrating independent and possibly distributed software modules (components), via well defined interfaces, into higher level components. The main benefit from such an approach is improved productivity, firstly, by abstracting away network level functionalities, thus reducing the technical demands, secondly, by combining components into higher level components, component libraries can be built up incrementally and made available for reuse. The development platform, which is tightly integrated with Eclipse software framework, was designed to empower the developer with all the tools necessary to compose, deploy, monitor, and steer Grid applications. In the first part of this report, we present our design for the final tuned version of an integrated development environment for Grids (GIDE) to support component-oriented development, deployment, monitoring, and steering of large-scale Grid applications. The second part consists of a manual for the installation and application of the GIDE.

Table of Content

1	DETAILED ARCHITECTURAL DESIGN	4
1.1	OBJECTIVES	4
1.2	DESIGN	5
1.3	COMPOSITION PERSPECTIVE	6
1.3.1	Component Repository View	7
1.3.2	Properties View	8
1.3.3	GIDE Log View	9
1.3.4	Editor	9
1.3.5	Validation	11
1.4	DEPLOYMENT PERSPECTIVE	11
1.4.1	Descriptor Repository	12
1.5	MONITORING AND STEERING PERSPECTIVE	14
1.5.1	Monitoring View	15
1.5.2	Live Monitoring Canvas	16
1.5.3	Steering	17
1.5.4	Resource Monitor View	18
2	MANUAL	19
2.1	SETTING UP	19
2.1.1	Eclipse Project Version	19
2.1.2	Behavioural Skeletons Rules file Editor	19
2.1.3	IC2D plugins for GIDE Monitoring	19
2.1.4	Installation	19
2.1.5	Directory and File Structure	20
2.2	BASIC USAGE	21
2.2.1	Starting Up	21
2.2.2	Using GIDE Tools and Perspectives	21
2.3	COMPOSITION	22
2.3.1	Importing ADL files	22
2.3.2	Exporting to ADL files	24
2.3.3	Component Repository	27
2.3.4	Properties	28
2.4	DEPLOYMENT	30
2.4.1	Descriptor Repository	30
2.5	MONITORING AND STEERING	31
2.5.1	Architecture Monitoring	32
2.5.2	Steering	32
2.5.3	Resource Monitor	33
2.6	MISCELLANEOUS	33
2.6.1	GIDE Log Viewer	33
	BIBLIOGRAPHY	35

1 Detailed Architectural Design

1.1 Objectives

The role of WP4 evolves around the design and implementation of a grid IDE for the rapid development, deployment, monitoring and steering for Grid systems, as shown in the following diagram.

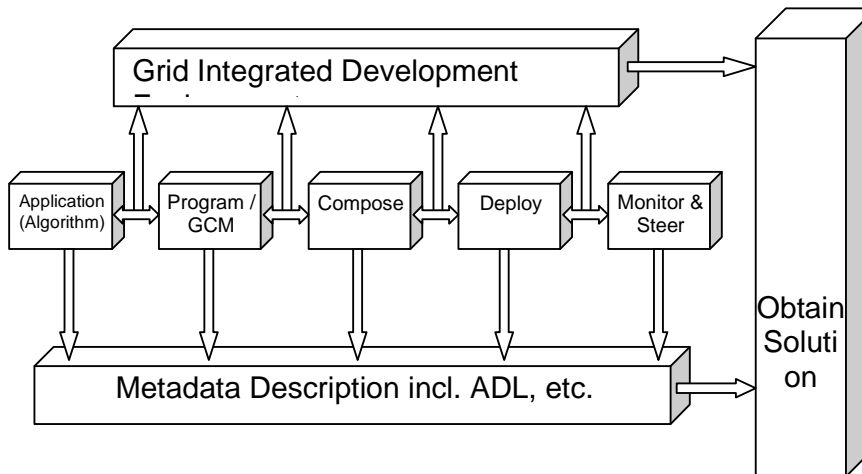


Figure 1: Component-Based Program Development Pipeline

This Grid Interactive Development Environment (GIDE) will support the assembly of Grid components from existing off-the-shelf components. The GIDE is being developed as a series of Eclipse plug-ins [12], and is based on the Grid Component Model (GCM) [13] specification for the representation of components. GCM is an extension of a widely adopted component model, Fractal [2], which allows the hierarchical composition of component-based applications, i.e. components can comprise of simpler sub-components. The GCM implementation used is the one developed in WP2 and provided through the ProActive middleware.[9][13]

Support from GIDE comes at four distinct stages at the lifecycle of a component-based application: development, deployment, monitoring, and steering. Specific objectives for each of these stages follow.

At **development**, the main goal is to significantly reduce development times by providing such tools as component repositories for easy re-use and sharing of components; intuitive visual representation of application architecture; automatic source code generation; legacy code wrappers; and import/export from/to Architecture Definition Language (ADL) files –that standard means for describing application architecture in Fractal/GCM. Following the early prototype implementation and our initial experiences, the objectives set for the 2007-2008 period included the finalization of the Eclipse model that would provide the foundations of

GIDE, the finalization of the development perspective, and further development support with a series of helpful tools.

For **deployment**, the main objective of UoW was the verification of correct integration between the IC2D[14] plug-in and the GIDE (for application deployment) and the need for potential extensions.

For **monitoring and steering** (during execution) the objectives of UoW were to produce a prototype of the monitoring perspective, and to set the roadmap for finalizing monitoring support. Partner TU's objectives were to provide an extended prototype of a Node Resource Monitor.

Regarding **node resource monitoring**, at execution, the application programmer has to be able to visualize his/her components, the hosts on which they execute, and their status. As a complementary feature, one can also take advantage of computational mobility and components migration in order to dynamically and graphically change the location of execution of components, and re-deploy them on a new set of machines. To achieve this, the monitoring of the resource information of the machine is needed plus to the component behaviour monitoring infrastructure. The objective of this Node Resource Monitor is monitoring the resource consumption information of the machine.

Apart from these specific requirements, it was decided that the packaging of the GIDE must come in the form of regular Eclipse plug-ins that are easy to install and maintain by developers. Testing was carried out through application to use cases.[4][10][11]

1.2 Design

The design of the GIDE was based on fulfilling all the requirements for WP4 from the Description of Work Document. An analysis of the requirements led to the high level block diagram as depicted in Figure 2. The aim of the GIDE is to abstract the middleware and platform dependent features as much as possible. In doing so we reduce the learning curve as well as the development and debugging effort, which can often be prohibitive when developing in distributed and grid environments. The GIDE was designed with two different user groups in mind: application developers and data centre operators[5]. For the application developers we provide support for developing through graphical composition, but ensure that the developer always has access to the middleware functionalities and source-code based development if required. We recognise that access to lower level functionalities is sometimes necessary for debugging and improving program efficiency. It also helps to avoid the cases where applications become bulky and inefficient if development is forced to adhere to using only pre-built components. Additional tools are also necessary to enable easy deployment and monitoring of both component status and resource usage to facilitate the development process.

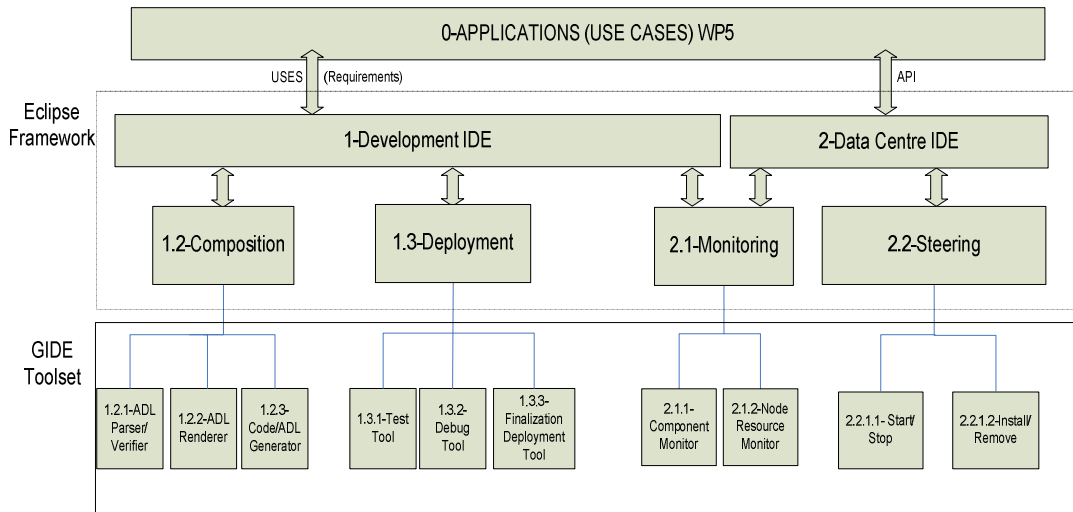


Figure 2: GIDE Block Diagram

Such monitoring functionalities are also applicable to a data centre environment. By repackaging the GIDE functionality, a simplified tool for installing, monitoring and mapping necessary component code to available resources can be provided for a Data Centre environment. However, the framework [7] must provide additional support for steering to enable the data centre operators to install, remove, and upgrade to new versions of component code if required. Data centres have high turnover rates. Hence there is a need for a simple design that would facilitate fast handovers, and enable operators to assist newcomers in coming to terms with the application quickly. The Data Centre version will be deployed once all the functionality of the development has been finalised.

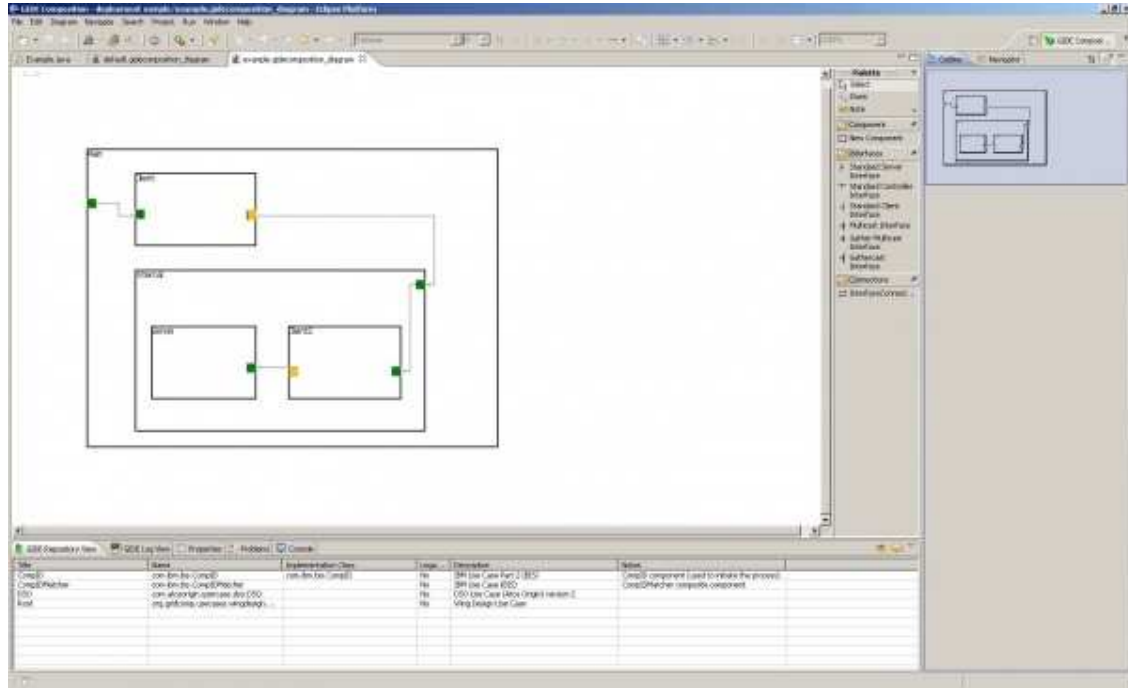
The GIDE is built on the Eclipse framework leveraging the facilities provided by the Eclipse Modelling Framework (EMF) and the Graphical Modelling Framework (GMF). A detailed discussion of these frameworks is outside the scope of this document, but excellent resource material is available from <http://www.eclipse.org/modeling/gmf/> for the interested reader. Choosing Eclipse guards from obsolescence and enables seamless customisation and extension through its plug-in architecture. It is a leading Java development environment, and allows easy integration with many libraries, including the ProActive middleware libraries being used for the CFI. In addition, it provides access to countless other plug-ins available online from other developers. The development environment has been distributed as a set of plug-ins. The GIDE provides its main functionalities for composition, deployment, and monitoring as different so-called perspectives – an Eclipse specific method for grouping a set of functionalities as a graphical view. For the data centre operators, we intend to deliver a standalone application as a Rich Client Platform (RCP) application. [8]

The following is a detailed description of the functionalities within each of the main perspectives, editors and views of the GIDE.

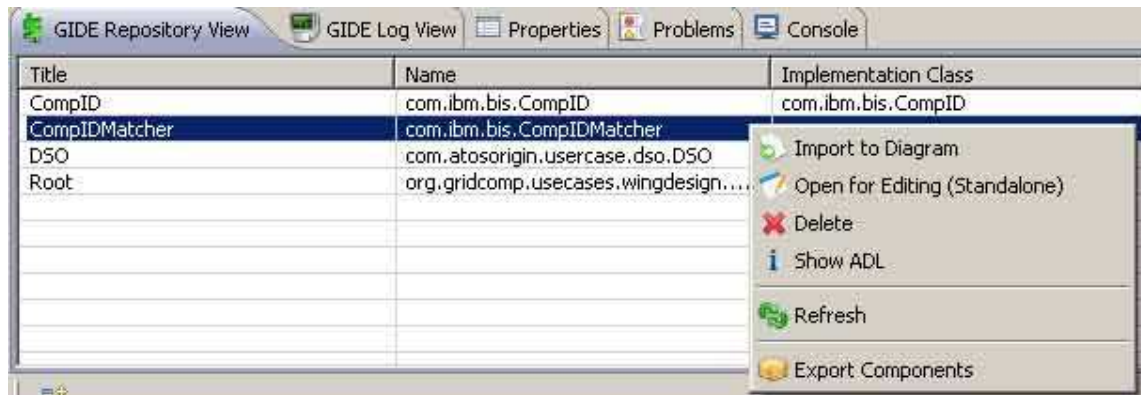
1.3 Composition Perspective

The composition perspective represents the part where developers will spend most of their time. It consists of the main composition editor, a collection of supporting views, and a variety of tools to assist in the composition of distributed applications, importing/exporting from/to ADL files, component source code management, legacy code support[6], behavioural

skeletons support, sharing and reusing of existing components, and various other development related functions.



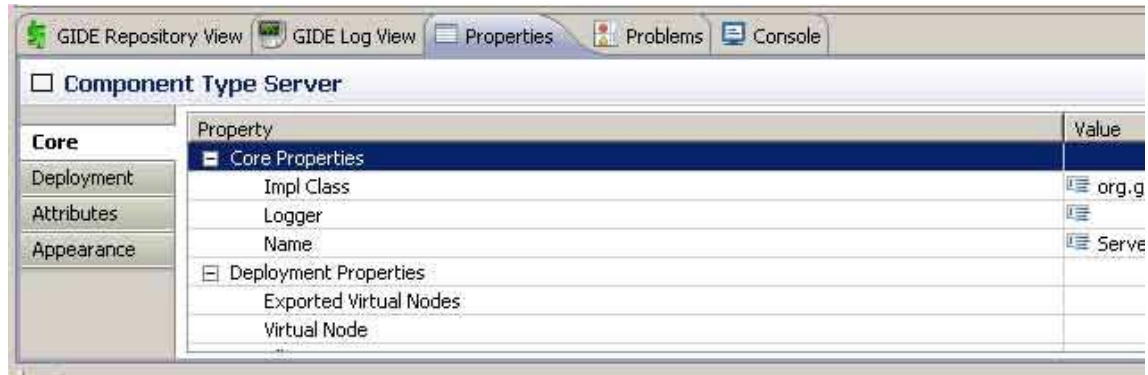
1.3.1 Component Repository View



The component repository holds stored components along with their associated ADL files, source code files, semantic and diagram files. The repository is stored in simple XML format in the GIDE distribution root folder and allows the re-use, manipulation, and sharing of components. In more detail, the user can store an existing composition (whether it is a simple primitive component, a composite component, or even a whole application) into the repository for future usage. Such repository components can then be imported into other compositions. This allows the user to develop an application in convenient parts and then assemble it hierarchically from existing pieces. Stored component can be manipulated in a variety of ways: apart from the typical introspect, edit, and delete, the user can also associate

source code files with the stored components, e.g. the implementation of a primitive component or the source code for an attribute controller. Finally, an important feature of the repository is the ability to export and import collections of components into a single archive for distribution and sharing between multiple GIDE instances. Each component in the archive will be stored complete with its ADL description, source code files, semantic model, and diagram model.

1.3.2 Properties View

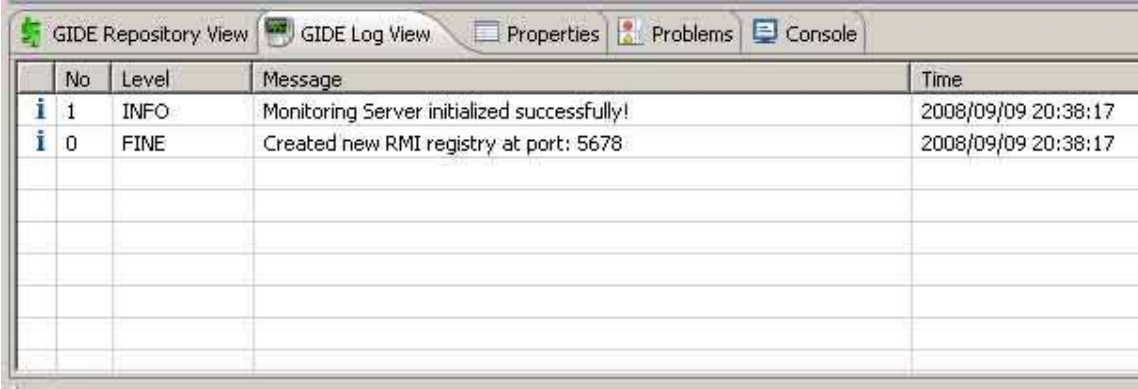


The properties view displays the properties of the currently selected composition item, including its GCM properties, appearance properties, and various other project specific properties.

Among the GCM properties available for components are its name, implementation class, logger name, virtual node, exported virtual nodes, and attributes, as per the GCM specification. A number of miscellaneous properties such as author and comment, as well as appearance properties such as font size and typeface, are also available.

Similarly, interfaces have the following GCM properties: cardinality, contingency, implementation class, name, and the read-only role property; and a number of miscellaneous other properties such as an optional comment, and id.

1.3.3 GIDE Log View



No	Level	Message	Time
1	INFO	Monitoring Server initialized successfully!	2008/09/09 20:38:17
0	FINE	Created new RMI registry at port: 5678	2008/09/09 20:38:17

The GIDE Log view is available in all GIDE perspectives, although composition is arguably the one where it is the most useful. It provides an independent view dedicated solely to reporting informational, debug, and error messages to the GIDE user.

The GIDE Log view provides a configuration facility to filter reported messages to the user based on their type or severity and the number of displayed items, while it also supports exporting and importing log files, a function useful in the case of bug reports.

1.3.4 Editor

The editor is where all the graphical component composition takes place. It consists of a composition canvas and a palette of available items. The user can drag and drop standard items from the palette to the canvas (components and interfaces) and use the interface connection tool in order to bind client to server interfaces and thus create the required application architecture.

Importing existing components into the diagram through the repository view is another helpful option: the user can develop his application in pieces (for clarity and ease of maintenance) and can then merge them all together to assemble the final application composition.

The editor makes available a number of tools through the context menu, namely support for adding and editing legacy component wrappers and behavioural skeletons and automatic generation and manipulation of component source code (for primitive components, server interfaces, and attribute controllers).

Legacy component wrappers enable the developer to incorporate legacy code in the component-based application. The wrapper is simply a template where the developer sets a number of required legacy parameters, including the path to the executable, any command line arguments necessary and the input file permissions. For more details on legacy code wrapping please refer to D.CFI.04.

Support for behavioural skeletons follows the same logic as for the legacy code wrappers. At design level skeletons are provided in the form of different design templates depending on the

type of the skeleton, and the developer then specifies the skeleton parameters. At the lower level each skeleton is merely a hierarchy of interdependent ADL templates, associated with a number of standard source code files.

Currently two behavioural skeletons are supported: the Data-Parallel and the Farm skeletons. Nevertheless, the logic behind skeletons is very similar and other types of skeletons can be easily supported in the future.[1]

The final major tool available through the editor is the automatic source code generation and manipulation. For each component in the composition a number of source code files might be required before moving to the next phase of deployment. For example, for primitive components, the implementation class is required, while for components with server interfaces the interface files are needed. The GIDE code generation tools attempts to abstract away from the developer the lower level source requirements of the GCM model (interface bindings, attribute controllers etc.) by providing the necessary code automatically. For instance, in the source file of a component that contains client interfaces, a binding controller interface must be implemented that enables the introspection and acquisition of each of the client interfaces. The following listing provides a simple example of a component “ClientImpl” with a single client interface, “service” of type `org.gridcomp.gide.component.examples.Service` (all generated helping comments have been removed for clarity).

```

import org.objectweb.fractal.api.NoSuchInterfaceException;
import org.objectweb.fractal.api.control.IllegalBindingException;
import org.objectweb.fractal.api.control.IllegalLifecycleException;

public class ClientImpl implements
org.objectweb.fractal.api.control.BindingController{
    private org.gridcomp.gide.component.examples.Service service;

    private static final java.lang.String SERVICE_CLIENT_INTF = "service";

    public void bindFc(String myClientItf, Object serverItf) throws
NoSuchInterfaceException, IllegalBindingException,
IllegalLifecycleException {
        if (myClientItf.equals(SERVICE_CLIENT_INTF)) {
            service = (org.gridcomp.gide.component.examples.Service)
serverItf;
        }
    }

    public String[] listFc() {
        return new String[] { SERVICE_CLIENT_INTF };
    }

    public Object lookupFc(String itf) throws NoSuchInterfaceException {
        if (itf.equals(SERVICE_CLIENT_INTF)) {
            return service;
        }
        return null;
    }

    public void unbindFc(String itf) throws NoSuchInterfaceException,
IllegalBindingException, IllegalLifecycleException {
        if (itf.equals(SERVICE_CLIENT_INTF)) {
            service = null;
        }
    }
}

```

```
}

```

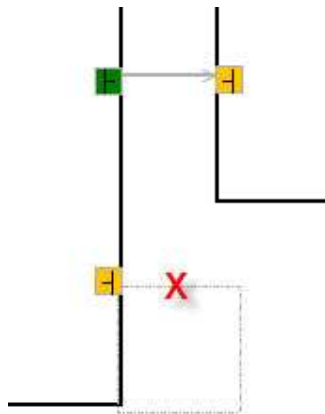
The user can choose to have the complete implementation source file generated, or just a relevant sub-part that he can then integrate to an existing source file. The following figure illustrates these options.



The user can always open and edit any associated source files for a component through the editor context menu as shown above.

1.3.5 Validation

The GMF Framework provides facilities for runtime validation during the development process. As component diagrams are developed, a set of OCL constraints will stop the developer from creating invalid diagrams. One example is to not allow self loops, i.e. an interface can not connect to itself.



More constraints, such as limiting the connection of client to server and server to client interfaces, as well as limiting connections to one level of nesting, are being implemented.

1.4 Deployment Perspective

The deployment perspective provides support for the deployment of an application using independent architecture and application descriptors. Architecture descriptors provide information about the available infrastructure resources while application descriptors clarify the requirements of an application.


```

private static File desc = null;
private static Node[] nodes = null;
private static HashMap<String, Object> context = null;
private static Factory factory = null;

public static void main(String args[]){

    try{

        /**
         * @todo edit (if necessary) the deployment descriptor location
         */
        gcma = null;
        context = null;
        nodes = null;
        desc = new File("GCMD_app.xml");

        deploy();

        //@todo: execution code

        //undeploy();

    } catch (Exception ex){
        ex.printStackTrace();
    }
}

public static void deploy(){

    try{

        boolean deploying = false;
        if (gcma == null) {
            gcma = PAGCMDDeployment.
                loadApplicationDescriptor(desc.toURI().toURL());
            gcma.startDeployment();
            gcma.waitReady();

            /**
             * @todo edit the virtual node name
             */
            GCMVirtualNode vn = gcma.getVirtualNode("<default node>");
            nodes = vn.getCurrentNodes().toArray(new Node[] {});
            (context = new HashMap<String, Object>(1)).
                put("deployment-descriptor", gcma);

            deploying = true;
        }

        if (deploying) {
            factory = FactoryFactory.
                getFactory(FactoryFactory.FRACTAL_BACKEND);

            /**
             * @todo edit (if necessary) the location and name
             * of the root component ADL description
             */
            Component root = (Component)factory.
                newComponent("<root ADL location>", context);
        }
    }
}

```

```

/**
 * @todo uncomment if you want to start
 * the root component at this point
 */
//((LifeCycleController)root.
    getFcInterface("lifecycle-controller")).startFc();

/**
 * @todo uncomment and edit the name of the
 * initiating interface and (if necessary) the related
 * interface method
 */
//((<initiating interface>)root.
    getFcInterface("<initiating interface>")).main(null);
}

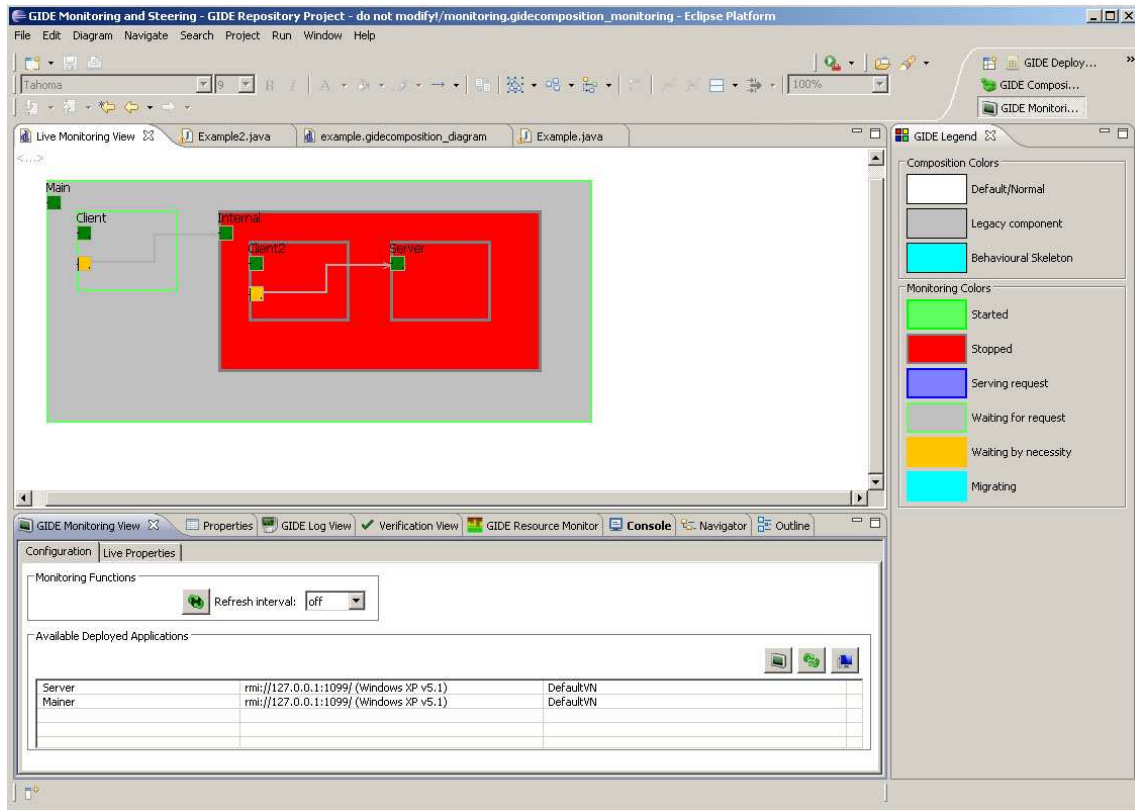
} catch (Exception ex){
    ex.printStackTrace();
}
}

public static void undeploy() throws Exception {
    if (gcma != null) {
        Registry.instance().clear();
        gcma.kill();
    }
}
}
}

```

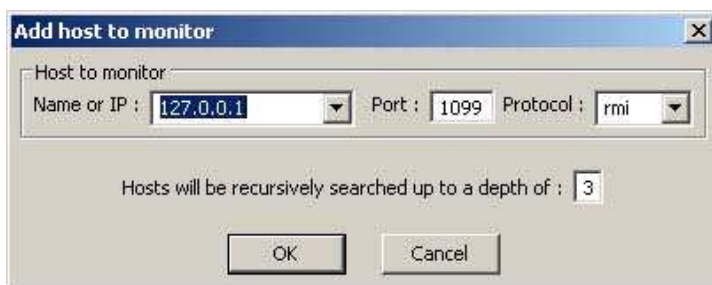
1.5 Monitoring and Steering Perspective

The Monitoring and Steering Perspective is used during runtime to dynamically monitor a running application, and allow the user to perform some manual runtime reconfiguration (steering) -aside of the autonomic adaptations [3] that might take place during runtime. Live monitoring and steering is performed with the use of a live monitoring canvas very similar to the composition canvas (all shapes are equivalent).

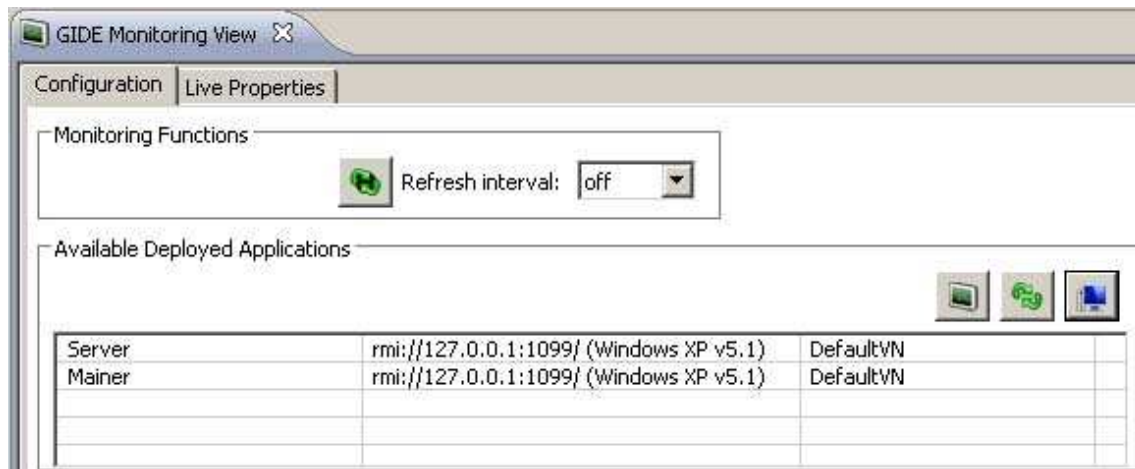


1.5.1 Monitoring View

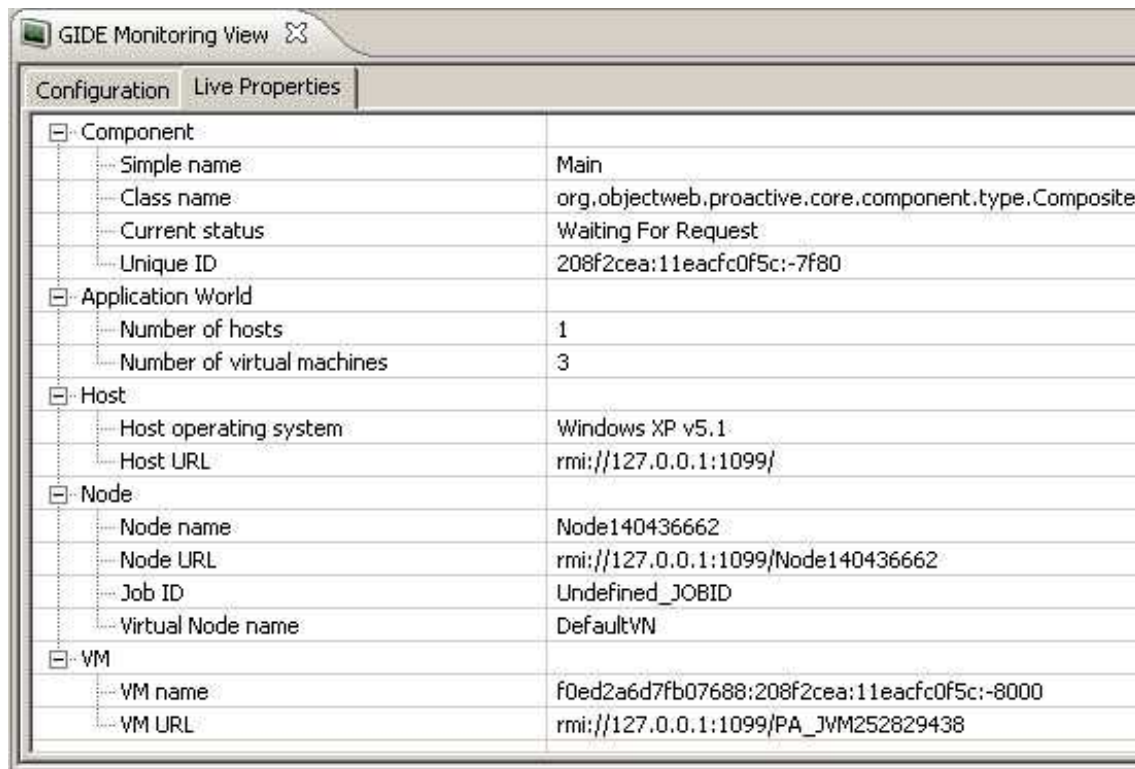
The main view in the monitoring perspective is the monitoring view which enables the user to select a remote host for introspection. The user must provide the name or IP address of the remote host, the relevant port, and the communication protocol used to start the remote GCM process.



Upon introspection of the host, the GIDE will identify any running GCM applications and will fill in the table with those (as illustrated in the following figure).



The user can then select any of the available running applications for monitoring. Updates to the running application will be signaled by an underlying notification system (each component notifies GIDE for any changes it endures). A secondary update mechanism is also available in the form of immediate refresh and periodic polling (in user-defined intervals). A second tab in the monitoring view, shows the runtime information for the selected component. This information ranges from application-wide details (such as the total number of hosts and virtual machines), to node specific details (such as name, URL, and operating system) and refined component-specific details (such as its name, status, and id).



1.5.2 Live Monitoring Canvas

The live monitoring canvas will pick-up any changes to the runtime architecture of the application, both manual (when full steering functionality is available) and automatic (for

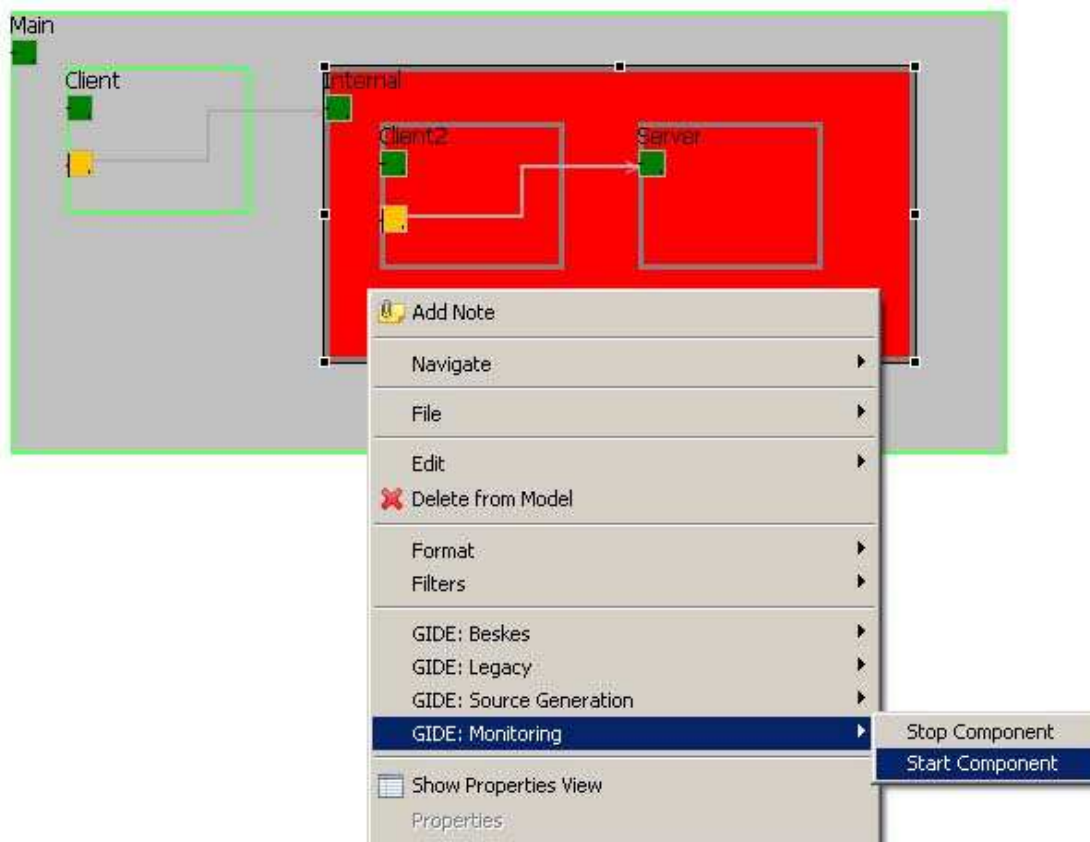
example due to load balancing with the addition of extra workers). Standard colors are used to identify the different states of components, e.g. green for live components, red for stopped components, etc.



1.5.3 Steering

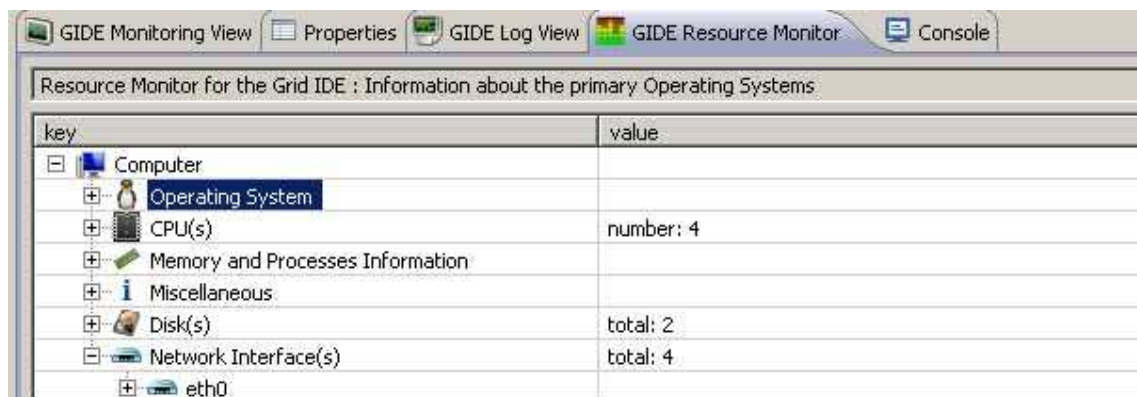
Basic steering functionality is also available through the live monitoring canvas. The user has the option to start or stop a particular component (either primitive or composite), while a number of other features will be made available soon, such as adding extra worker components, rearranging interface bindings and more.

Stopping a hierarchical component through the GIDE monitoring canvas, delegates the request to the GCM component which will send stop requests recursively to its internal components before eventually stopping itself. The reverse process takes place when starting a hierarchical component.



1.5.4 Resource Monitor View

The monitoring perspective also includes a resource monitor view that gives information about the current host, with the ultimate goal of providing resource metrics on distributed hosts as well.



2 Manual

2.1 Setting Up

2.1.1 Eclipse Project Version

It is highly recommended to download and install Eclipse Ganymede with Modelling Tools (around 300MB in size) which comes prepackaged with all required GMF Modelling plugins.

The following sections apply to the Eclipse Ganymede version with Modelling Tools.

2.1.2 Behavioural Skeletons Rules file Editor

The JBoss Drools project provides an Eclipse plugin, an editor with syntax highlighting for editing rules files to be used with Behavioural Skeletons. You can get the plugin here: <http://www.jboss.org/drools/downloads.html>. Select one of the Eclipse plugins depending on your Eclipse version.

2.1.3 IC2D plugins for GIDE Monitoring

Due to the restructuring of the GIDE Monitoring mechanisms, a dependency on the IC2D has now been introduced. However, because of some inconsistencies in the latest IC2D released plugins and the requirements of GIDE, slightly customized IC2D plugins are provided with the GIDE distribution.

Further, since IC2D required the BIRT plugin in order to work correctly, users must update Eclipse and install BIRT with the following simple steps:

- Start Eclipse Ganymede (similar steps for Europa and other versions) and go to **Help** → **Software Updates** → **Available Software** and expand the **"Ganymede"** or **"Ganymede Project Updates"** site (if it is not there click refresh and wait for the site to appear)
- Expand **"Charting and Reporting"** and select all available BIRT plugins
- Choose **"Install"**
- When asked, accept the license agreement and continue
- Make sure the process completes successfully and restart Eclipse when prompted.

2.1.4 Installation

- Download the GIDE package from <http://perun.hscs.wmin.ac.uk/dis/gide/wiki/index.php/GIDE:Download>

- Unzip it in a folder of your choice (e.g. C:\gide)
- Edit eclipse.ini from the eclipse root folder and add after -vmargs in separate lines the following bold-faced lines (where "C:\GIDE" is whatever your GIDE root folder is):


```

      -showsplash
      org.eclipse.platform
      --launcher.XXMaxPermSize 256M
      -vmargs
      -Dcom.sun.management.jmxremote
      -DGIDE_HOME=C:\GIDE
      -Djava.security.policy=C:\GIDE\policies\gide.policy
      -Dosgi.requiredJavaVersion=1.5
      -Xms512m
      -Xmx512m
      
```

The eclipse.ini file is located under <eclipse_home> in Windows and Linux systems, and under <eclipse_home>/Eclipse.app/Contents/MacOS in MacOS systems.

- Delete any previous versions of the plugins from the eclipse/plugins directory. Do not rely on overwriting them because the version numbers may have changed!
- Copy the GIDE plugins (and the modified IC2D plugins) in the eclipse/plugins directory. If your GIDE plugins directory only contains one or two zip/rar files, it means that all plugins are archived inside them. In that case you will need to extract the archived plugins into the eclipse/plugins directory
- Start eclipse

2.1.5 Directory and File Structure

Inside both versions there are the following folders:

- **descriptors**: the root folder for the GIDE deployment descriptor repository
- **doc**: documentation files
- **dtd**: the default ProActive DTD file
- **libs**: libraries that are needed for GIDE to function properly
- **logs**: folder that contains all session log files
- **plugins**: the folder that contains the GIDE Eclipse plugins
- **policies**: policy files required by some internal GIDE components
- **repository**: the root folder for the GIDE component repository
- **temp**: temporary folder for various GIDE purposes
- **templates**: contains the template ADLs, model, and diagram files for the Behaviourl Skeletons and the Legacy component wrappers
- **userlibs**: this folder contains GIDE jar files that might be used by developers if they wish to make use of the current GIDE Monitoring tool

2.2 Basic Usage

2.2.1 Starting Up

With the latest plugin version there are no explicit startup procedures as all GIDE tools and perspectives are readily available through Eclipse. Just follow the instructions under "Using GIDE Tools and Perspectives".

2.2.2 Using GIDE Tools and Perspectives

There are a number of available tools specific to GIDE, ranging from Composition diagrams, import and export facilities, resource monitoring and more. This is only a quick start guide for the most common functionality. For a detailed step by step guide have a look at the following sections and the Tutorials section in the Wiki (<http://perun.hscs.wmin.ac.uk/dis/gide/wiki/index.php/GIDE:Documentation#Tutorials>).

- To create a new composition, you must already have a project in your workspace and then go to **File** → **New** → **Example** → **Gidecomposition Diagram**.
- To change the appearance of the composition diagram, right click anywhere on the empty canvas (not inside any components) and select **Show Properties View** → **Rulers and Grid** to set the Grid and Rulers appearance or select **Show Properties View** → **Appearance** to set the font and colors.
- To show the properties of the composition items (components, interfaces, bindings) select the component you want and its attributes will be loaded in the Properties View Panel. If the latter is not currently visible or in focus, right click on the composition item of interest and select **Show Properties View**
- To import an existing ADL into a new composition go to **File** → **Import** → **Other** → **Import ADL** and follow the Wizard instructions
- To export a composition diagram into one or more ADL files go to **File** → **Export** → **Other** → **Export to ADL** and follow the Wizard instructions
- To view the GIDE log for the current session go to **Window** → **Show View** → **Other** → **GIDE** → **GIDE Log View**
- To view the GIDE Component Repository go to **Window** → **Show View** → **Other** → **GIDE** → **Repository View**
- To move into the GIDE: Composition perspective and have all related views arranged by default (including the Log and Repository views) go to **Window** → **Open Perspective** → **Other** → **GIDE Composition** (same procedure for the GIDE Deployment and GIDE Monitoring and Steering perspectives)
- To import a component from the repository you must already have a composition diagram open and in focus. Then, select the component you want from the repository view, right click and select 'Import'.

- To generate source files for primitive components, server interfaces, or attribute interfaces, go to the relevant composition item, right click and go to the GIDE context menu to see all available options
- To edit a component in the repository go to the repository view, select the component, right click and select **Edit**
- To add a legacy component wrapper in the existing diagram right click on the existing component where the legacy wrapper will be added to (or the canvas) and select **GIDE: Legacy → Add Legacy Component Wrapper**
- For an application to be available for monitoring the jar file ProActive_Extensions.jar must be included in the classpath of the deployment project. It is located under GIDE_HOME/userlibs. Then, from the Monitoring view, you can select a host to start monitoring for deployed applications. The table will then be filled with all applications running on the current host. Click on the "prepare for monitoring" button to start monitoring. Once the live monitoring canvas shows up the underlying notification system will automatically update it with any changes that happen. You can also use the refresh now button, or set the refresh interval to a valid option to enable a polling type of refresh.
- To add a Data-Parallel Behavioural Skeleton in the existing diagram right click on the existing component where the skeleton will be added to (or the canvas) and select **GIDE: Beskes → Add Data-Parallel Behavioural Skeleton**. The system will then ask for the skeleton parameters to be set, i.e. select a worker component for the skeleton, a rules file path, and a descriptor path. Adding a Farm Behavioural Skeleton requires a very similar process.

2.3 Composition

2.3.1 Importing ADL files

2.3.1.1 General

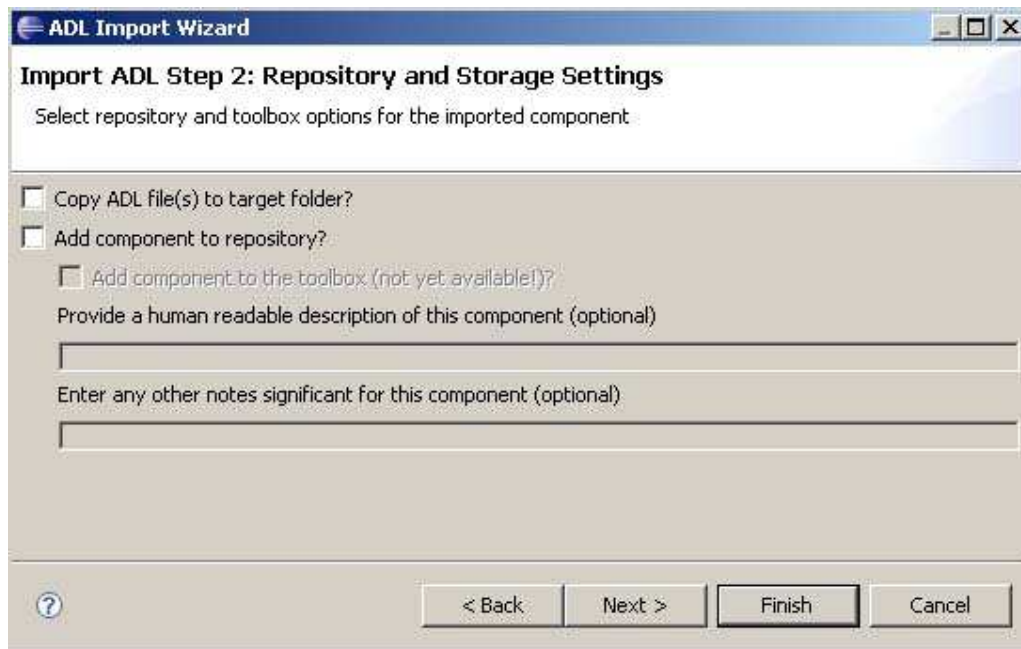
Current requirements for importing: if the top level definition name is package-qualified then the file must be located in a directory structure that reflects the specified package. Any referenced files must be either in the same directory or in their respective directory hierarchy (if on their own they are package qualified) in either the same root directory or in another directory that is accessible through the classpath.

- Any referenced DTDs must be available through http or file URLs, as absolute filenames, or as relative filenames that are available through the classpath. It is the responsibility of the developer and not the GIDE, to provide a DTD in his ADL files that is accessible. However, the GIDE will resort to a default location in the installation directory where it looks for the default DTD (for the GIDE purposes this is the ProActive DTD).
- When adding to the repository only the imported component is added – not its internal components independently. This is on purpose in order to avoid cluttering and keep things simple. If the user wants to add some of the internal components separately he can do so by importing those separately.

- When importing a component and an identical name already exists in the repository the user will have the option to provide an alternative name. In such case, and after any potential name conflicts have been identified and alerted the user, the repository and the imported component have the new name. However, the locally copied ADL files (if the user opted to copy locally) will be the *original* ADL files.
- The option to add to the toolbox an imported component has not been implemented yet

2.3.1.2 Wizard Options

- Select **File** → **Import** → **Other** → **Import an ADL**.
- In the first page, use the "Browse" button or type in the location of the top level ADL file you wish to import. All dependencies will be resolved automatically.



- In page 2 (see Figure) there are a number of options relating to the import:
 - **Copy ADL file(s) to target folder:** check this to have the ADL files copied to the target import folder in a hierarchical directory structure following the ADL names, e.g. a component with name "org.gridcomp.examples.SomeComponent" will be copied to a folder \org\gridcomp\examples inside the import target folder selected.
 - **Add component to repository:** check this to have the top level (and only the top level!) component of the supplied ADL file added in the GIDE repository for future usage. There will be only one component added to the repository, the top level one with all its potential subcomponents, but there will be no recursive addition of the internal subcomponents as separate repository entries.

You can leave this option unchecked if you are testing some ADLs or you are currently exploring the GIDE.

- **Add component to the toolbox:** this is not yet available functionality. A toolbox with all available GIDE repository components will be graphically available in the GIDE composition perspective that will allow users to quickly import components in the current composition diagram. This option is only available if the component will be added to the repository.
- **Descriptions:** you can optionally provide a human readable description as well as other notes for the imported component. This option is only available if the component is to be added to the repository.

2.3.2 Exporting to ADL files

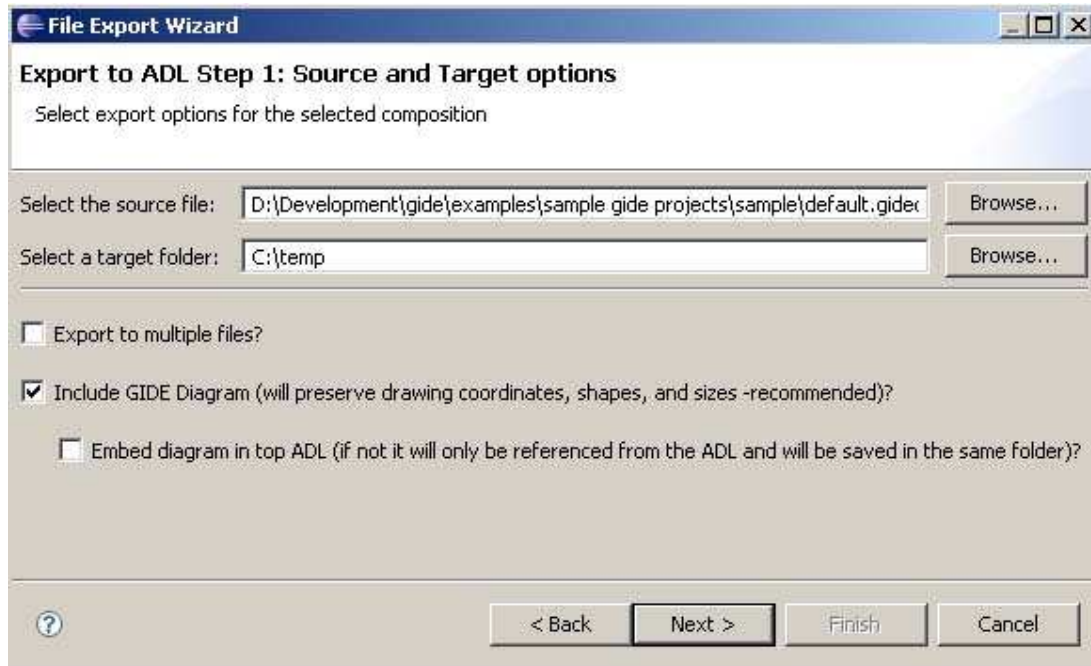
2.3.2.1 General

General Naming Requirements: top level canvas **MUST** have a name. All components and interfaces must also have names.

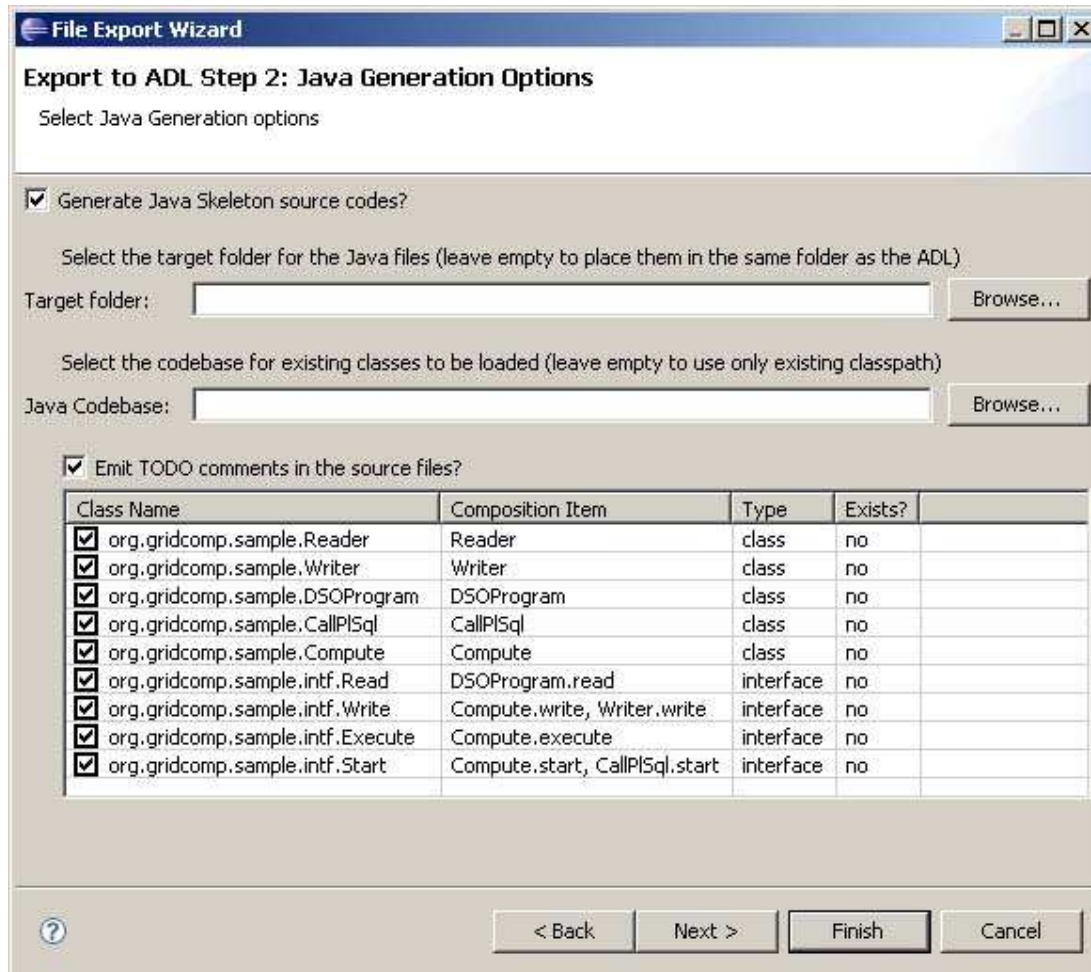
- Component/definition names may be package qualified in the composition. The package dictates the directory hierarchy where the resulting ADL will be placed. If a sub component is not package qualified, it is assumed to inherit the package from its parent. If it is package qualified, it will be placed in this potentially different directory hierarchy, nevertheless in the same export root that the user selects
- When exporting to an ADL, the potentially different packages of a component will only be significant when splitting the files. Otherwise it might not even be possible to determine the difference.
- If one or more components in the composition have either an "extends" or an "arguments" ADL attribute, then the "Export o multiple files" option is enforced

2.3.2.2 Wizard Options

- Select **File** → **Export** → **Other** → **Export to ADL file(s)**.



- Use the "Browse" button or type in the location of the source GIDE composition diagram file you wish to export.
- Use the "Browse" button or type in the target location for the exported ADL file(s).
- **Export to multiple files:** check this option to export the composition to multiple ADL files, one for each (sub) component. The files will be organized in a hierarchy based on their names: package qualified names will go into the equivalent directory hierarchy inside the target folder, e.g. a component named `org.gridcomp.examples.SomeComponent` will go inside a folder `org/gridcomp/examples/` inside the target folder selected. Components with simple (i.e. non package qualified) names, inherit their parents packages. Note: if this option is unchecked, only the top level package (if one exists) will be used to determine the directory structure.
- **Include GIDE diagram:** select this option to export the GIDE diagram file as well. This will preserve the drawing, shapes, sizes, arrangements etc. so that if the exported ADL(s) are imported again (in the same or another GIDE) the diagram will be preserved. This is highly recommended.
- **Embed diagram in top level ADL:** if the GIDE diagram is to be included, it can be done in two ways: through a reference from the top level ADL to an external diagram file (which will have to be moved along the ADL(s)) or through embedding the whole GIDE diagram in the top level ADL in the form of a special comment recognizable from the GIDE parsers. The latter option has the benefit of not requiring a separate file to be maintained but it "litters" the top level ADL. This option is only available if the "Include GIDE diagram" option has been selected.



- **Generate Java Skeleton Code:** the user has the option to generate Java skeleton files for the specified implementation and interface classes (if any were specified in the composition). All subsequent options are only available when this option has been selected.
- **Target folder:** optionally create the Java files in a different folder than the ADL target folder specified in the previous step.
- **Java Codebase:** the GIDE will make an attempt to discover existing classes in the current classpath in order to generate skeleton methods for classes implementing already existing interfaces. However, the user can also specify an extra implementation codebase folder where existing classes not available in the current classpath can be found and loaded. If none of the specified classes can be found the Java generator will merely create the directory and file hierarchy and provide the class header lines, e.g.

```
package org.gide.composition.somePackage;
/**
 * Header JavaDoc
 */
```

```
public class SomeClass implements
org.gide.composition.SomeInterface {
}
```

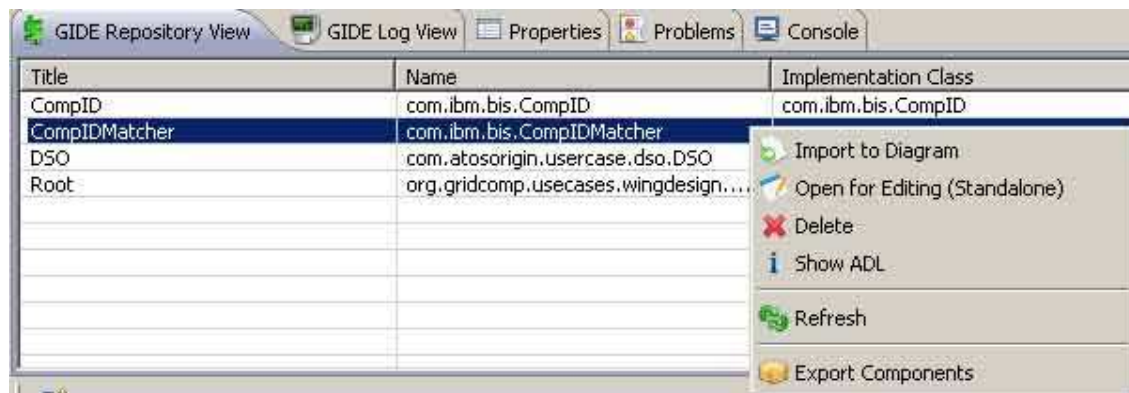
However, if one of the specified interfaces can be found (either in the classpath or the specified codebase) any implementation source files implementing this interface will be filled with sample methods, as in the following example which assumes that one of the composition components was implementing the org.xml.sax.EntityResolver

```
package org.gide.composition.somePackage;
import org.xml.sax.InputSource;
/**
 * Header JavaDoc
 */
public class SomeClass implements org.xml.sax.EntityResolver {
/**
 *
 * string_1 String
 * string_2 String
 */
public InputSource resolveEntity (String string_1, String
string_2){
/**
 * @todo Fill in this method's logic code!
 */
return null;
}
}
```

- **Emit TODO comments:** check this option to have TODO Javadoc comment generated for each class and each method (if any) in the class.
- **Names Table:** this table shows which implementation class names were specified in the composition diagram and the respective composition items. It further shows which of these classes were found in the classpath or in the specified codebase. The user can select which of these to be automatically generated.

2.3.3 Component Repository

The component repository holds stored components along with their associated ADL files, semantic and diagram files.



2.3.3.1 Selection Options

When a repository item has been selected the following options are available:

- **Import to Diagram:** the selected component will be imported inside the currently selected component in the currently opened GIDE diagram. This option is only available when a GIDE diagram is already open.
- **Open for Editing (standalone):** the component will be opened for editing in a separate standalone GIDE editor. Upon finalizing the required changes, the user has two options: "Save" which will overwrite the existing component in the repository; and "Save as..." which will prompt the user to enter a new location and a new name for the component. The latter option will neither automatically save the component in the repository as well nor remove the previous one! For the newly named component to be saved to the repository the "Add/Update Repository" option in the "Save as..." dialog must be checked.
- **Delete:** will first ask for confirmation and if confirmed it will remove the selected component from the repository.
- **Show ADL:** will open a new window with a tree representation of the selected component ADL.
- **Copy source files:** if the repository component has source files associated with it, the user can select to copy them in a workspace project for editing and/or use in a composition
- **Update existing source files:** the user can update the source files associated with the stored component by selecting a directory. The GIDE will then pick up all relevant source files from that directory based on the implementation attributes specified in the stored component definition.

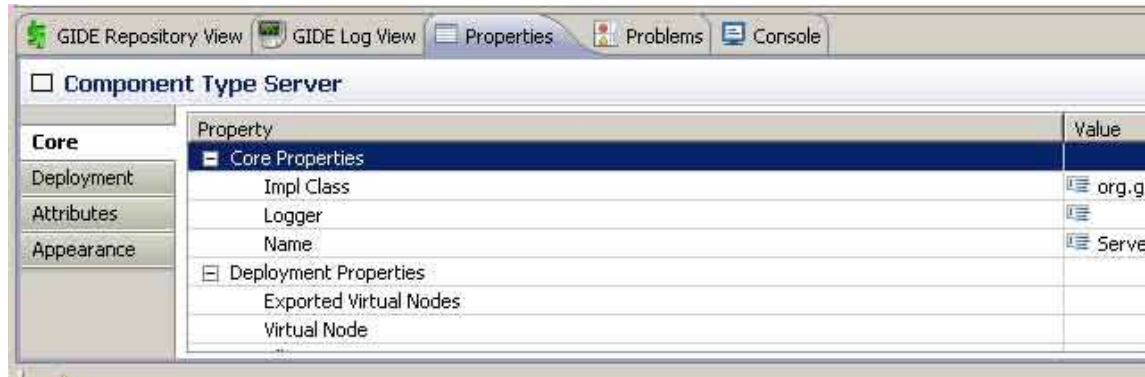
2.3.3.2 General Options

The component repository supports the import and export of a collection of components to enable sharing and reusing between different GIDE instances

- **Export Components:** will create an '.grc' archive in the specified location, with the ADL, semantic, and diagram files of all selected components.
- **Import Components:** will ask for a '.grc' archive file to import its content components to the repository. In case of name conflicts these will be resolved interactively with the user's guidance.

2.3.4 Properties

The properties view displays the properties of the currently selected diagram item, including its GCM properties, appearance properties, and various other project specific properties.



2.3.4.1 Core Properties

This tab contains the core GCM properties of the selected diagram item, as well as some project specific properties.

For components this tab shows the following properties:

- **GCM core properties:** name, implementation class, logger name;
- **GCM deployment related properties (read-only):** virtual node, exported virtual nodes;
- **Miscellaneous properties:** author, comment, extends, controller, id, and the read-only last-modified property;
- **Parameters:** instance arguments list, and formal parameter list.

For interfaces this tab shows:

- **GCM core properties:** cardinality, contingency, implementation class, name, and the read-only role property;
- **Miscellaneous properties:** comment, and id.

For the canvas the following properties are available:

- **GCM core properties:** name;
- **Miscellaneous properties:** author, comment, id, and the read-only last-modified property;
- **Source management properties:** bin, and src; these properties specify the source and class folders for the current project and are used for source management. They can be made relative to the current project (instead of providing absolute path names) by including the tag <project>, for example: '<project>/src'.

2.3.4.2 Deployment Properties

This tab contains the GCM deployment properties and is only available when a component has been selected. It contains the following property groups:

- **Virtual Node:** allows the user to set the virtual node name and cardinality. Changes will not be saved until the 'Apply changes' button is clicked
- **Exported Virtual Node:** provides support for defining the exported virtual nodes as per the GCM specification. Functionality for editing and deleting exported virtual nodes is also available.

2.3.4.3 Attributes Properties

This tab provides support for defining the GCM attributes for the selected component (and is obviously only available when a component has been selected). It contains the following property groups:

- **Attribute Controller:** allows the user to set the signature for this attribute controller and provide an optional comment. Changes will not be saved until the 'Apply changes' button is clicked
- **Attributes:** enables the user to add attributes in the common name/value pair form with an optional comment for each one. Functionality for editing and deleting attributes is also available.

2.3.4.4 Appearance and Ruler Properties

The appearance properties section enables the user to manipulate the font size, type, and face of the lettering and the border lines in the diagram.

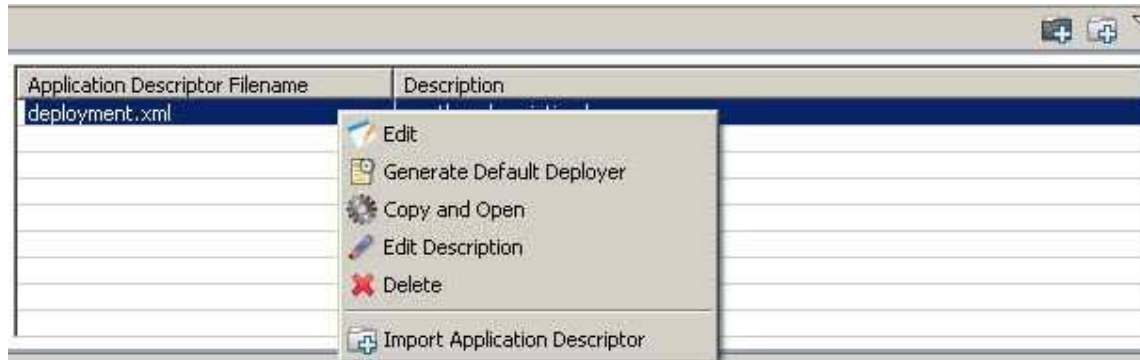
The ruler properties section is only available for the canvas and allows manipulation of the grid, the ruler spacing, and other related properties.

2.4 Deployment

The main view in the deployment perspective is the deployment view which contains the descriptor repository.

2.4.1 Descriptor Repository

The descriptor repository holds stored deployment descriptors along with an optional human-readable description. The repository actually holds two collections of files: architecture descriptors and application descriptors (as per the new GCM Deployment framework).



2.4.1.1 Selection Options

When a repository item has been selected the following options are available:

- **Edit:** opens the selected descriptor for editing
- **Generate Default Deployer:** generates a default deployment class that the user can customize. In more detail: through the application deployment descriptor table the user can have the GIDE automatically generate a default deployment class. This class will contain the default procedure for deployment (i.e. reading the descriptor, creating the component hierarchy, starting execution etc.) which the user must fill-in with his application specific values and customize to fit his needs. There two options available: use the typical ProActive deployment framework, or the new GCM deployment framework.
- **Copy and Open:** allows the user to copy the selected descriptor to a specified project in the workspace and open it for editing outside of the repository. This is useful for example for finalizing template descriptors based on the specific project requirements.
- **Edit Description:** allows the editing of the human-readable description.
- **Delete:** asks for confirmation and deletes the selected descriptor.

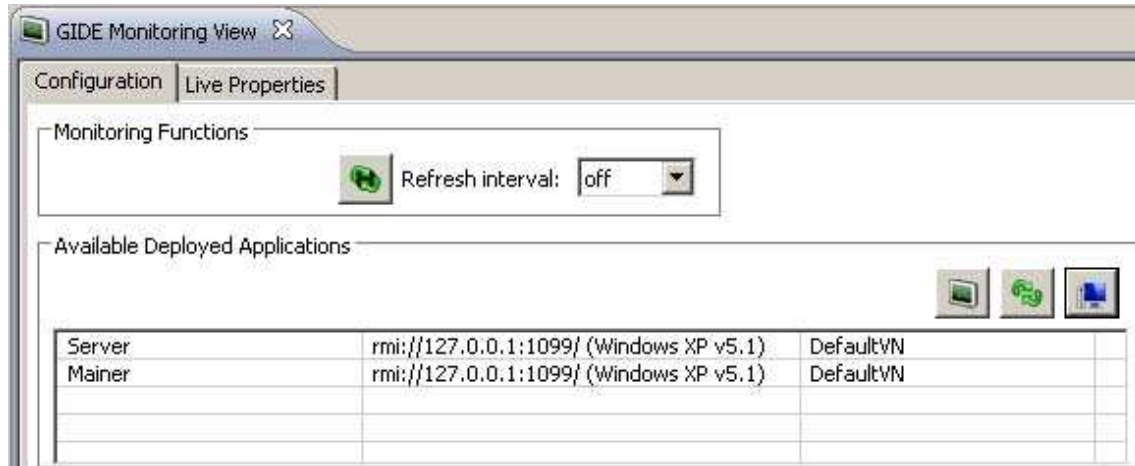
2.4.1.2 General Options

The component repository supports the import and export of a collection of components to enable sharing and reusing between different GIDE instances

- **Import Descriptor:** this option will enable the user to select an existing descriptor file to import in the relevant descriptor repository (architecture or application). An optional human-readable description can also be provided

2.5 Monitoring and Steering

The main view in the monitoring perspective is the monitoring view which contains a collection of applications currently available for live monitoring.



Live monitoring and steering is performed with the use of a live monitoring canvas very similar to the composition canvas (all shapes are equivalent). The canvas is however, read-only in most aspects except moving and resizing the depicted components.

2.5.1 Architecture Monitoring

The monitoring view contains a table of currently available applications for monitoring in the selected host. To find deployed applications for monitoring the user must select a new host to monitor (IP address, port, and communication protocol). The table will then be filled in with any applications running on the selected host. For this functionality to work correctly, the `ProActive_Extensions.jar` must be included in the application's classpath. This file is available in the `userlibs` folder of the GIDE distribution.

To initiate monitoring of a running application select the application from the table and click the “prepare for monitoring” button. Once the live monitoring canvas shows up with the selected application depicted, the underlying notification system will automatically update it with any changes that happen. You can also use the refresh now button, or set the refresh interval to a valid option to enable a polling type of refresh.

The live monitoring canvas will pick-up any changes to the runtime architecture of the application, both manual (when full steering functionality is provided -see below) and autonomic changes (for example due to load balancing with the addition of extra workers).

The Monitoring view contains a second tab which displays the properties of the selected monitored component. These properties include the name, implementation class, virtual node name of the component, as well as information about the host and the application as a whole.

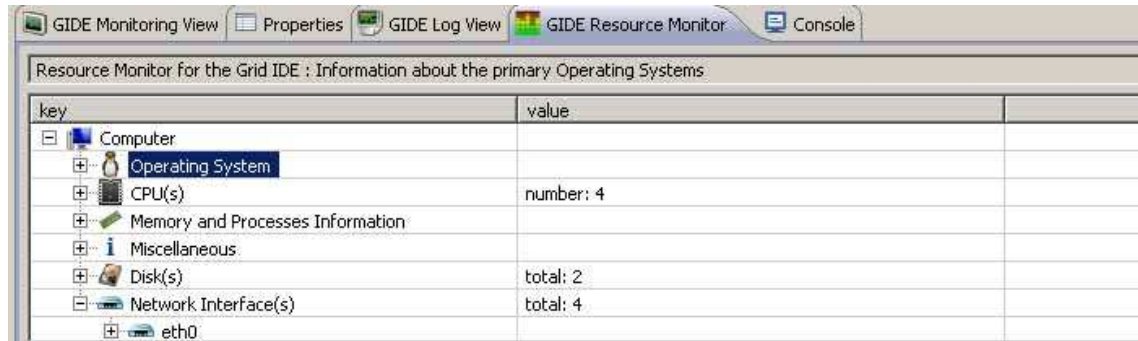
2.5.2 Steering

Basic steering functionality is currently available through the live monitoring canvas.

The user can right-click on the desired component and select either **GIDE: Monitoring** → **Start Component** or **GIDE: Monitoring** → **Stop Component** to try and start or stop the component respectively.

2.5.3 Resource Monitor

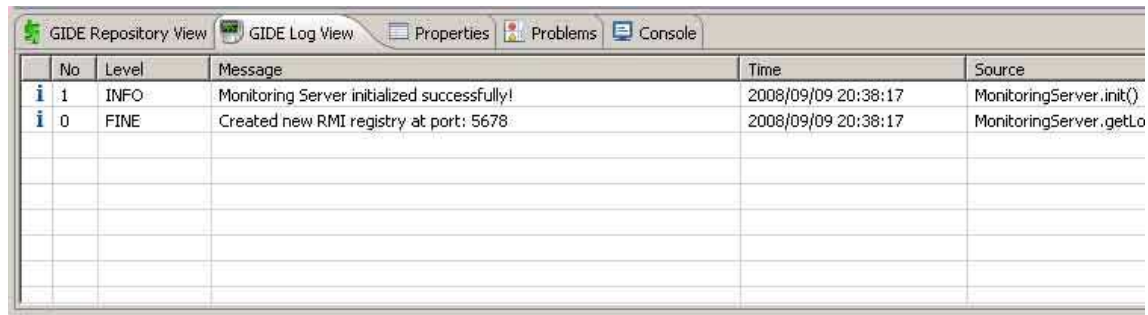
The monitoring perspective also includes a resource monitor view that gives information about the current host.



2.6 Miscellaneous

2.6.1 GIDE Log Viewer

The GIDE Log Viewer provides an independent view dedicated solely to reporting informational, debug, and error messages to the GIDE user.



2.6.1.1 Selection Options

When a log item has been selected the following options are available:

- **Open Item** (equivalent to double clicking): opens up an information window with more details and a potential stack trace for the selected log item.

2.6.1.2 General Options

The log viewer supports the import and export of a previous GIDE log (for example to send to the developers along with bug reports)

- **Clear Log Viewer:** clear the messages in the log viewer. This will not delete the current session log, and the log viewer can be restored through the **Restore GIDE Log Viewer** option (see below).
- **Delete:** deletes the current session log and clears the log viewer. This is an irreversible action (as opposed to the 'Clear Log Viewer' action).
- **Restore Log Viewer:** populates the log viewer with the log items of the current session.
- **Open Log File:** opens the underlying log file in an editor.
- **Configure:** opens a dialog window that enables the user to filter the types of messages displayed in the viewer, and select the number of items appearing in it.
- **Import Log:** replaces the current session log with the one selected from the filesystem.
- **Export Log:** exports the current session log to the selected file in the filesystem.

Bibliography

- [1] M. Aldinucci, S. Campa, P. Dazzi, N. Tonello, G. Zoppi. D.NFCF.04: NFCF prototype and early documentation.
https://bscw.ercim.org/bscw/bscw.cgi/d510923/D.NFCF.04_final.pdf
- [2] F. Baude, D. Caromel, C. Dalmaso, M. Danelutto, V. Getov, L. Henrio, C. Pérez, *GCM: A Grid Extension to Fractal for Autonomous Distributed Components*, Annals of Telecommunications, Springer, 2009, (to appear).
- [3] A. Basso, A. Bolotov, V. Getov, *Behavioural Model of Component-based Grid Environments*, In: From Grids to Service and Pervasive Computing, pp. 19-30, Springer, 2008.
- [4] A. Basukoski, P. Buhler, V. Getov, S. Isaiadis, T. Weigold, *Methodology for Component-based Development of Grid Applications*, Proc. ACM Workshop on Component-based High-Performance Computing, ACM Press, 2008.
- [5] A. Basukoski, V. Getov, J. Thiyagalingam, S. Isaiadis, *Component-based Development Environment for Grid Systems: Design and Implementation*, In: Making Grids Work, pp. 119-128, Springer, 2008.
- [6] D. Caromel, L. Du, Y. Wu, X. Wu, C. Dalmaso, G. Peretti Pezzi. D.CFI.04: Methods and techniques for legacy code wrapping as components.
https://bscw.ercim.org/bscw/bscw.cgi/d510898/D.CFI.04_Final.pdf
- [7] V. Getov, *Integrated Framework for Development and Execution of Component-based Grid Applications*, Proc. IEEE IPDPS, IEEE CS Press, 2008.
- [8] V. Getov, S. Isaiadis, A. Basukoski, J. Thiyagalingam. D.GIDE.03: Grid IDE Prototype and Early Documentation, EU GridCOMP Project, June, 2008.
https://bscw.ercim.org/bscw/bscw.cgi/d510932/D.GIDE.03_Final.pdf
- [9] ProActive 3.90: <http://proactive.inria.fr/>
- [10] T. Weigold, P. Buhler, J. Thiyagalingam, A. Basukoski, V. Getov, *Advanced Grid Programming with Components: A Biometric Identification Case Study*, Proc. IEEE COMPSAC, pp. 401-408, IEEE CS Press, 2008.
- [11] T. Weigold, F. Tumiatti, G Freire. D.UC.04.A Use cases: early documentation.
https://bscw.ercim.org/bscw/bscw.cgi/d510892/D.UC.04.A_Final.pdf
- [12] Eclipse: <http://www.eclipse.org/>
- [13] Grid Component Model: <https://bscw.ercim.org/bscw/bscw.cgi/112268/ GridCompIST-5-034442-Final.CPF>
- [14] IC2D: http://docs.huihoo.com/proactive/3.2.1/IC2D_EclipsePlugin.html