



**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.UC.04.A – Use cases: early documentation

Due date of deliverable: 1 June 2008

Actual submission date: 5 June 2008

**Start date of project**: 1 June 2006 **Duration**: 30 months

Organisation name of lead contractor for this deliverable: GS

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

Keyword List: use case, prototype, component, GCM
Responsible Partner: Gastón Freire, GS

| *MODIFICATION CONTROL* | | | |
|---|---|---|---|
| Version | Date | Status | Modifications made by |
| 1.0 | 05-06-2008 | Draft | Fabio Tumiatti, Thomas Weigold, Gastón Freire |
| 1.1 | 05-06-2008 | Draft | Gastón Freire |
| 1.2 | 17-06-2008 | Draft | Fabio Tumiatti, Thomas Weigold, Gastón Freire |
| 1.3 | 25-06-2008 | Draft | Gastón Freire |
| 1.4 | 02-07-2008 | Final | Fabio Tumiatti |

**Deliverable manager**

- Gastón Freire, GS

**List of Contributors**

- Thomas Weigold, IBM
- Fabio Tumiatti, ATOS
- Gastón Freire, GS

**List of Evaluators**

- Magdalena Escalas, GS
- Marco Danelutto, UNIPI

**Summary**


- This document describes the early prototypes of the use case applications. Their current architectural design is explained in depth, along with the configuration and usage of the demonstrators. A summary of the leveraged features from GridCOMP (and GCM) and the next planned actions for each use case is also offered.

# Table of Content

# 1 Introduction

This document is part of the “D.UC.04 Use cases: early prototypes and early documentation” deliverable, due in M24 of the GridCOMP project.
Also, 4 compressed files (D.UC.04.B*.zip), contain the code (binaries and/or source) corresponding to the four use case early prototypes.

The applications selected for the use cases are the following:

1. Biometric Identification System
2. Computing of DSO Value
3. EDR Processor
4. Wing Design

Each one is covered in a separated section of this document, all of which have a common structure:

1. First, there is an update on the architectural design, highlighting any aspects that have changed or evolved since the primitive version [1]. Also, a description of the infrastructure needed to run the application (data bases, application servers, workflow systems, third-party software components, etc.) is offered.
2. The current (early) prototype is described, explaining its configuration and usage, while providing some examples.
3. A summary of the planned actions for the next period of the project is included.

# 2 Biometric Identification System

## 2.1 Architectural design

### 2.1.1 Architecture of the application

The high-level architectural design of the Biometric Identification System (BIS) as outlined in D.UC.03 (Section 2.2.2) and shown in Figure 1 has been retained for the prototype described in this document. However, under the covers, there have been many changes in the way the system is implemented. The parts of the system that have undergone significant changes are the GCM component architecture and the GCM adapter, the business processes (workflow scripts) interacting with the GCM adapter, and the demo application. The main reason for this is the fact that we have now considered the use of WP3 results, namely, we have implemented and improved the BIS with the autonomic behavioural skeletons. The details of changes in these parts of the system are described in the following subsections.

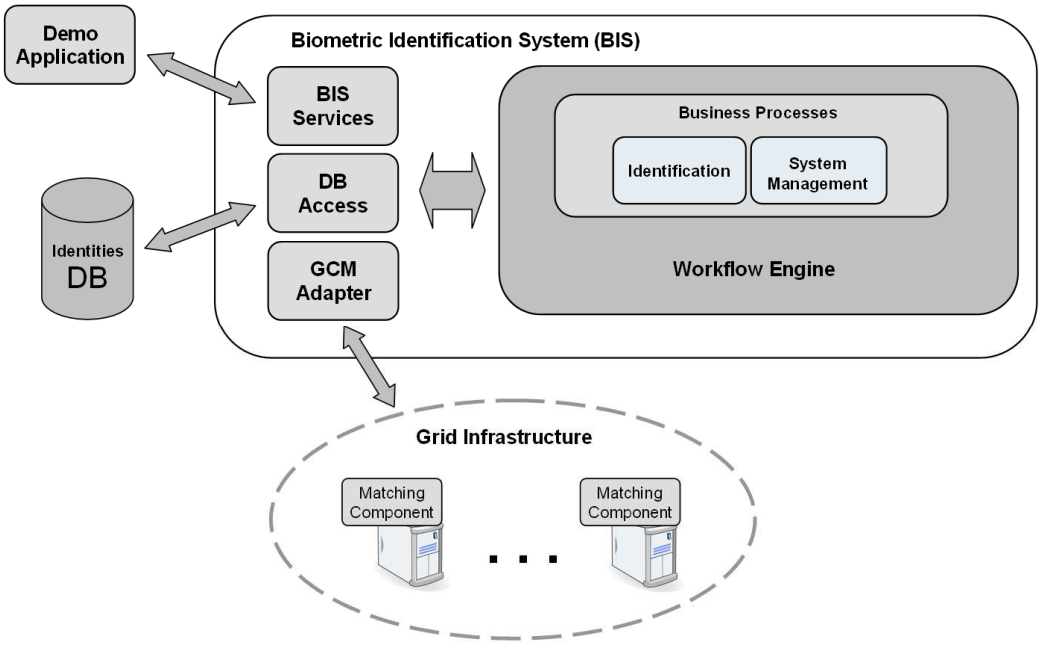



**Figure1:** Biometric identification system high-level overview

#### *2.1.1.1 GCM Component architecture and GCM adapter*

When looking for ways to take advantage of behavioural skeletons to implement the distributed fingerprint matching required for the BIS application, the so-called “task-parallel farm” behavioural skeleton was available (c.f. D.NFCF.01) from WP3. This skeleton assumes that a stream of independent tasks is available and that the tasks can be distributed (e.g. round-robin) to a number of workers. Furthermore, it is assumed that the workers do not maintain any state, which means that new workers can simply be allocated by cloning an existing worker. Obviously, this does not fit well to the distributed identification strategy we had implemented in the first primitive prototype. The approach there was to split the database of known identities into the appropriate pieces, distribute them across the available workers,

and then broadcast the identification requests to all workers. Each worker could then search its part of the database for the given identity. Unfortunately, this strategy did not produce independent tasks and implied that the workers maintained their state, which contradicts with the task-parallel farm requirements.

After providing this feedback to the WP3 partners and discussing the situation, they focused on the implementation of another skeleton, the so-called "data-parallel skeleton", which satisfies the requirements of the BIS and similar data-parallel applications in general. In the meantime, we decided to make use of the task-parallel farm by radically changing our distributed identification strategy as follows:

- We make use of the farm skeleton including an autonomic manager (AM) similar to the example presented in D.NFCF.02, Section 5. The AM measures the farm performance with respect to a given service contract (desired performance in tasks/sec.) and increases or decreases the number of workers if required.
- Instead of distributing parts of the database across nodes, it is assumed that each worker has access to the complete database. More precisely, each worker initially loads the complete database into memory for fast access.
- Instead of broadcasting an identification request to all nodes, the GCM adapter generates a number of tasks which it submits to the farm. Each task includes the biometric information (fingerprints) of the person to be identified and the part of the database (index and length) to be searched in the context of this task.

With this strategy, we have transformed the data-parallel problem into a task-parallel problem which can be solved with the available farm skeleton. This represents the approach used to implement the current prototype described in the following subsections.

#### *2.1.1.2 Business Processes*

The business processes for BIS management and for the actual identification functionality are interacting with the Grid via the GCM adapter. Consequently, the change in the strategy also affects the logic implemented in the corresponding workflow scripts. The "startup" (c.f. D.UC.03) process, as illustrated in Figure 2, now calculates and submits the quality of service (QoS) contract to the AM in activity 3 and then allocates the desired number of initial workers within the farm in activity 4. In the previous design, the node performance was determined and the database was distributed at this point in the workflow.

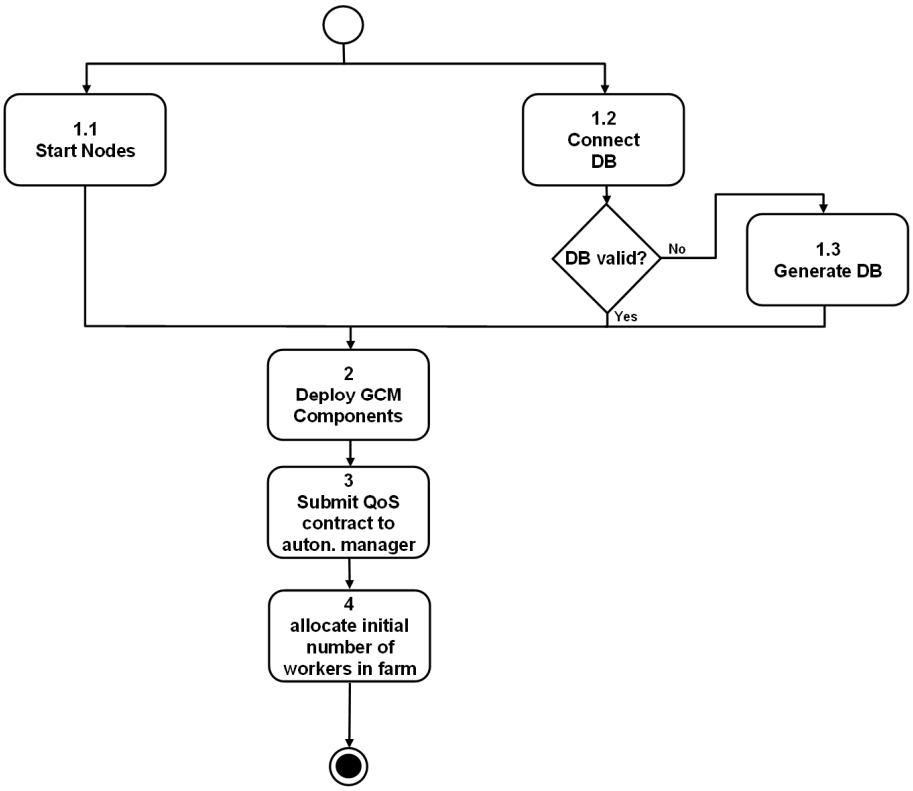



**Figure 2:** Business process "startup" activity-flow diagram

The identification process, named "identify", has changed as well. It now initiates the generation and submission of tasks. The results are collected asynchronously until all tasks have been processed. While doing this, the process sends monitoring events to the workflow monitor attached to the workflow engine. These events report the current state of the identification process (e.g. number of tasks processed etc.), which is then visualized in the command shell described below.

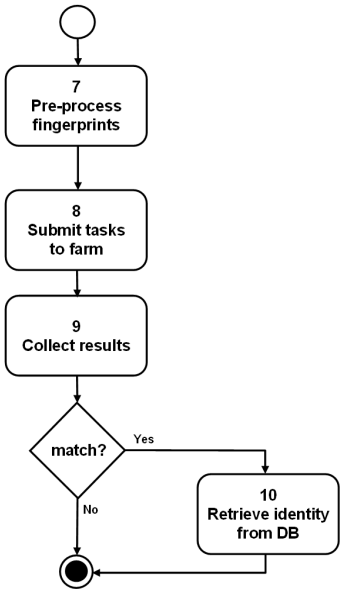



**Figure3:** Business process "identify" activity-flow diagram

#### 2.1.1.3 Demo Application

The first version of the BIS as described in D.UC.03 was static with respect to user interaction. The system was started with fixed parameters, only one person was identified, and the application terminated afterwards. To provide a more flexible and interactive demo, it was planed to implement a simple graphical user interface to vary system parameters and initiate identification requests interactively. However, due to the fact that the code base of the BIS had been changing constantly while moving towards the autonomic version, we decided to postpone the GUI implementation. Instead, we implemented a command shell which offers the same level of interactivity as a GUI, but it is much easier to adapt to the changing base functionality. The GUI will be implemented in the last phase of the project to make the demo visually more attractive.

### 2.1.2 GCM Components

This section describes the GCM component architecture, component description, and interface description of the current BIS prototype.

#### 2.1.2.1 Components diagram

The current BIS prototype, in contrast to the first version as described in D.UC.03, is now based on the autonomic farm skeleton developed within WP3. Figure 4 illustrates the GCM component architecture used in the prototype and indicates how it interacts with the non-componentized part of the application.
At the heart of the component system, there is the farm skeleton which consists of a composite component named *matcherfarm*. It includes a custom controller, the autonomic behaviour controller (*ABC*), which implements the autonomic functionality, for instance, increasing or decreasing the number of worker components within the farm. Furthermore, the farm includes a default implementation of an autonomic manager component, here named *automan*. These three components represent the farm skeleton. To apply the skeleton, we need to parameterise it by adding a worker component, here named *matcher*. This is the component which the ABC clones and adds to the farm as many times as required to increase parallelism. By default, the farm starts with one matcher component. When the number of matchers is increased the ABC automatically replaces the interface I1 of the *matcherfarm* component with a collective (unicast) interface so that multiple matchers can be bound to it.

Besides the farm, the BIS architecture also includes a *collector* component and an *application* component. The *collector* component collects the biometric matching results coming from the farm and forwards them to the workflow engine controlling the BIS. The dashed lines in Figure 4 illustrate the interaction between the non-componentized part of the BIS application, namely the workflow engine, and the GCM component system. Furthermore, Figure 4 indicates that all *matcher* components have access to a shared database storing the identities known by the BIS.
As far as deployment is concerned, the idea is that only the *matcher* components, which represent the workers, are to be distributed.

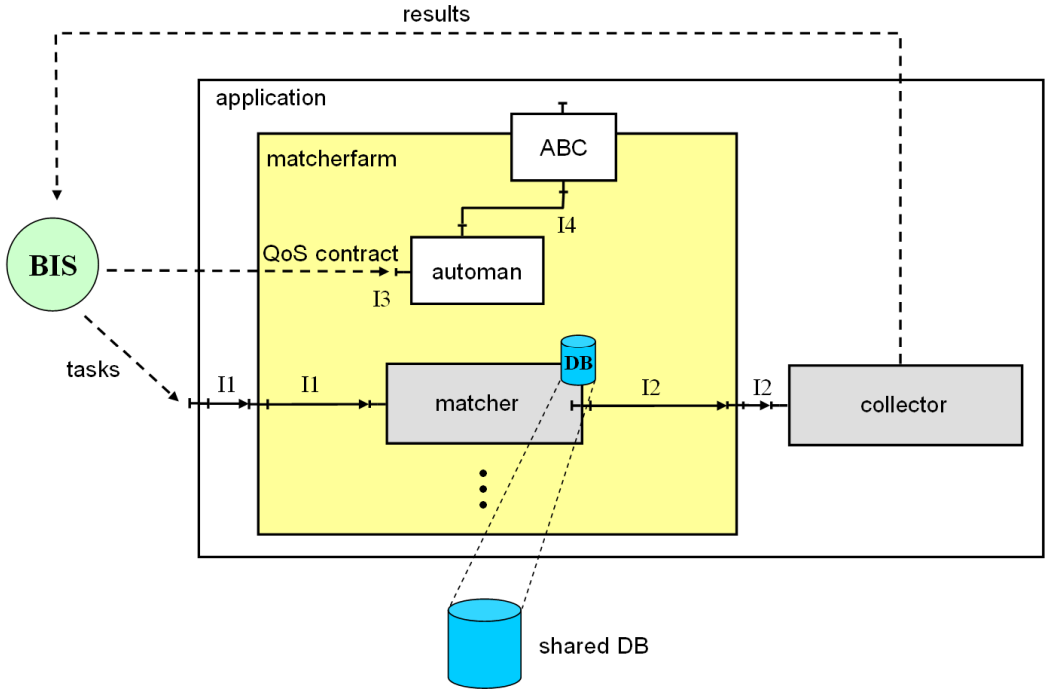



**Figure 4:** GCM component architecture

Figure 5 shows how the component system has been graphically composed within the GIDE to generate the required ADL files.

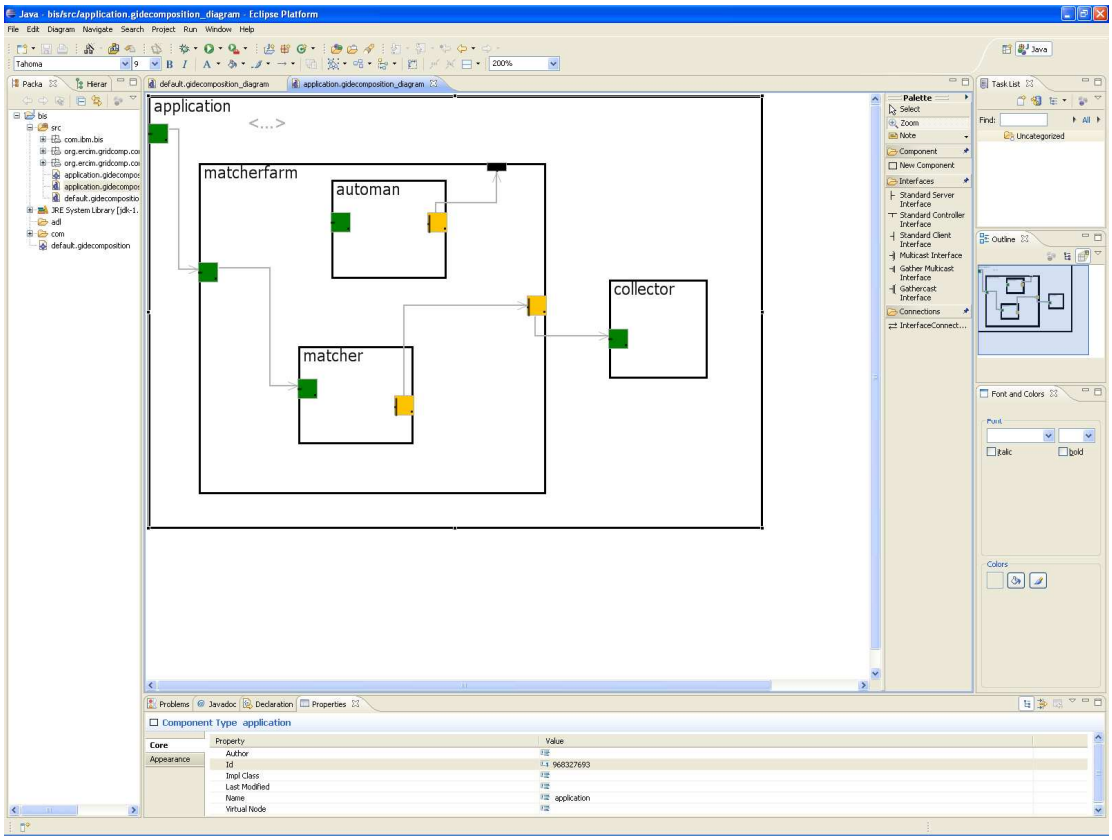



**Figure 5:** BIS component composition in GIDE

#### 2.1.2.2 Components description

##### 2.1.2.2.1 Application

The *application* component is a composite component enclosing the complete component system of the BIS application. As such it hides internal components and only offers one server port to receive identification tasks via interface 1 (I1).

##### 2.1.2.2.2 Matcherfarm

The *matcherfarm* component represents the autonomic farm skeleton as provided by WP3.

##### 2.1.2.2.3 ABC

To implement the behavioural skeleton the default component controller of the *matcherfarm* component has been replaced by an autonomic behaviour controller (*ABC*). The *ABC* is part of the farm skeleton. It offers autonomic operations (as defined in I4) which can be triggered by the autonomic manager (*automan*) or any other external entity.

##### 2.1.2.2.4 Automan

The *automan* component constantly monitors the behaviour of the farm and decides when it is required to trigger an autonomic operation via the *ABC*. The decision is based on the QoS contract which has been previously committed via I3.
The default implementation of the autonomic manager included in the farm skeleton monitors the average service time of the tasks passing through the *matcherfarm* (from I1 to I2). If the average service time reaches the threshold defined by the QoS contract the autonomic manager increases the parallel degree (number of matcher components in the farm) via the *ABC*. Correspondingly, the parallel degree is decreased if the average service time drops significantly below the threshold. The QoS contract defines the desired performance of the farm in tasks/second. We have used the default implementation of the autonomic manager almost unchanged. We just modified some minor details such as the number of tasks over which the average service time is calculated.

##### 2.1.2.2.5 Matcher

The *matcher* component includes the actual fingerprint matching functionality. When a matcher component is created it accesses the shared database and loads it into RAM for faster access. Afterwards, it receives tasks via its interface I1 and returns results via I2. While processing a task, it matches the fingerprints of a given person against a given part of the database.

##### 2.1.2.2.6 Collector

The *collector* component collects the results from the farm and sends them back to the BIS, more precisely, to the identification workflow which further processes them.

#### 2.1.2.3 Interfaces

This section describes the interfaces I1-I4, as denoted in Figure 4 in more detail.

##### 2.1.2.3.1 Interface I1

Interface I1 is used to transfer tasks from the *application* component to the *matcherfarm* and from there to the matcher components. Thus, it is named *IDInput* and includes the *identify()* method as shown below. Apart from the live scan data (fingerprints) of the person to be

identified and the information about the part of the database to be searched, it also includes the handle of the workflow the identification task belongs to. Therefore, it is possible to use the farm for processing multiple identification requests concurrently.

```
public interface IDInput {

/** Match live scan against (part of) the database.
 *
 * @param liveScan Fingerprints of the person to be identified.
 * @param numFingers #fingers which need to match to consider it a match.
 * @param far Desired false accept rate.
 * @param start DB record number (0-n) to start with.
 * @param numRecords Number of records to match.
 * @param procHandle Handle of the identification workflow this
 *                   identification request belongs to.
 * @return RFU.
 */
 public DoubleWrapper identify(LiveScan liveScan, int numFingers,
                               double far, int start, int numRecords,
                               String procHandle);
}
```

#### *2.1.2.3.2 Interface I2*

Interface I2 is used to transfer the result of a biometric matching task from the *matcher* components to the farm and from there to the collector component. The interface, named *IDOutput*, is shown below. As the *identify()* method in the interface I1, the *result()* method in I2 includes the workflow handle. The *collector* component uses this handle to return the result to the corresponding instance of the identification workflow. Furthermore, it includes the name of the node the task was processed on, the number of DB records searched, and the ID of the matching record, if a match was found.

```
public interface IDOutput {

/** Receives result from matcher component.
 *
 * @param nodeName Node name.
 * @param numRecords Number of records matched.
 * @param match RID of matching record, zero if no match, -1 if error.
 * @param procHandle Handle of the identification workflow this
 *                   identification request belongs to.
 */
public void result(String nodeName,  int numRecords, int match,
                   String procHandle);
}
```

#### *2.1.2.3.3 Interface I3*

Interface I3 is used by the BIS to initially define or update the QoS contract with the autonomic manager.

```
public interface AutonomicServerManager {

 /** Commit Qos contract to the autonomic manager.
  *
  * @param qosContract Contract string in the format “TaskPerSecond := N”
  * /
 GenericTypeWrapper commitContract(String qosContract);
}
```

#### *2.1.2.3.4 Interface I4*

Interface I4 is used by the autonomic manager to trigger autonomic operations offered by the *ABC*. The available operations are: *"Farm::ServiceTime"*, *"Farm::IncreaseParallelDegree"*, and *"Farm::DecreaseParallelDegree"*. This allows the manager to retrieve the average service time and to take appropriate action if required.

```
public interface AutonomicController {

/** Lists available autonomic operations.
 * @return Array of strings defining the available operations.
 */
public String[] listAutonomicOperations();

/** Execute the desired autonomic operation.
 *
 * @param op String defining the operation to execute
 * @param params varialble argument list
 * @return Result of the operation.
 */
public GenericTypeWrapper execOperation(String op, Object... params);
}
```

### *2.1.2.4 Summary of the GCM features used*

During the second year of the project, the focus was on exploring the WP3 results and integrating the autonomic functionality into the BIS prototype. This implies the use of almost all of the other features of the CFI such as composite components, collective interfaces, deployment descriptors, etc. Furthermore, we have use the GIDE for graphical component architecture design, composition, ADL file generation and monitoring. Also, we have published our experiences with the CFI and the GIDE in [2].

## 2.2 Early prototype

### 2.2.1 Description

The prototype (V2) as described in the previous sections brings a number of improvements over the first version (V1) delivered in D.UC.03, mostly because it uses the autonomic farm developed in WP3. However, due to the fact that the task-parallel farm does not satisfy all needs of the BIS, a few trade-offs had to be made. The pros and cons of the current prototype with respect to the first version and its overall functionality can be summarized as follows.

- Thanks to the autonomic farm skeleton, the V2 prototype dynamically scales depending on system parameters to maintain the desired performance. In V1 the performance estimations were only made during system startup meaning that it was completely static.
- In addition to performance, the V2 prototype also scales with respect to concurrent identifications. In V1, all nodes where working on one identification request at any point in time. Consequently, people could only be identified sequentially. V2, in contrast, can be used to work on an arbitrary number of identifications concurrently.
- The use of the farm skeleton significantly reduced the development time, which becomes clearly visible when comparing the code size of V1 and V2. Both prototypes required about the same amount of code to be written whereas V2 provides much more functionality. Adding all this functionality to V1 manually, without the use of the skeleton, would have required significantly more effort.

- The trade-off made is the fact that the workers have to load the complete database into RAM. This limits the scalability of the solution to the amount of RAM available. Also, loading the complete database requires quite some time such that the farm grows relatively slow.

Although the current prototype does not represent the optimal solution to the problem, it works well and is a good demonstrator for both, the autonomic farm skeleton and the CFI as a whole. Furthermore, its development has generated very important feedback for the WP3 partners and significantly influenced the development of the data-parallel farm skeleton.

### 2.2.2 Configuration and usage

The current prototype is available in the file D.UC.04-IBM.zip (available through BSCW). For running the prototype the file *js.jar* including Rhino 1.6R7 must be downloaded from http://www.mozilla.org/rhino/ and stored into the subdirectory *lib/ePVM/*. This is required by the workflow engine included in the BIS.

The prototype is configured to run on Grid5000. It makes use of two deployment descriptors. Firstly, the file descriptor/BIS-Grid.xml defines the node on which the initial worker of the farm is running on. The farm itself is running on the default node (the local JVM of the application). Secondly, the file deployment/deployment-descriptor.xml defines all nodes available to the farm for allocating additional workers. It assumes the file nodesBIS.properties to be present including a list of machine names reserved in Grid5000 (an example file is included).

The application can be started via the included *run* script. The application takes command line arguments with the following syntax: *<max-time> <db-size> <task-size> <additional-workers>*. *Max-time* denotes the desired maximum identification time (time to search the complete DB for a matching identity) in seconds, *db-size* defines the desired database size, *task-size* indicates the number of identities matched per task, and *additional-workers* defines the number of workers in addition to the initial worker the farm should start with.

When the application is started, it displays the initial parameters defined at the command line. Then, the nodes are started by activating the deployment descriptors, the database is accessed (and generated if required), and the GCM components are deployed. Finally, the QoS contract is calculated based on the current parameters and submitted to the autonomic manager. Afterwards, the BIS application is ready and enters its command shell mode displaying the prompt *GridCOMP BIS>* as shown below.

```
Starting BIS:
      Max. identification time: 10 sec.
      DB size                : 50000
      Task size              : 50
      Additional workers     : 100

Starting nodes
Connecting to database
Deploying grid components
Submit QoS contract (100 tasks/sec.) and allocated additional workers
BIS startup successful (identities: 50000, QoS contract: 100 tasks/sec.,
additional workers: 100)

GridCOMP BIS>
```

Once started, the BIS application can be used interactively via the command shell. Typing the command "*?*" lists the available commands offered by the shell as shown below.

```
GridCOMP BIS> ?
 identify [<id>]
      id -> id (1-n) of the person to identify (0 means unknown person, no
      id means randomly choosen)
 time <max-time>
      max-time -> desired max. identification time in seconds
 task <task-size>
      task-size -> number of matches per task sent to the farm
 ls
      list the current application state (e.g. number nodes in the farm)
 quit
      quit BIS application.
```

The shell commands allow modifying application parameters, retrieving the application state, as well as triggering identification requests. This way, one can see how the farm automatically increases/decreases the parallel degree while processing identification tasks to reach the given performance goal.

### 2.2.3 Examples

The trace below shows how an identification request is processed by the BIS prototype. Firstly, the command *identify* is used to trigger the identification of a randomly chosen known person. Here, the person with the ID 42147 is retrieved from the database, and its fingerprints are used for identification. Secondly, 1000 identification tasks are generated and submitted to the farm. Thirdly, the BIS prints the progress of the identification periodically. Finally, after one of the nodes reported the matching ID to be 42147, the person is retrieved from the database, and it can be seen that the identification was successful.

```
GridCOMP BIS> identify
Identify known person:
      RID         : 42174
      First Name  : John
      Last Name   : Doe 42174
      Adress      : 1st Avenue, New York City, USA
1000 identification tasks submitted to farm
----------------
Outstanding tasks : 999
Number of nodes   : 1
Identities matched: 50
----------------
Outstanding tasks : 998
Number of nodes   : 2
Identities matched: 100
----------------
Outstanding tasks : 989
Number of nodes   : 11
Identities matched: 550
----------------
Outstanding tasks : 976
Number of nodes   : 24
Identities matched: 1200
----------------
Outstanding tasks : 712
```

```
Number of nodes   : 96
Identities matched: 14400
---------------
Outstanding tasks : 621
Number of nodes   : 96
Identities matched: 18950
---------------
Outstanding tasks : 529
Number of nodes   : 96
Identities matched: 23550
---------------
Outstanding tasks : 378
Number of nodes   : 96
Identities matched: 31100
---------------
Outstanding tasks : 355
Number of nodes   : 96
Identities matched: 32250
---------------
Outstanding tasks : 231
Number of nodes   : 96
Identities matched: 38450
---------------
Outstanding tasks : 32
Number of nodes   : 101
Identities matched: 48400
---------------
Outstanding tasks : 0
Number of nodes   : 101
Identities matched: 49974

Person successfully identified, rid: 42174, retrieving identity from DB…
Identification successful:
      First Name  : John
      Last Name   : Doe 42174
      Adress      : 1st Avenue, New York City, USA
GridCOMP BIS>
```

## 2.3 Next actions

Together with this deliverable, a new version of the farm skeleton will become available. Also, a first version of the new data-parallel farm will be finished. Depending on the time constraints, we envision to either migrate to the new version of the task-parallel farm or even switch to the data-parallel skeleton. The latter would more significantly improve the prototype but requires substantial change to the workflow logic (back to the concept of distributing parts of the database across workers).
Furthermore, we will replace the command shell with a Java GUI to make the user interaction with the prototype more appealing.

Finally, we will continue to make use of the GIDE and provide feedback on its functionality in technical meetings as carried out during the second year.

# 3 Computing of DSO Value

## 3.1 Architectural design

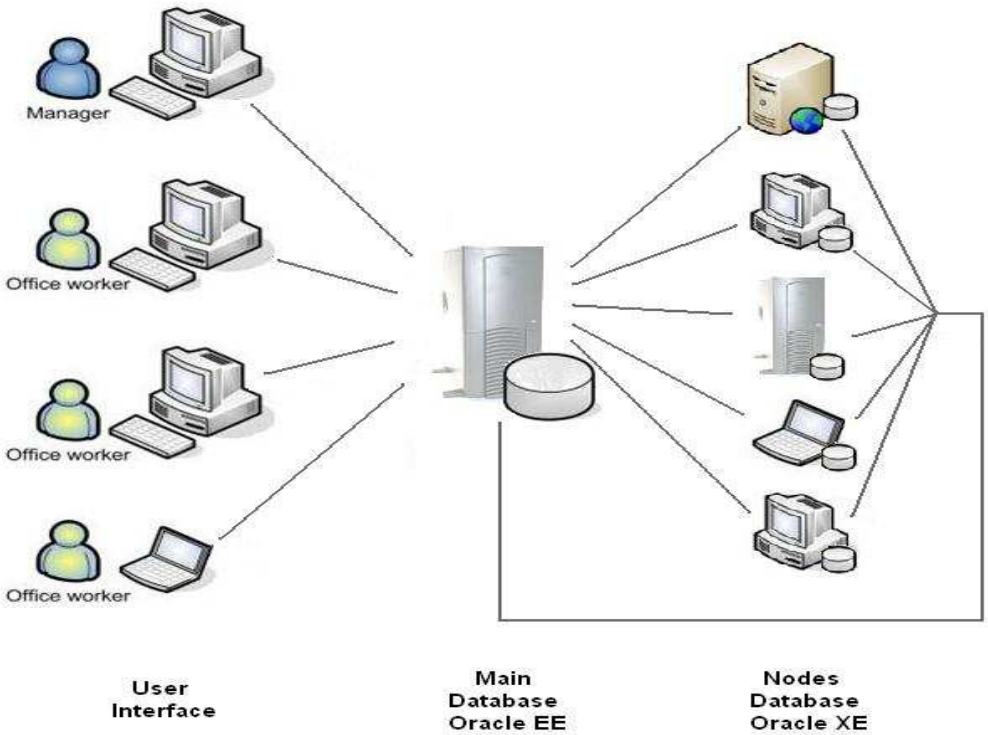
You can find the complete description of this use case and its background in the D.UC.03 deliverable. This section will only include a summary of the architectural design and an update from the previous deliverable.

### 3.1.1 Architecture of the application

The application selected by Atos to be used by this use case was the "Computing of DSO value". The DSO (Days Sales Outstanding) is the mean time needed for an invoice to be paid. The application is based on PL/SQL code and needs to be run the following infrastructure and programs:

- One main server where the master database will be installed
- Install the database on the main server: Oracle Enterprise Edition or Oracle Standard Edition *
- Several nodes computers (server, desktops, laptops) to be used as workers
- Install the database in each node: Oracle Express Edition (free of charge) *
- Install the Java runtime environment 1.6 in all computers/servers
- If you are using Windows operating system, you need to install the following applications to enable ssh connections:
    - Cygwin
    - SSH server for Cygwin

The following picture illustrates the infrastructure needed to run the Computing of DSO Value application:

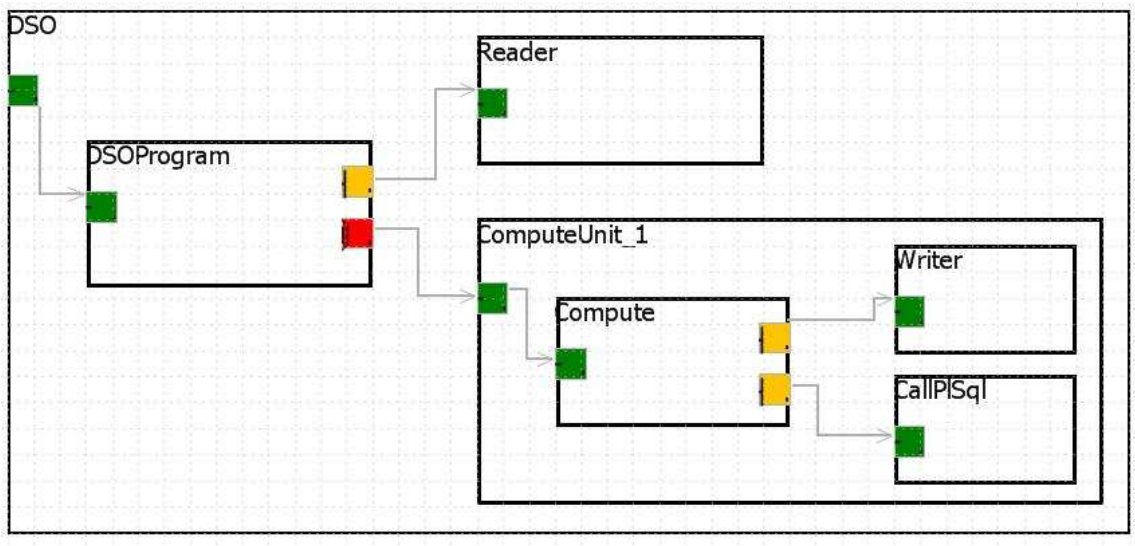


* The database information (PL/SQL code, tables, etc.) will not be described and distributed with the documentation because the DB code is confidential.

## 3.1.2 GCM Components

### *3.1.2.1 Components diagram*

The following picture illustrates the components diagram proposed to use with the Computing of DSO Values use case:

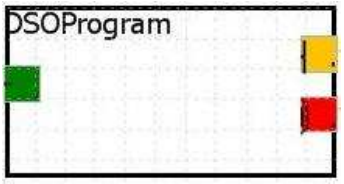



The application workflow used with this diagram is:

1. The client user interface makes a request to the DSOProgram component to start the application
2. The DSOProgram component obtains the list of clients' IDs to be processed from the Reader component
3. The DSOProgram component breaks the list of clients' IDs into chunks
4. These chunks are sent to the ComputeUnits component on the remote nodes to be processed
5. The Compute component receives the chunks from the ComputeUnit and inserts it on the slave database using the Writer component
6. After that, the Compute component calls the stored procedure to execute the PL/SQL code using the CallPlSql component

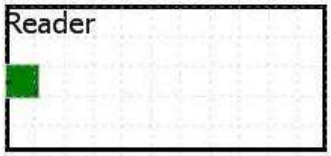
### *3.1.2.2 Components description*

#### *3.1.2.2.1 DSOProgram component*



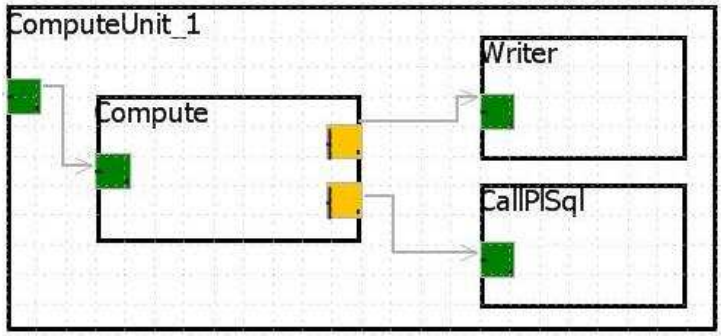

The DSOProgram is the master component of the application, and it is responsible of the program workflow. It offers a *runnable* server interface and 2 client interfaces, called *read* and *ourTaskMulticast.*

#### *3.1.2.2.2 Reader component*



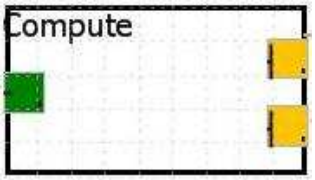

The Reader component offers the functionality to connect to the master database and gets the list of clients' IDs that will be processed by the application.

#### *3.1.2.2.3 ComputeUnit component*



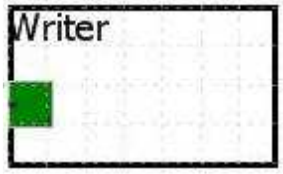

The ComputeUnit component is a composite component with 3 sub-components. It is responsible of the execution of tasks on the remote nodes and offers an *ourTask* server interface.

#### *3.1.2.2.4 Compute component*



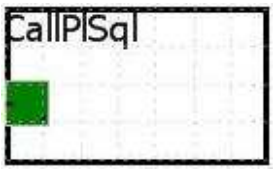

The Compute component offers the functionality to receive the tasks from the ComputeUnit and execute them. The component receives the list of clients' IDs and sends it to the Writer component that will insert it in the node database. After that, the component starts the CallPlSql component to execute the PL/SQL code.

#### *3.1.2.2.5 Writer component*




The Writer component offers the functionality to write on the node database the list of clients' IDs to be processed by the PL/SQL code.

#### *3.1.2.2.6 CallPlSql component*




The CallPlSql component offers the functionality of wrapping PL/SQL code. This component calls an Oracle stored procedure stored in the node database that will execute the PL/SQL code.

### *3.1.2.3 Interfaces*

The following codes illustrate the interfaces used to build the components listed above.

The first interface called is the Reader interface. It's responsible for connecting to the master database and getting the list of clients' IDs from the clients table.

```
public interface Reader {

        /**
         * Gets the list of client's Ids from the database.
         *
         * @param clientId      Id from the specific client to be processed
         * @param groupId       Id from the group of clients to be processed
         * @return              list of clients' IDs
         */
        String[] getClients(String clientId, String groupId);
}
```

The DSOProgram component implements a multicast client interface, OurTaskMulticast, which sends the task to several ComputeUnit components. This interface implements a method called *compute* with two parameters, one using the parameter dispatch mode to be ROUND_ROBIN and the other using the parameter dispatch mode to be BROADCAST.

```
public interface OurTaskMulticast extends Serializable {

    /**
     * A multicast client interface to send the tasks to the nodes
     *
     * @param clients   the list of the list of clients' IDs
     * @param dates             the period to be processed
     * @return          success or failure
     */
    public List<BooleanWrapper> compute(
    @ParamDispatchMetadata(mode=ParamDispatchMode.ROUND_ROBIN) List<List<String>>
```

```
    clients,
    @ParamDispatchMetadata(mode=ParamDispatchMode.BROADCAST) List<String> dates);
}
```

The OurTask interface is responsible for starting the process in the remote nodes.

```
public interface OurTask extends Serializable {

    /**
     * A server interface to receive the tasks on the nodes
     *
     * @param clients   the list of clients' IDs
     * @param dates             the period to be processed
     * @return          success or failure
     */
    public BooleanWrapper compute(List<String> clients, List<String> dates);
}
```

The Writer interface is responsible for connecting to the node database and inserts the list of clients´ IDs on the temporary table to be executed by the PL/SQL code.

```
public interface Writer {

        /**
         * Insert the list of client's ids in the database.
         *
         * @param clients               the list of clients' IDs
         * @param start_date    the initial date of the process
         * @param end_date              the final date of the process
         * @return                      success or failure
         */
        public boolean insertClients(String[] clients , String start_date, String end_date);
}
```

The CallPlSql interface if responsible for executing the PL/SQL code stored inside the Oracle Stored Procedure on the node database.

```
public interface CallPlSql {

        /**
         * Call the oracle stored procedure to be executed.
         *
         * @return success or failure
         */
        public boolean executePlSql();
}
```

#### *3.1.2.4 Summary of the GridCOMP features used*

The following GridCOMP features are used in the Compute of DSO Value implementation:

- Primitive components defined via ADL files (XML fractal files).
- Composite components including the bindings of the sub-components are defined via the ADL.
- Use of deployment descriptors and Virtual Nodes to define the Grid infrastructure and component deployment. The support of network protocols standards such as SSH is used in the deployment descriptor.
- Server and client interfaces including one multicast server interface are defined.
- Use of different parameters of dispatch mode: BROADCAST and ROUND_ROBIN.

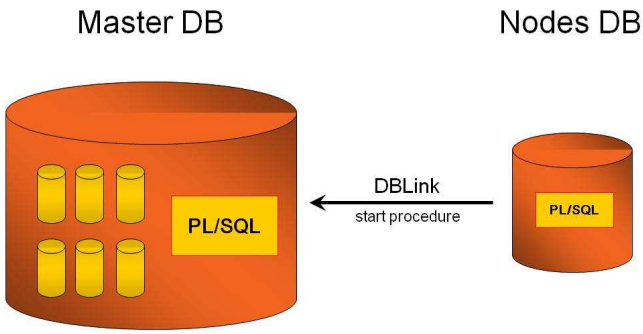
## 3.2 Early prototype

### 3.2.1 Description

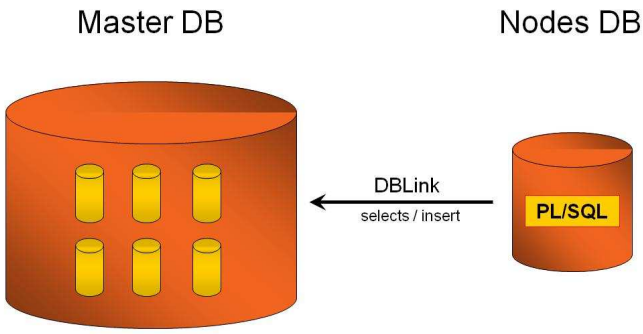
The main differences between this prototype implementation and the primitive one are:

1. Current implementation is based on components, since the primitive one was based on Active Objects.
2. Scheduler implemented inside the DSOProgram component. The temporary Master/Slave API is not used any more.
3. Implementation of a graphical user interface to input parameters and view the execution logs.
4. Implementation of the original DSO PL/SQL code: creation of the original tables, packages and functions.

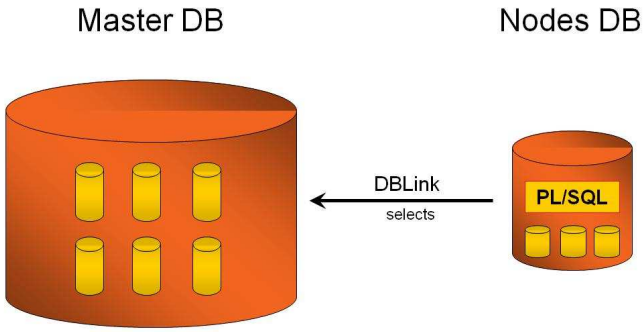
Some effort was spent in analyzing the different ways to distribute the PL/SQL code with a grid solution. After some research and analysis, we identified 4 different ways to distribute a PL/SQL code. The following images show the database structure and our analysis/comments about each possibility.




The first analysis was to have the following database structure: all tables and PL/SQL code are inside the master database and a part of the code is inside the node database. The code put inside the node DB is only to start the PL/SQL process and to do the first calculation without data access. This code will start the original PL/SQL code stored inside the master DB through DBLink (direct connection between the databases). This option can be used when the PL/SQL code does a lot of calculations without specific data access.

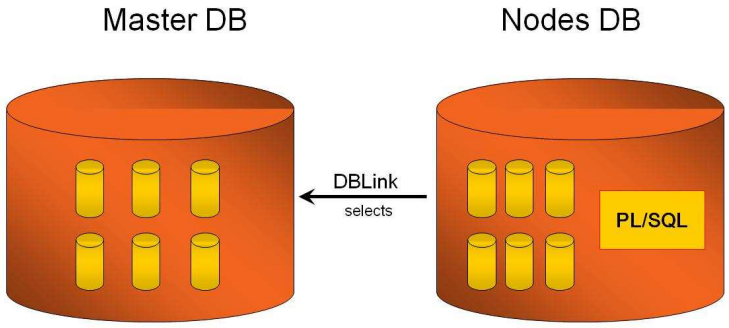


The second database structure is to have all tables inside the master database and a full copy of the PL/SQL code inside the node database. The execution of the PL/SQL code will be inside the node DB and all data access will be done through DBLink to the master DB. This option was discarded because require too much network throughput, making the execution slow.




The third database structure is to have all PL/SQL code and main tables inside the node database. At the beginning of the process, the node DB will select from the master DB the main data needed to execute the PL/SQL code and store it in the node tables. If the process needs more data, it will take it from the master DB through DBLink. This option can be used when the PL/SQL code does a lot of calculations with specific data access.

This database structure was selected to be used with the Computing of DSO Values use case because the application PL/SQL code use specific information stored in specific tables to do the calculations. The PL/SQL code will use the information stored inside the node tables to do the calculation and if needs more date, it will take it from the master database through DBlink.




The fourth database structure is to have all PL/SQL code and all tables inside the node database. All processes will be executed inside the node DB without access to the master DB and requiring database replication. This method was discarded because Oracle Express Edition, which will be installed in the nodes, has limitation on disk space.

### 3.2.2 Configuration and usage

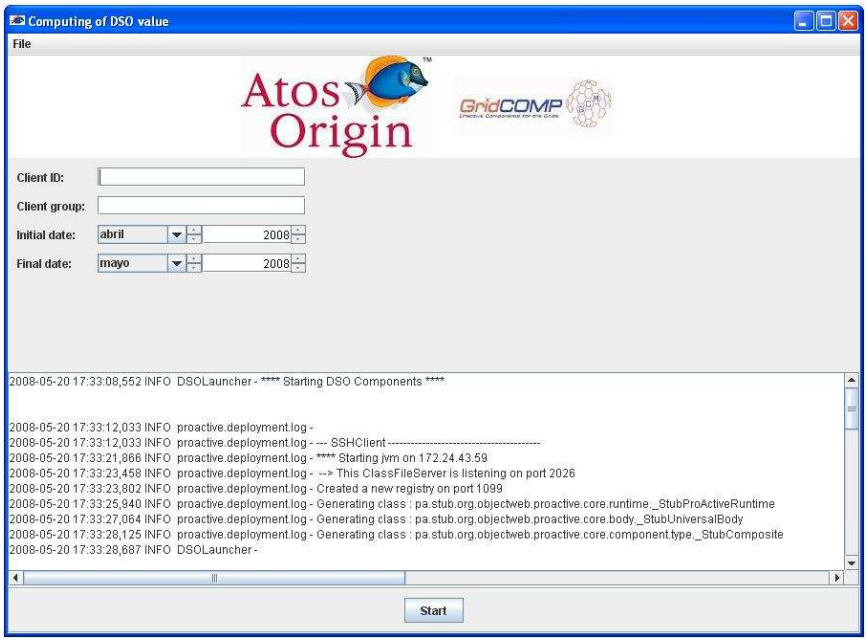
The first thing to do before executing this prototype is to install the required software listed in section 3.1.1. After installing, testing and running all required software you can start to configure the prototype.
You can find the binary code inside the file “D.UC.04 – DSO early prototype.zip”. Uncompress the zip file and add the following libraries to the lib directory:

- ProActive 3.9 (ProActive binaries and related libraries)
- classes12.jar (JDBC library)

The following files need to be changed to configure and run the prototype on your environment:

- *\classes\com\atosorigin\usercase\dso\deployment.xml* - open the deployment file and rewrite it with the nodes information
- *\classes\com\atosorigin\usercase\dso\comp\CallPlSqlImp.fractal* - open the fractal file and change the attributes *url*, *user* and *pwd* with the node database information. It is necessary to have one fractal file for each node
- *\classes\com\atosorigin\usercase\dso\comp\DSO*.fractal – open the fractal file and rewrite it with the virtual nodes information and the binding connections
- *\classes\com\atosorigin\usercase\dso\comp\DSOProgram*.fractal – open the fractal file and change the attribute numTasks value with the number of nodes used
- *\classes\com\atosorigin\usercase\dso\comp\ReaderImp.fractal* - open the fractal file and change the attributes *url*, *user* and *pwd* with the master database information
- *\classes\com\atosorigin\usercase\dso\comp\WriterImp.fractal* - open the fractal file and change the attributes *url*, *user* and *pwd* with the node database information. It is necessary to have one fractal file for each node

Start the main application "DSOProgram," and the graphical user interface will start.




The first thing that the application will do is to create the remote nodes. After that, the *Start* button will be enabled. Set the parameters and push the button.

### 3.2.3 Examples

To test the application you can use the following parameters:

- Client ID: <leave empty>

- Client group: <leave empty>
- Initial date: enero 2007
- Final date: febrero 2007

When the executions finish, you can check the result in the result table.

## 3.3 Next actions

The early prototype is a sample of the “tuned” final prototype that will be presented at the end of the project. Some actions need to be done to refine this prototype and make it more powerful and useful, turning it an example to be used in real industrial world application.

To make this happen, some enhancements need to be made on the next period:

- Refine the user interface, showing the results in the GUI (data from the result table)
- Integrate the autonomic controller (FARM code) with the DSO code, providing a way to add or remove a specific worker at execution time.
- Test the application with the real database, same amount of data, to check the performance against the original application (without GRID)
- Refine the code documentation

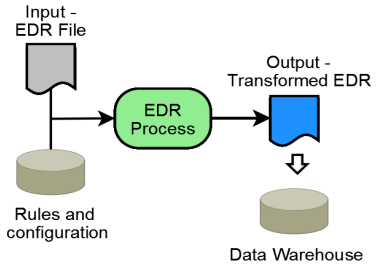
# 4 EDR Processor

## 4.1 Architectural design

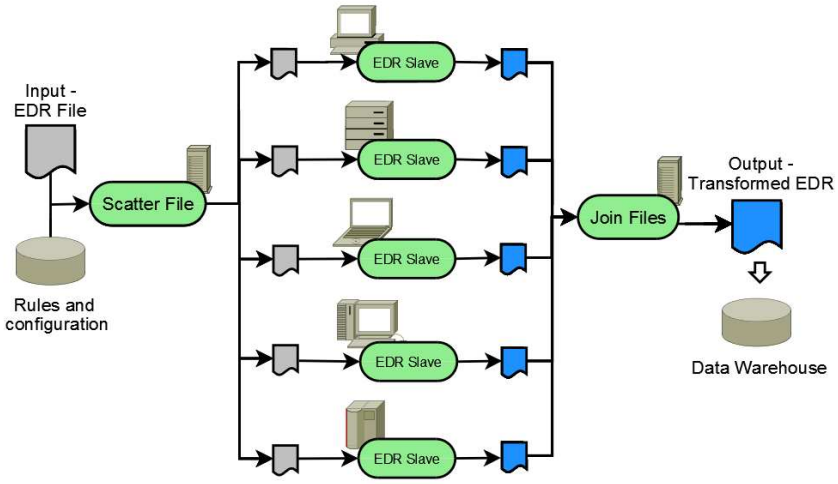
A complete description of this use case and its background can be found in D.UC.03 deliverable [1]. This section will only include a summary of the architectural design and an update from the previous deliverable.

### 4.1.1 Architecture of the application

An actual EDR Processor application will work unattended, inside a nightly batch process, taking information from a sequential file (previously generated from some source database) and storing the results into another sequential file (eventually imported to a target database). For the purposes of this project, the source and target databases will be ignored.



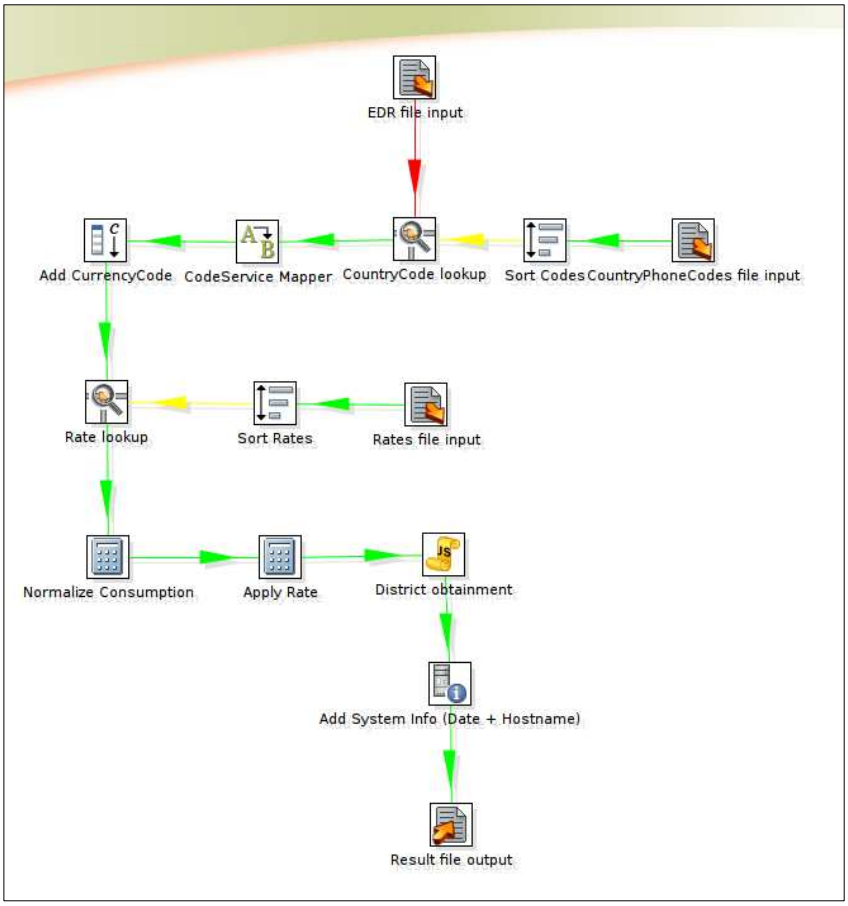

Being an embarrassingly parallel process, the EDR processing can be easily distributed among a set of (likely heterogeneous) computing resources. In order to do that, the input EDR file must be split into fragments. Each fragment will be processed by a grid resource, and the results will later be joined.
The following picture shows the conceptual behaviour of the application:



The scattering and joining of the files is performed by a “master” resource (the one running the application). The processing of the fragments is done by the “EDR slaves”, which transfer the result files back to the “master”.
As explained in previous deliverable documents and presentations, the processing of an EDR is a rather simple case of an Extract, Transform and Load (ETL) process.

#### 4.1.1.1 Extract, Transform and Load

The Extract Transform and Load processing is done using Pentaho Data Integration, also known as Kettle Project [3]. Kettle is an open source ETL library that includes a very user-friendly integrated development environment. Using that IDE the user can easily design the ETL process and save it to a metafile. That metafile can later be used to execute the ETL process through the Java API of the Kettle libraries.
The following picture shows the design of the ETL corresponding to the processing of an EDR file:




Each one of the steps of the transformation is described in the following sections.

##### 4.1.1.1.1 EDR File Input

In this step, the EDR input file is read, parsing the different fixed length fields.
The following table shows the format of the EDR input file:

| # | Name | Type | Format | Position | Length |
|---|---|---|---|---|---|
| 1 | ID | BigNumber | | 0 | 13 |
| 2 | State | String | | 16 | 1 |

| | | | | | |
|---|---|---|---|---|---|
| 3 | Distributor | String | | 17 | 4 |
| 4 | CountryDialingCode | Integer | ##0 | 21 | 3 |
| 5 | Phone | String | | 24 | 12 |
| 6 | SStart | Date | yyyyMMddHHmmssSSS | 36 | 17 |
| 7 | Duration | String | | 53 | 4 |
| 8 | ProgConv | String | | 57 | 4 |
| 9 | ProgElem | String | | 61 | 2 |
| 10 | CodeSelection | String | | 63 | 2 |
| 11 | IndexKey | String | | 65 | 5 |
| 12 | AreaCode | String | | 70 | 5 |
| 13 | TypeCall | Integer | #0 | 75 | 2 |
| 14 | TypeISDNFlux | String | | 77 | 3 |
| 15 | TypeCodeFlux | String | | 80 | 3 |
| 16 | DataCodePage | String | | 83 | 19 |
| 17 | TotalConsumption | Number | | 102 | 6 |
| 18 | TotalTax | Integer | | 108 | 6 |
| 19 | TotalNetUse | Integer | | 114 | 6 |
| 20 | SingleConsumption | Number | | 120 | 6 |
| 21 | SingleTax | Integer | | 126 | 6 |
| 22 | SingleNetUse | Integer | | 132 | 6 |
| 23 | StdMtpn | String | | 138 | 6 |
| 24 | UseRecalc | String | | 144 | 6 |
| 25 | UM | String | | 150 | 2 |
| 26 | CauseTaxation | String | | 152 | 4 |
| 27 | CodeAnomaly | String | | 156 | 3 |
| 28 | DataCEM | Date | yyyyMMdd | 159 | 8 |

#### *4.1.1.1.2 CountryPhoneCodes file input*

This step reads the file containing the mapping between phone prefix and ISO country codes. This is a CSV (Character Separated Values) file, with the following format:

| *#* | *Name* | *Type* | *Format* | *Length* |
|---|---|---|---|---|
| 1 | PhoneCode | Integer | | 3 |
| 2 | CountryCode | String | | 2 |

#### *4.1.1.1.3 Sort codes*

This step sorts the contents of the file read at the previous step, by `PhoneCode`, and prepares them to perform a lookup.

#### 4.1.1.1.4 CountryCode lookup

This step adds a new field (`CountryCode`) to the output of the processing. This field is populated with the ISO country code corresponding to the phone prefix matching the one contained in the EDR.

#### 4.1.1.1.5 CodeService Mapper

It maps the `TypeCall` field from the EDR to a new target field, `CodeService`, using a pre-defined mapping table:

| *Source value* | *Target value* |
|---|---|
| 0 | Voice |
| 1 | SMS |
| 2 | MMS |
| 3 | GPRS |
| 4 | WAP |
| 5 | 3G |
| 6 | ISDN |
| 7 | ADSL |

#### 4.1.1.1.6 Add currency code

This step adds a new `CurrencyCode` field, containing "EUR" (all prices are in EURO).

#### 4.1.1.1.7 Rates file input

It reads the file containing the rates to apply to an EDR to compute its price. This file contains the following fields (in CSV format):

| *#* | *Name* | *Type* | *Format* | *Length* |
|---|---|---|---|---|
| 1 | CountryCode | String | | 2 |
| 2 | ServiceCode | String | | 5 |
| 3 | Rate | Number | #,##0.00 | 6 |

#### 4.1.1.1.8 Sort rates

It sorts the file from the previous step by `CountryCode` and `ServiceCode` fields, preparing its contents to be looked up.

#### 4.1.1.1.9 Rate lookup

For each EDR, and using the `CountryCode` and `ServiceCode` fields, it obtains the corresponding `Rate` from the Rates file. It adds this as a new field to the result.

#### 4.1.1.1.10 Normalize consumption

It normalizes de contents of the field `TotalConsumption`, expressed in milliseconds, to seconds (rates are per second). A new field, `TotalConsumptionNorm` is added to the results.

#### 4.1.1.1.11 Apply rate

It adds a new field, `Total`, as the result of the multiplication of `TotalConsumptionNorm` and `Rate`, obtained from previous steps.

#### 4.1.1.1.12 District obtainment

It adds a new field, `District`, containing the first 3 digits of the `Phone` number.

#### 4.1.1.1.13 Add system info

It adds two new fields, `RecordDate` and `Hostname`, when and where each record has been processed, respectively, for debugging purposes.

#### 4.1.1.1.14 Result file output

It writes the results of the process to a file, using CSV format. Each line contains the fields from the original EDR plus the new fields added from previous steps.

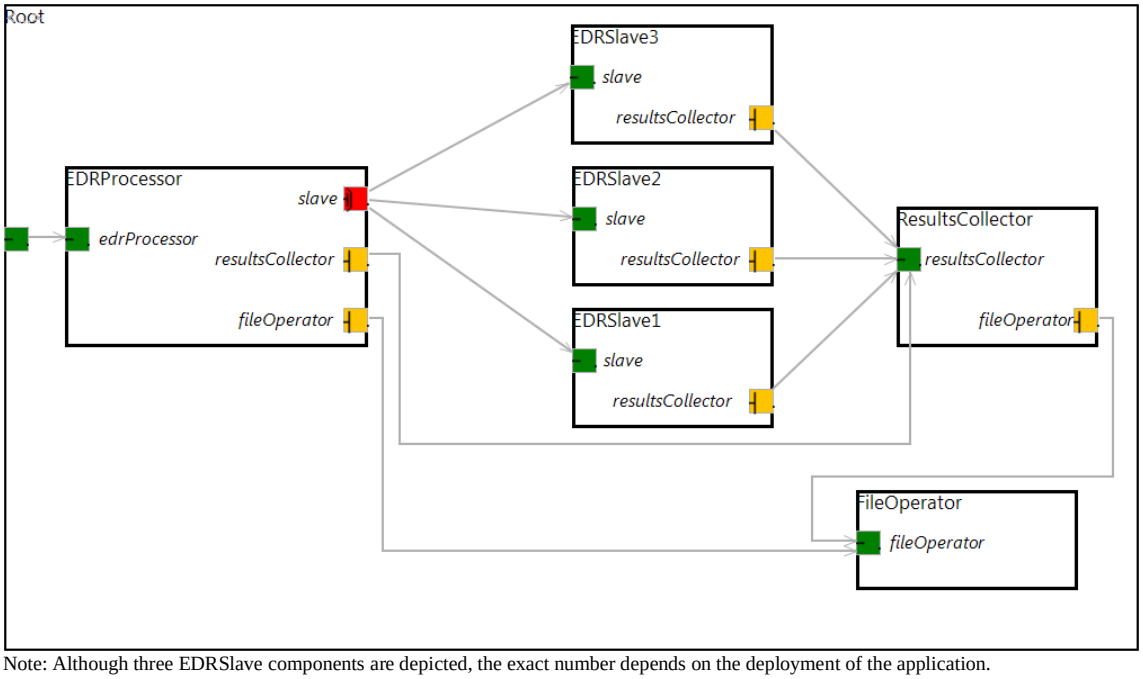
### 4.1.2 GCM Components

The architectural design of the prototype has suffered some changes from its primitive version. The FileSupplier component has been removed, and two new components have been introduced, one of them only for the autonomic version of the architecture.
All the logic related to the generation of random content for the EDR files has been moved to an auxiliary tool and will not be commented any further in this document.

#### 4.1.2.1 Components diagram

##### 4.1.2.1.1 Non-autonomic

The non-autonomic components diagram (made using the Grid IDE) corresponding to the early prototype is the following:




Note: Although three EDRSlave components are depicted, the exact number depends on the deployment of the application.

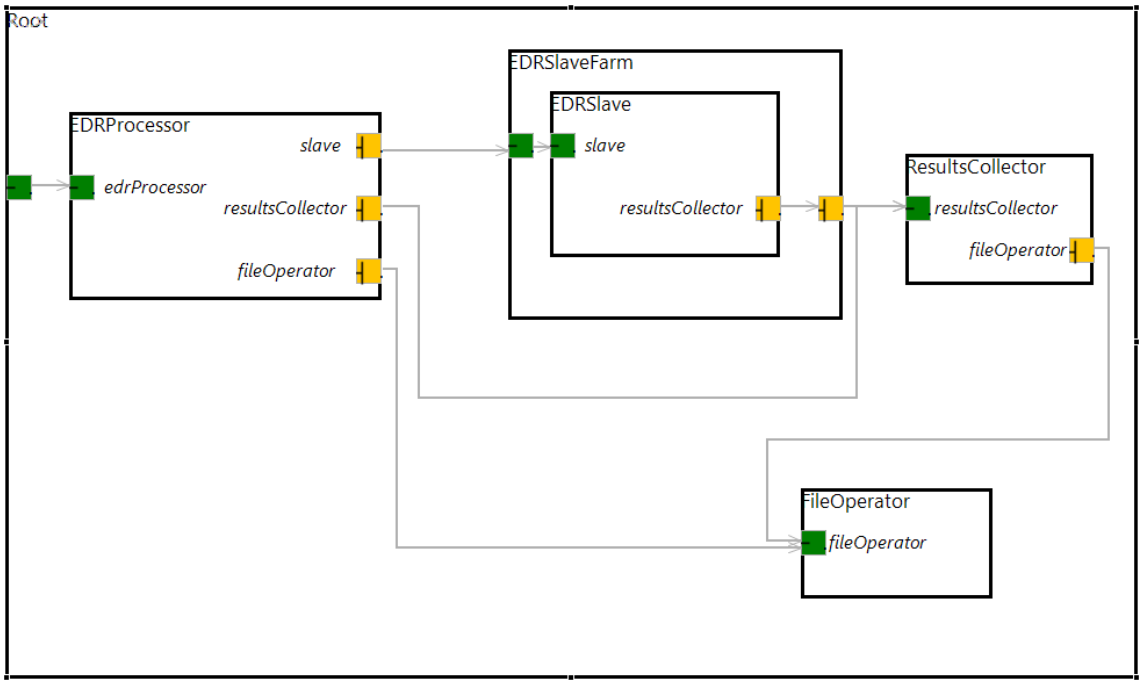
Summarizing, the architectural design is as follows:

- The EDRProcessor receives the request to process an EDR file
- Using the FileOperator, the EDR file is split into fragments.
- Using a multicast interface, the fragments are processed by the EDRSlave components
- The partial results are sent to the ResultsCollector.
- When all fragments have been processed, the ResultsCollector, using the FileOperator, merges the partial results, obtaining the final result.

Comparing this architectural design to the one from the primitive prototype, we can see that the EDRSlave components are now connected to the new ResultsCollector component. This way, they can inform the latter when they have finished processing one of the fragments of the EDR file. This information will allow the ResultsCollector to know the progress achieved, and thus, it can be displayed in the user interface. This new component will be described in depth in the following sections of this document.

#### *4.1.2.1.2 Autonomic*

The current state of the autonomic version of the components diagram is the following:




In this version, the multicast interface between the *EDRProcessor* and the *EDRSlave* components has been replaced by a *TaskFarm* component, the *EDRSlaveFarm*. This component will take care of deploying as many *EDRSlaves* as needed, controlling the parallelism degree. This degree can vary during the execution of the application, in order to adapt the performance to the requirements of the user.
Only the *EDRProcessor* is affected by this change, the rest of the components remain the same. This fact shows the easiness in turning a non-autonomic architecture into an autonomic one.

#### 4.1.2.2 Components description

##### 4.1.2.2.1 EDRProcessor

The *EDRProcessor* acts as the master component, offering a homonym server interface. Through this interface, the GUI can submit processing requests to the components. The behavior of this component is the following:

- Scatters the file using the fileOperator client interface.
- Initializes the *ResultsCollector* component through its interface, telling it how many fragments must be processed.
- Processes the fragments using the slave client interface. The non-autonomic version of the component uses the multi-cast interface and the autonomic one the single-cast interface (connected to the *EDRSlaveFarm*). The more slaves bound to the interface (or the farm), the higher level of parallelization achieved.

##### 4.1.2.2.2 EDRSlaveFarm

This component is only present in the autonomic version of the application. This composite component extends the *MonitorBalanceFarmController* from the NFCF, offering a *farm* of *EDRSlave* components. Using the non functional interfaces of this component, the user can modify the parallelism degree of the application.

##### 4.1.2.2.3 EDRSlave

This is the component in charge of applying the ETL process implemented using Kettle and described in a previous section of this document. When receiving the first request, the Kettle library will be initialized with all the needed configuration files. Subsequent requests will be processed faster, as no initialization has to be performed again.
When Kettle initialization is done, the component transfers the corresponding fragment of the EDR file from the node where the *EDRProcessor* component is deployed. Then, the transformation is applied to the file using the Kettle library. The result is transferred back and the *ResultsCollector* is invoked to notify another fragment has been processed.

##### 4.1.2.2.4 ResultsCollector

This component collects the intermediate results, sent from the *EDRProcessor* components. When all results are collected, they are joined, using the *FileOperator*.
Also, this component offers information about the progress of the processing.

##### 4.1.2.2.5 FileOperator

The *FileOperator* component offers the functionality to scatter and join files. Those files must reside in the local file system.

#### 4.1.2.3 Interfaces

In this section, the interfaces from the different components are presented.

##### 4.1.2.3.1 EDRProcessor

This is the server interface offered by the *EDRProcessor* component:

```
public interface EDRProcessor {

  /**
   * Processes the given EDR input file.
   *
```

```
    * @param inputFilePath path to the EDR input file
    * @param outputFilePath path to store the results file to
    * @param recordsPerFragment number of EDR to include on each fragment of the input file
    * @param gzipFiles whether to compress fragment files when transferring them
    */
  void process(String inputFilePath, String outputFilePath, int recordsPerFragment,
      boolean gzipFiles);
}
```

It contains a single, straight-forward method.

#### *4.1.2.3.2 EDRSlave*

This is the server interface offered by the *EDRSlave* components:

```
public interface EDRSlave {

  /**
   * Processes an EDRRequest.
   * <p>
   * This means:
   * <UL>
   * <LI>Downloading the given fragment of the EDR file from source node</LI>
   * <LI>Processing all the contained EDRs, generating a partial results file</LI>
   * <LI>Uploading the partial results file to the source node</LI>
   * <LI>Calling the ResultsCollector component to let it know processing is finished</LI>
   * </UL>
   *
   * @param request the EDRRequest to be processed
   */
  void process(EDRRequest request);
}
```

An `EDRRequest` contains the path to one of the fragments of the EDR input file, the node where it is stored, the path to store the partial results to, and whether to compress those files when transferring them.

#### *4.1.2.3.3 EDRSlaveMulticast*

This is the multi-cast client interface used by the non-autonomic version of the *EDRProcessor* component to invoke the *EDRSlave* components.

```
public interface EDRSlaveMulticast {

  /**
   * Processes a list of EDRRequests, using Round Robin parameter dispatch mode.
   * <p>
   * This means:
   * <UL>
   * <LI>Downloading EDR fragment files from source node</LI>
   * <LI>Processing all the contained EDRs, generating partial results files</LI>
   * <LI>Uploading the partial results files to the source node</LI>
   * <LI>Calling the ResultsCollector component to let it know processing is finished</LI>
   * </UL>
   *
   * @param requests the list of EDRRequest to be processed
   */
  @MethodDispatchMetadata(mode = @ParamDispatchMetadata(mode = ParamDispatchMode.ROUND_ROBIN))
  void process(List<EDRRequest> requests);
}
```

#### *4.1.2.3.4 FileOperator*

This interface contains the needed operations with files: `scatter()` and `join()`.

```
public interface FileOperator {

  /**
```

```
   * Scatters the given file into fragments of the given size (expressed in number of
   * lines/Extended Data Records).
   * <p>
   *
   * @param source the file to be scattered
   * @param linesPerFile size of the fragments
   * @param gzipFiles whether to gzip fragments
   * @return List of the resulting Files
   */
  List<File> scatter(File source, int linesPerFile, boolean gzipFiles);

  /**
   * Joins the given source files into the given output one.
   *
   * @param sources list of files to be joined
   * @param outputFile where to save the resulting file
   * @param gzipFiles whether the source files are gzipped
   */
  void join(List<File> sources, File outputFile, boolean gzipFiles);
}
```

#### *4.1.2.3.5 ResultsCollector*

The server interface offered by the new ResultsCollector component is the following:

```
public interface ResultsCollector {

  /**
   * Initializes the ResultsCollector component.
   *
   * @param numOfResultFilesToCollect number of partial results to be collected
   * @param resultsFile path to the results file
   * @param gzipFiles whether the files are compressed
   */
  void init(int numOfResultFilesToCollect, File resultsFile, boolean gzipFiles);

  /**
   * Informs the collector a new partial result is available, providing its location.
   * <p>
   * If the number of results files to be collected has been reached, this triggers the
   * joining of the partial results files into the final result, using the FileOperator.
   *
   * @param remoteFile path to the new partial result.
   */
  void collect(File remoteFile);

  /**
   * Gets the number of partial result files already collected.
   *
   * @return the number of partial result files already collected.
   */
  IntWrapper getResultFilesCount();

  /**
   * Gets the number of partial result files to be collected.
   *
   * @return the number of partial result files to be collected.
   */
  IntWrapper getResultFilesToCollect();
}
```

The `init()` method is invoked from the *EDRProcessor* component, the `collect()` method from the *EDRSlave* components, and the `getResultFilesCount()` and `getResultFilesToCollect()` from the Graphical User Interface, in order to provide feedback about the progress of the processing.

### *4.1.2.4 Summary of the GridCOMP features used*

The EDR Processor use case is making good use of the following GridCOMP features:

- Primitive components: as seen previously, the prototype features several primitive components (EDRProcessor, FileOperator, EDRSlave).

- Composition: the whole prototype is a composite prototype (Root). Also, the EDRSlaveFarm is a composite component, containing multiple instances of EDRSlave components.
- Collective interfaces: a multi-cast interface is being used by the non-autonomic version of the architectural design to connect the EDRProcessor component to the EDRSlave components and carry out the processing of the fragments of the EDR input file.
- Autonomic features: a task farm (from WP3) takes care of EDRSlave components replication in the autonomic version of the architectural design.
- Grid Integrated Development Environment: the architectural designs have been done using the GIDE prototype from WP4.

## 4.2 Early prototype

### 4.2.1 Description

The early prototype of the EDR Processor use case application fixes most of the limitations the primitive one had:

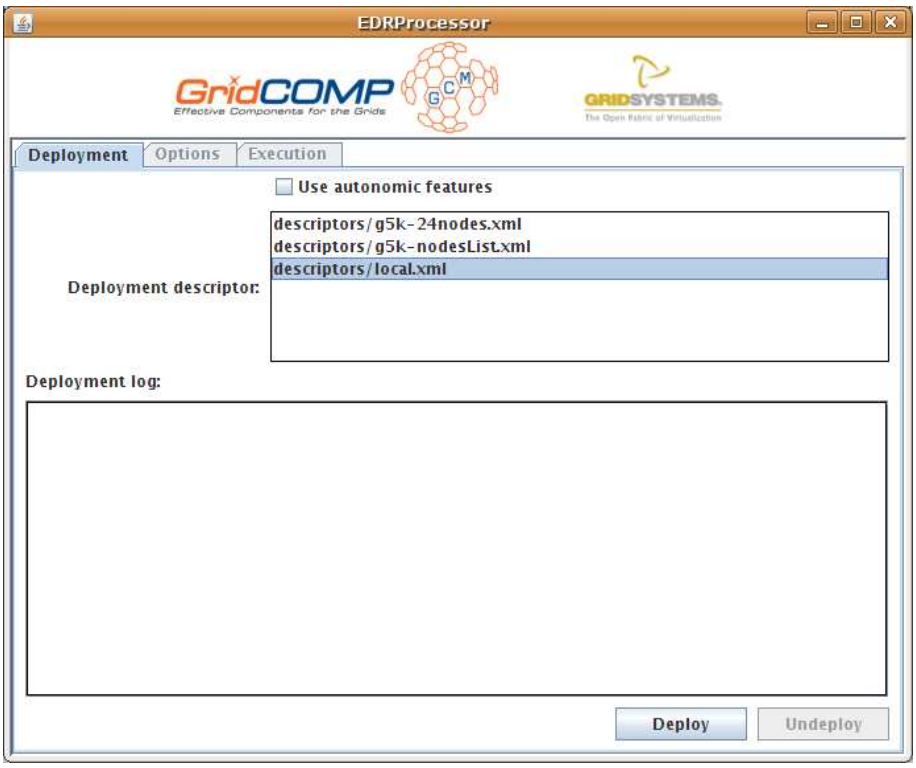
- Includes a graphical user interface, letting the user select all the invocation parameters.
- Progress information is displayed through the user interface, as a progress bar.
- The implementation of the EDR processing has been completed, using Kettle.
- Dynamic deployment without editing the architecture, using task farm autonomic controller or a programmatic approach.

### 4.2.2 Configuration and usage

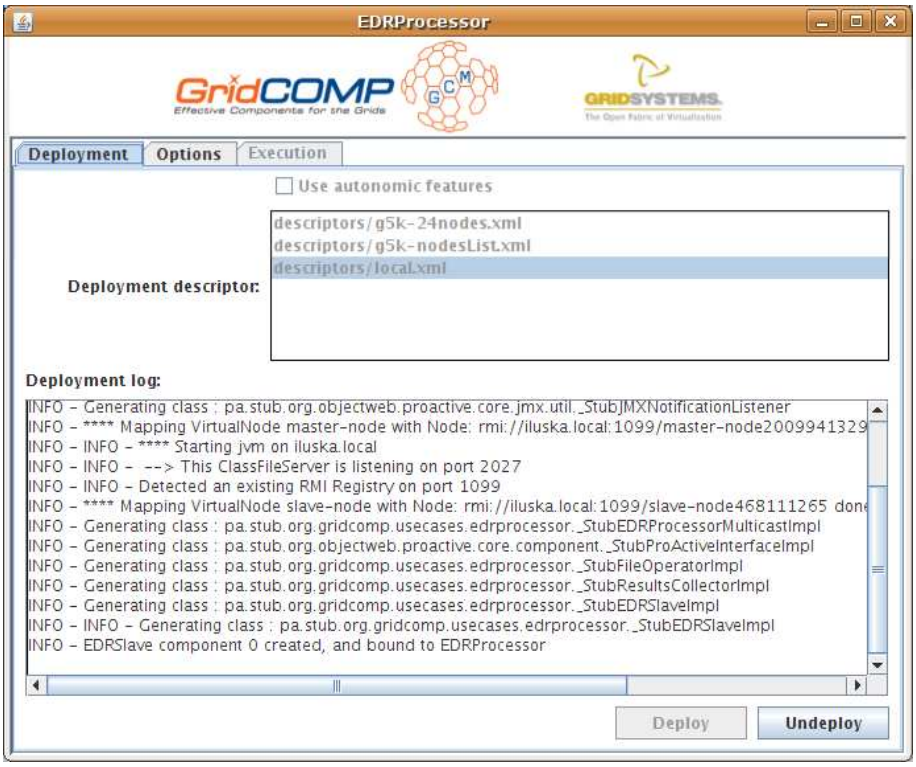
Both the source code and the binaries of the early prototype are included in the file "D.UC.04 – EDR Processor early prototype.zip". The latest version of this prototype is also publicly available at INRIA's GForge `gridcompwp5gs` project [9].
These are the system requirements in order to run the application:

- Java 1.6 [4]
- Ant [5]
- ProActive 3.90 [6]

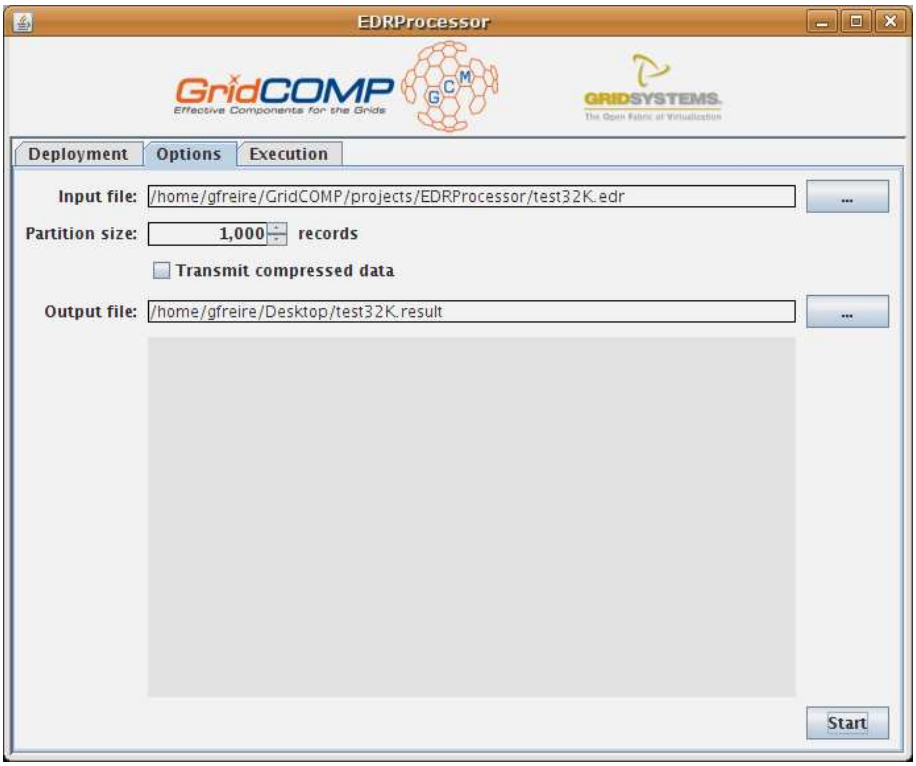
After uncompressing the aforementioned zip file, and assuming that both java and ant are in the path, just type `ant processor` to invoke the EDR Processor. The application will request to enter the path to the distribution folder of ProActive 3.90. After that, the user interface will appear:




This first "tab" contains the deployment details. Depending on your infrastructure, select one of the included deployment descriptors and press the "Deploy" button.
The "Deployment log" text box will show the log trace output during the deployment:



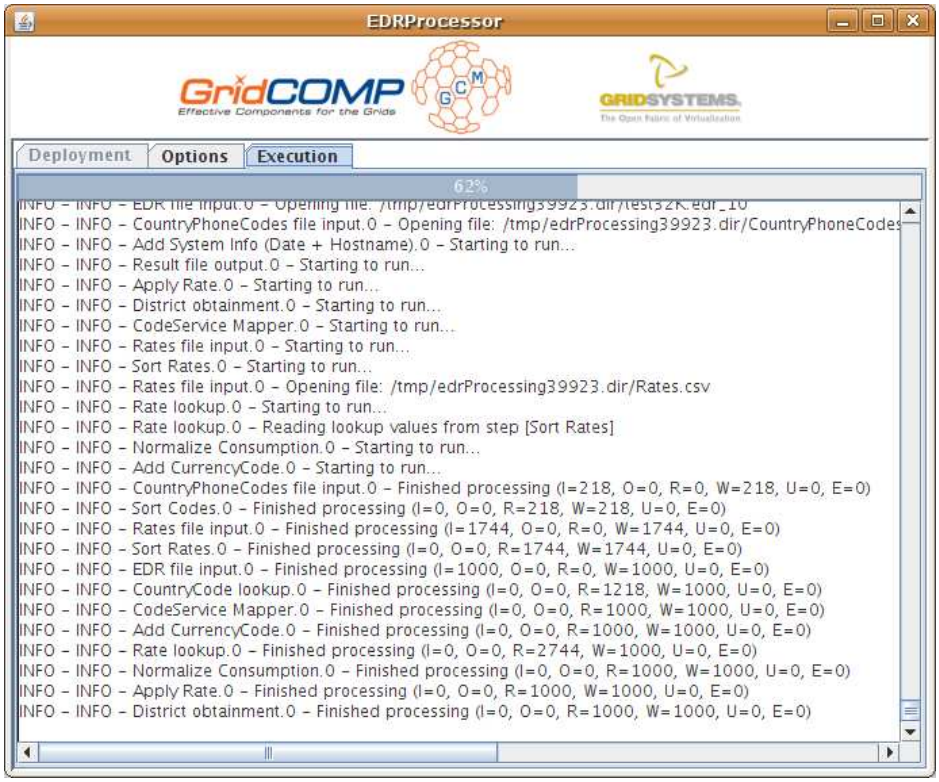

After the deployment is done, the "Options" tab is enabled:




The “options” tab contains the controls to select the desired input parameters:

- Input file: path to the file containing the EDRs to be processed.
- Partition size: number of EDRs each fragment file will contain.
- Transmit compressed data: whether to compress the fragments of the input file before transferring them. This may reduce the time needed to transfer the data through the network.
- Output file: path to the file where the results of the processing will be stored.

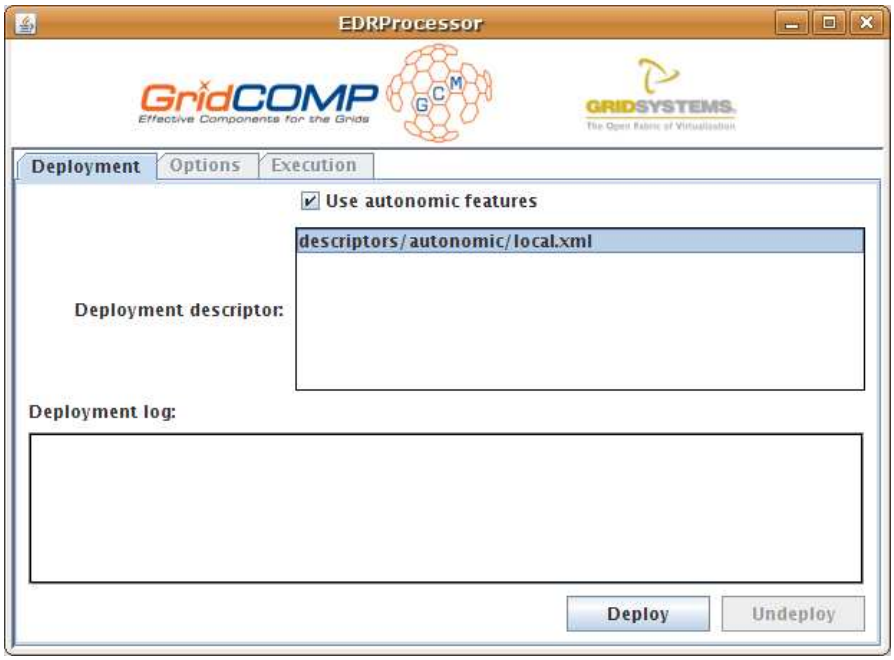
When all of the above fields have been complimented, the “Start” button can be pushed. The request will be submitted to the components, and the “Execution” tab will be enabled, showing the log trace of the execution and a progress bar.



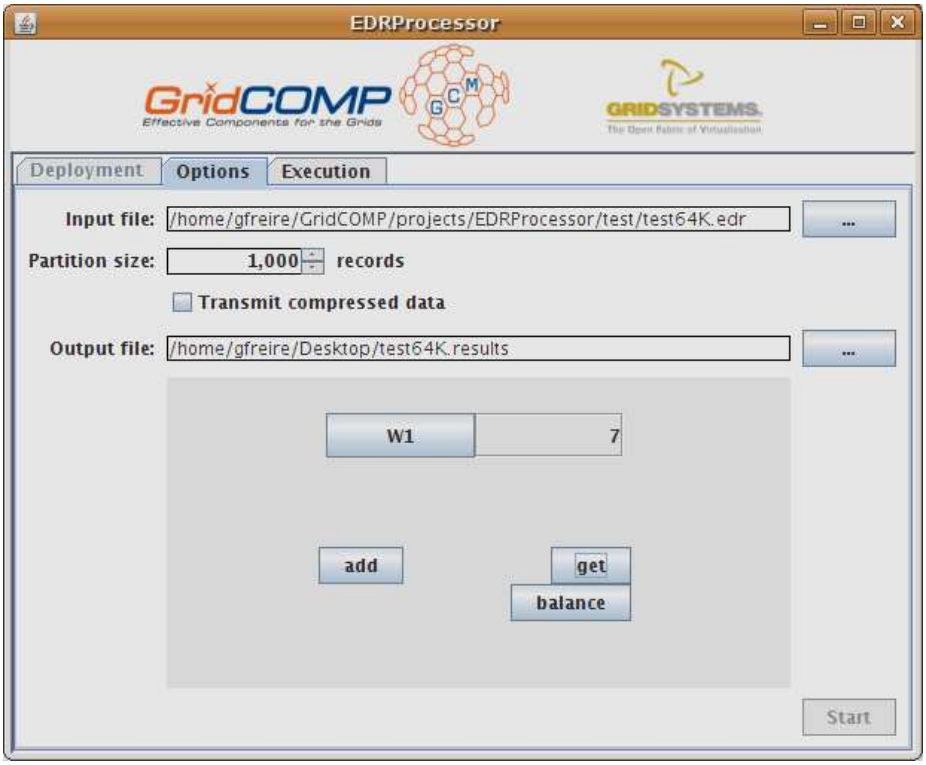

When the progress bar reaches 100%, the execution is done (all fragments from the EDR file have been processed and their results joined), and new requests can be submitted.

#### *4.2.2.1.1 Autonomic version*

In order to test the autonomic version of the application, the “Use autonomic features” check box must be checked in the “Deployment” tab, and one of the specific deployment descriptors must be selected (at the time of this writing only a local deployment descriptor is offered).




While running a request, a set of controls will be displayed in the "Options" tab. Using these controls, the autonomic behaviour of the application can be monitored and/or altered, adding or removing workers.



### 4.2.3 Examples

The /test folder contains several sample input files (generated using the script provided on the same folder), ranging from one thousand EDRs to one million EDRs. If needed, more files can be generated, invoking the EDRGenerator tool (`ant generator` in the main folder). Generating random EDR files is a time consuming task so, in order to create a new file, it is advised to use one of the included ones to repeatedly append it to the new one.

## 4.3 Next actions

For this use case, the planned actions in order to turn the early prototype into the final, "tuned", one are:

- Finish the integration with the autonomic controller, providing a way for the user to specify QoS (Quality of Service) requirements for the execution of the experiments.
- Measure the performance of the application, running on different deployments and with different parameters.
- Clean up and document in depth the source code.

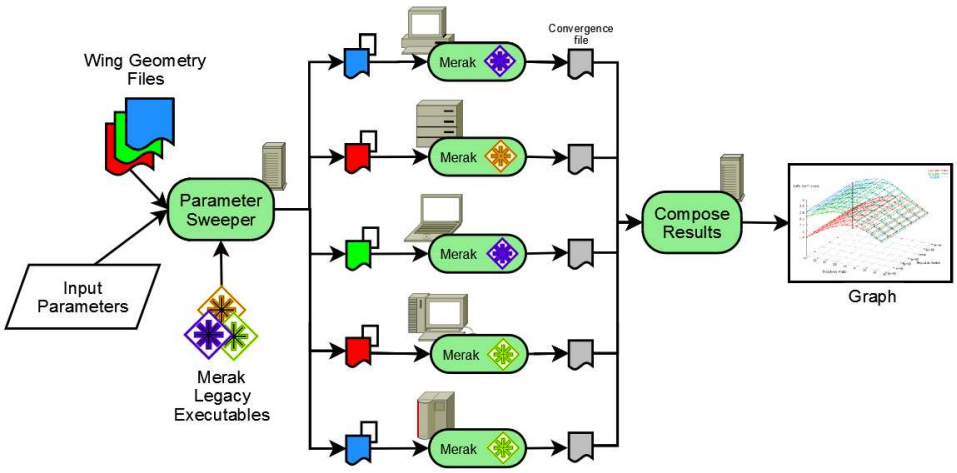
# 5 Wing Design

## 5.1 Architectural design

A complete description of this use case and its background can be found in D.UC.03 deliverable [1]. This section will only include a summary of the architectural design and an update from the previous deliverable.

### 5.1.1 Architecture of the application

Although not changed from previous versions of the prototype, for clarity, a depiction of the operation of the Wing Design application is offered next:




1. The user provides a set of wing geometry files and input parameters for the experiment.
2. The legacy application binaries (Merak) are provisioned to the resources on the grid, as new components.
3. The complete set of parameter combinations is obtained by the Parameter Sweeper component.
4. Each parameter combination is sent to a Merak component, which performs its simulation.
5. Results are collected, composed and a graph is generated.

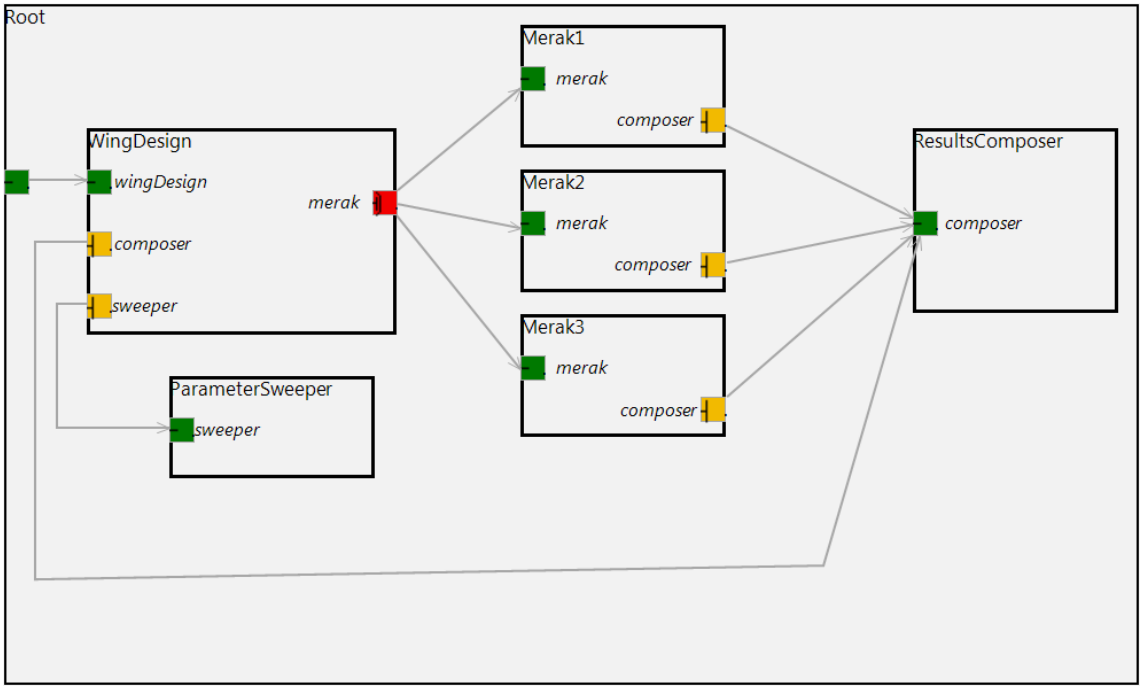
Legacy executable files are only available for Windows, Linux and Solaris, so resources running these operating systems are needed. Also, result graph is generated using gnuplot, which must be installed in the computer running the application.
In order to provide a better user experience, an interactive graph is built during the execution (per wing geometry involved). The user can change the point of view; zoom in and out, etc. These interactive graphs are generated using Visad[7], which is based on Java3D[8] that must be present in the computer running the application.

## 5.1.2 GCM Components

### *5.1.2.1 Components diagram*

#### *5.1.2.1.1 Non-autonomic*

The non-autonomic components diagram (made using the Grid IDE) corresponding to the early prototype is the following:




Note: Although three Merak components are depicted, the exact number depends on the deployment of the application.

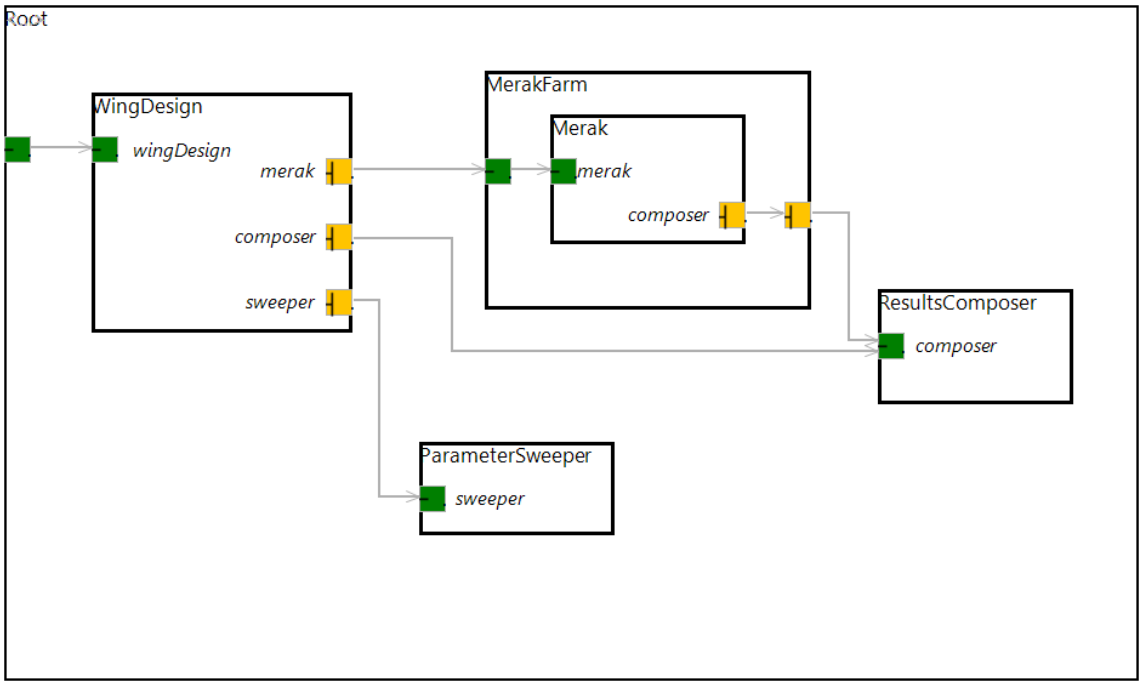
A brief explanation of this architectural design is the following:

- The *WingDesign* receives the request to perform a simulation, given a set of wing configurations and input parameters.
- Using the *ParameterSweeper*, the complete list of parameter combinations to evaluate is obtained.
- The above information is also passed to the *ResultsComposer*.
- Each one of the parameter combinations is delivered to a *Merak* component, using the multicast interface.
- Results are delivered to the *ResultsComposer*. When all results are received, the graph showing the comparison of the results is made.

The only change in the architectural design from its preliminary version is the direct connection between *Merak* components and the *ResultsComposer*. Now, the *ResultsComposer* is also in charge of providing information about the progress of the computations and of gathering the results in order to show in-progress graphs for each wing geometry.

#### 5.1.2.1.2 Autonomic

The current state of the autonomic version of the components diagram is the following:




This autonomic architectural design makes use of a Farm controller (the *MerakFarm)*, replacing the multicast interface between the *WingDesign* and the *Merak* components. The *MerakFarm* controls the parallelism degree of the application, deploying as many *Merak* components as needed.

As in the previous use case, only one component, *WingDesign*, is affected by this change, the rest of the components remain the same.

### 5.1.2.2 Components description

#### 5.1.2.2.1 WingDesign

This is the main component, and offers a server interface, named wingDesign, which is used by the graphical user interface. Through this interface the user can submit simulation requests, providing one or more wing geometry files and the needed input parameters.
The behaviour of this component is quite simple:

- Initializes the ResultComposer component with the input parameters.
- Initializes the Merak components, with the appropriate legacy application binaries for their platform.
- Obtains the list of all the parameter combinations that must be processed, calling the ParameterSweeper component.
- Processes the list of parameter combinations, invoking the Merak components (whether through the multicast interface or the farm controller).

#### 5.1.2.2.2 ParameterSweeper

This component computes the complete list of parameter combinations to be processed. This is simply the Cartesian product of:

- The range of incidence angles
- The range of Reynolds numbers
- The range of wing configurations

This component must be co-allocated with the *WingDesign* one, as it needs local access to the wing geometry files.

#### *5.1.2.2.3 MerakFarm*

As in the previous use case, this component is only present in the autonomic version of the application. This composite component extends the *MonitorBalanceFarmController* from the NFCF, offering a *farm* of *Merak* components. Using the non functional interfaces of this component, the user can modify the parallelism degree of the application.

#### *5.1.2.2.4 Merak*

The *Merak* component wraps the legacy application:

- Downloads the proper executable files from the master node on initialization (this is only done once).
- Processes each received request for execution, preparing the input parameters, invoking the executable, transferring the result file
- Deletes temporary files after finishing the execution.

#### *5.1.2.2.5 ResultsComposer*

The *ResultsComposer* gathers the result files from the simulations, generating a graph where the different wing geometries are compared. It also offers information about the progress of the process and temporary results, allowing the graphical user interface to display that results *live*.

### *5.1.2.3 Interfaces*

In this section, the interfaces from the different components are presented.

#### *5.1.2.3.1 WingDesign*

This is the interface offered by the WingDesign component. Its only method is the starting point of the whole process:

```
public interface WingDesign {
  /**
   * Performs a simulation, processing the given parameter specification.
   *
   * @param spec contains the wing geometries, incidence angles, reynolds and iteration
   *        numbers to perform the simulation
   * @return a graph comparing the different wing geometries.
   */
  File process(SweepSpecification spec);
}
```

A `SweepSpecification` is a java class that contains a list of geometry files, the initial and final values for the incidence angle and the reynolds number ranges, the number of samples to take from each range, and a number of iterations to perform the simulation.

#### *5.1.2.3.2 ParameterSweeper*

This interface contains the method to generate the list of parameter combinations to be used when invoking the legacy application (merak):

```
public interface ParameterSweeper {
  /**
   * Generates the combination of parameters to be submitted to merak.
   * <p>
   * This is the Cartesian product of the ranges contained in the sweep specification
   *
   * @param spec the specification of the ranges of values to be swept
   * @return a list of Merak Parameters
   */
  List<MerakParameters> sweep(SweepSpecification spec);
}
```

#### *5.1.2.3.3 Merak*

The interface of the component offering access to the legacy application is the following:

```
public interface Merak {

  /**
   * Prepares the component for the execution of the Merak legacy application.
   *
   * @param supplier the node to download the executable files from
   * @param platformFiles map with the executable files per platform.
   * @param resultsDir path on the supplier node where result files must be stored
   */
  void prepare(Node supplier, Map platformFiles, File resultsDir);

  /**
   * Runs the Merak legacy application, using the given parameters.
   *
   * @param params parameters to pass to the legacy application
   */
  void run(MerakParameters params);

  /**
   * Deletes all temporary files stored in the local file system during preparation or
   * execution.
   */
  void cleanUp();

}
```

#### *5.1.2.3.4 MerakMulticast*

This is the multi-cast client interface used by the non-autonomic version of the *WingDesign* component to invoke the *Merak* components:

```
public interface MerakMulticast {

  /**
   * Prepares the component for the execution of the Merak legacy application.
   *
   * @param supplier the node to download the executable files from
   * @param platformFiles map with the executable files per platform.
   * @param resultsDir path on the supplier node where result files must be stored
   */
  void prepare(Node supplier, Map platformFiles, File resultsDir);

  /**
   * Runs the Merak legacy application, using the given list of parameters, using Round
   * Robin parameter dispatch mode.
   *
   * @param params the list of parameters to pass to the legacy application
   */
  @MethodDispatchMetadata(mode = @ParamDispatchMetadata(mode = ParamDispatchMode.ROUND_ROBIN))
  void run(List<MerakParameters> params);

  /**
   * Deletes all temporary files stored in the local file system during preparation or
   * execution.
   */
  void cleanUp();
}
```

The only method offering a multicast behaviour is `run`. This is dispatched in a round robin fashion among the *Merak* components.

#### *5.1.2.3.5 ResultsComposer*

The server interface offered by the *ResultsComposer* component is the following:

```
public interface ResultsComposer {

  /**
   * Initializes the component.
   *
   * @param spec the input parameters to be processed
   * @param resultsDir the path to the directory where temporary results will be stored
   */
  void init(SweepSpecification spec, File resultsDir);

  /**
   * Adds a new result (a point in the graph for a certain wing geometry) to the composer.
   * When all results are added, the graph comparing the different wing geometries is
   * made.
   *
   * @param geoPoint a point in the graph for a certain wing geometry
   */
  void addResult(GeometryPoint geoPoint);

  /**
   * Gets the percentage of results received since the component was initialized.
   *
   * @return the percentage of results received since the component was initialized
   */
  Integer getResultsPercent();

  /**
   * Gets a list with all the results added since the last call to this method.
   *
   * @return a list with all the results added since the last call to this method
   */
  List<GeometryPoint> getPendingResults();
}
```

The `init` method is invoked by the *WingDesign* component at the start of the process. The `addResult` method is invoked by the *Merak* components each time a new result is obtained. Last, `getResultsPercent` and `getPendingResults` are invoked from the GUI in order to implement a progress bar and the interactive *live* graphs, respectively.

### *5.1.2.4 Summary of the GCM features used*

The following GridCOMP features are used:

- Primitive components: for example *WingDesign*, *Merak*, *ResultsComposer*, etc.
- Composition: *MerakFarm* is a composite component, containing multiple instances of *Merak* components.
- Collective interfaces: the non-autonomic version of the architectural design uses a multicast interface to connect the *WingDesign* component to the Merak ones, distributing the simulation effort among them.
- Autonomic features: a task farm (from WP3) takes care of *Merak* components replication in the autonomic version of the architectural design.
- Grid Integrated Development Environment: the architecture has been designed using the GIDE prototype from WP4.

## 5.2 Early prototype

### 5.2.1 Description

Very similar to the EDR Processor, the early prototype of the Wing Design use case fixes most of the limitations the primitive one had:

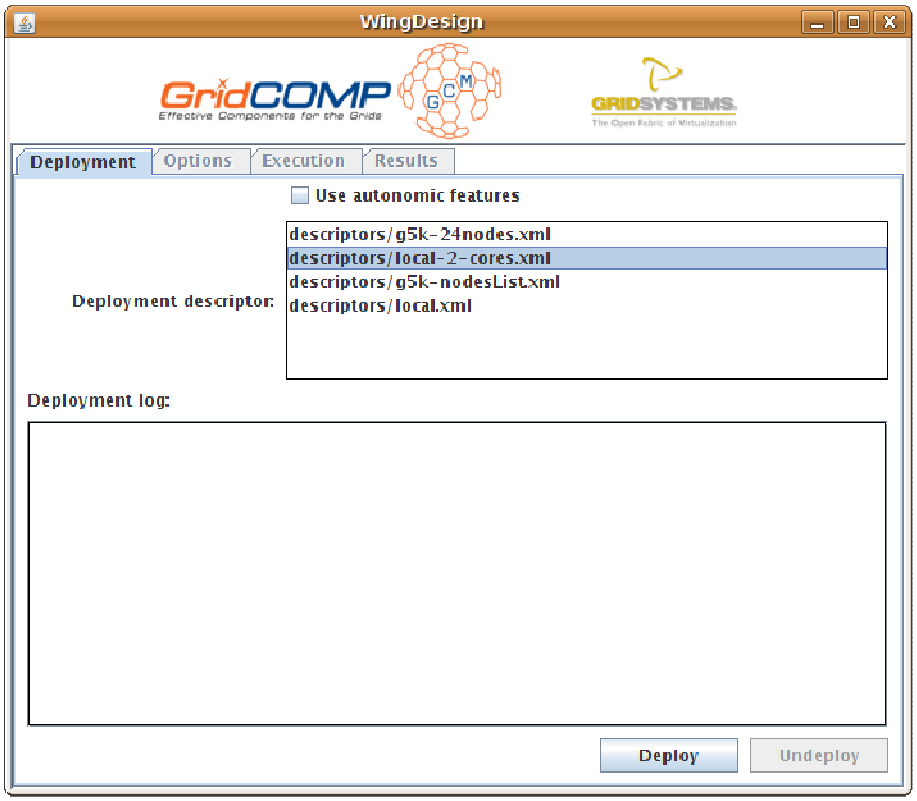
- Includes a graphical user interface, letting the user select all the invocation parameters.
- Progress information is displayed through the user interface, as a progress bar.
- The output of the results has been improved, displaying *live* interactive graphs for each wing geometry.
- Dynamic deployment without editing the architecture, using task farm autonomic controller or a programmatic approach.

### 5.2.2 Configuration and usage

The file "D.UC.04 – Wing Design early prototype.zip" contains both the source and the binaries of the early prototype. The latest version of this prototype is also publicly available at INRIA's GForge `gridcompwp5gs` project [9].
These are the system requirements in order to run the application:

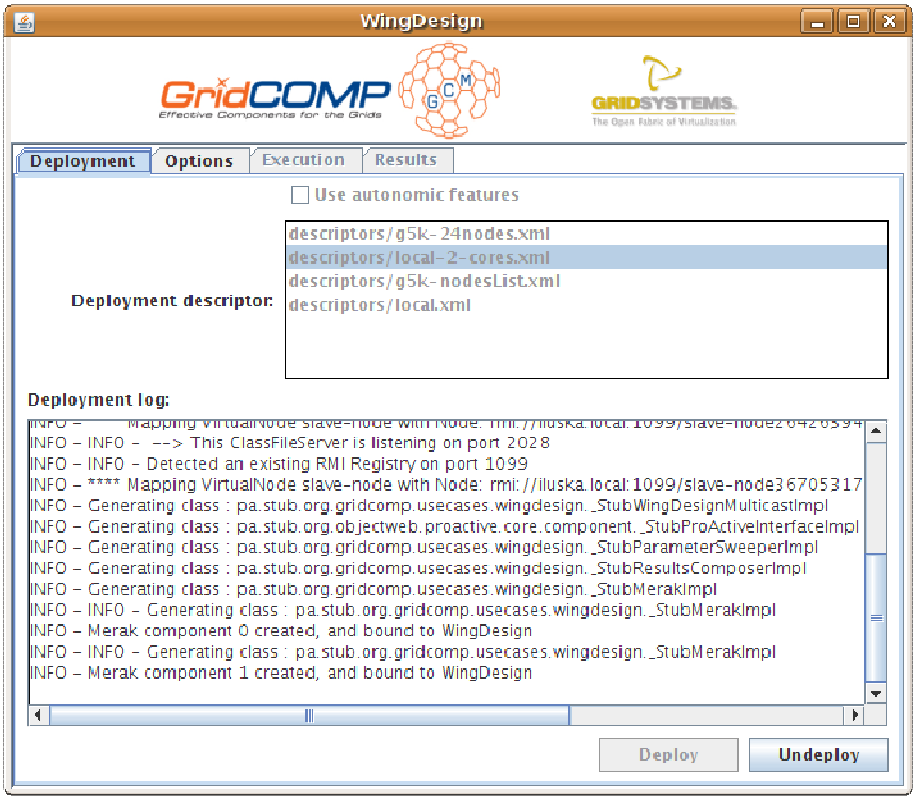
- Java 1.6 [4]
- Java3D [8]
- Ant [5]
- ProActive 3.90 [6]

Run `ant WingDesign` to invoke the Wing Design. The application will request you to enter the path to the distribution folder of ProActive 3.90. After that, the user interface will appear:



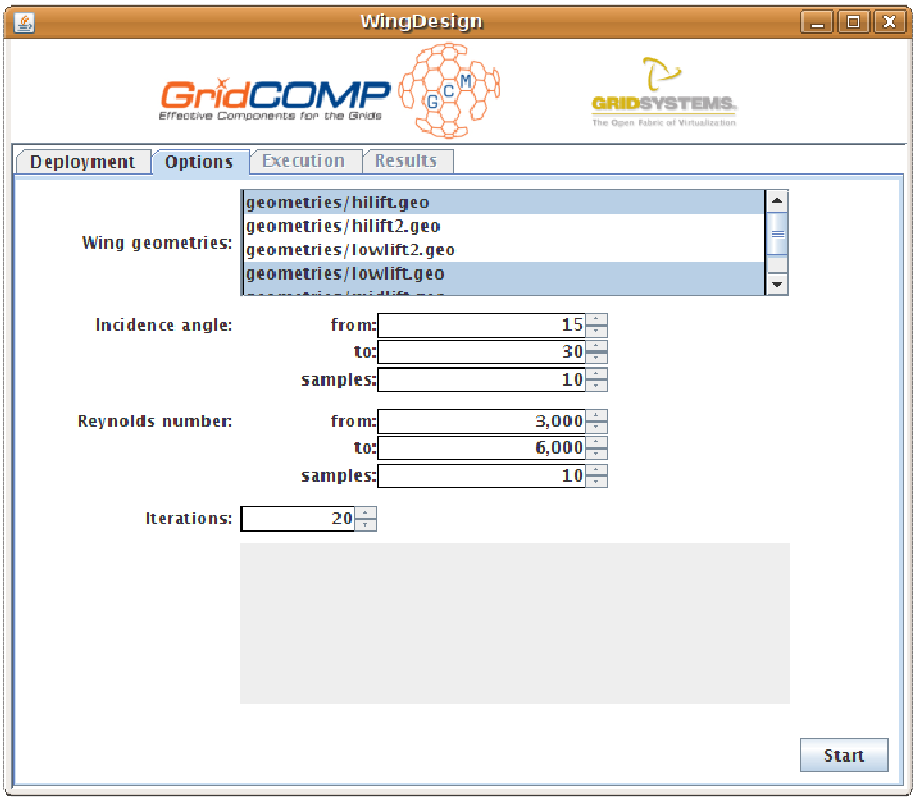

This first “tab” contains the deployment details. Depending on your infrastructure, select one of the included deployment descriptors and press the “Deploy” button.

The “Deployment log” text box will show the log trace output during the deployment:

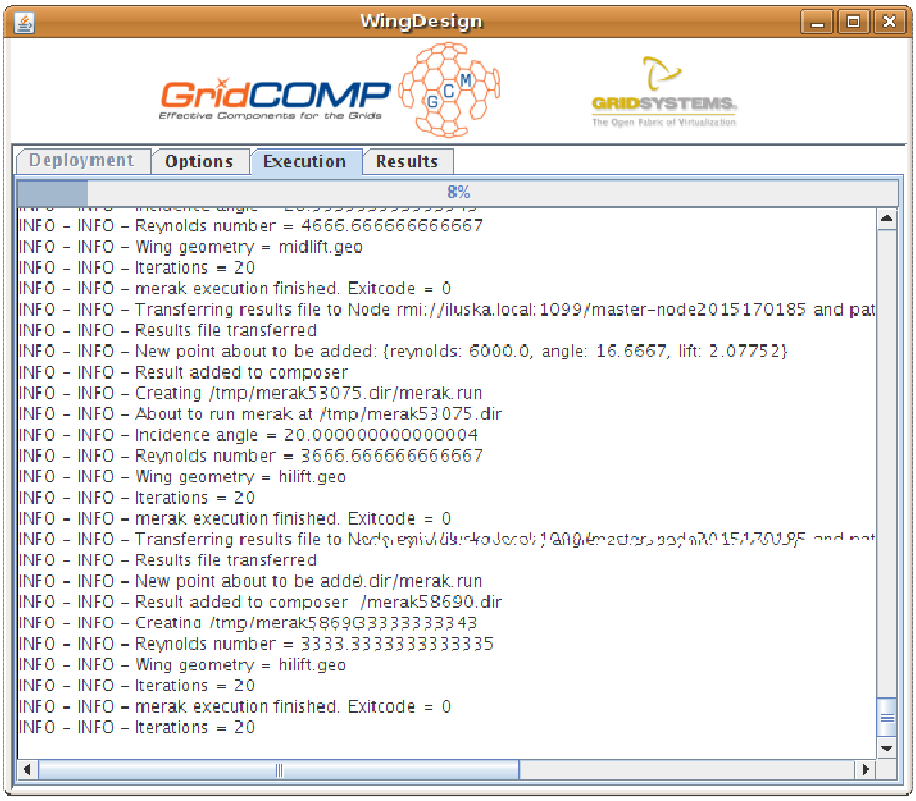



After the deployment is done, the “Options” tab is enabled. This tab includes controls to select the input parameters:

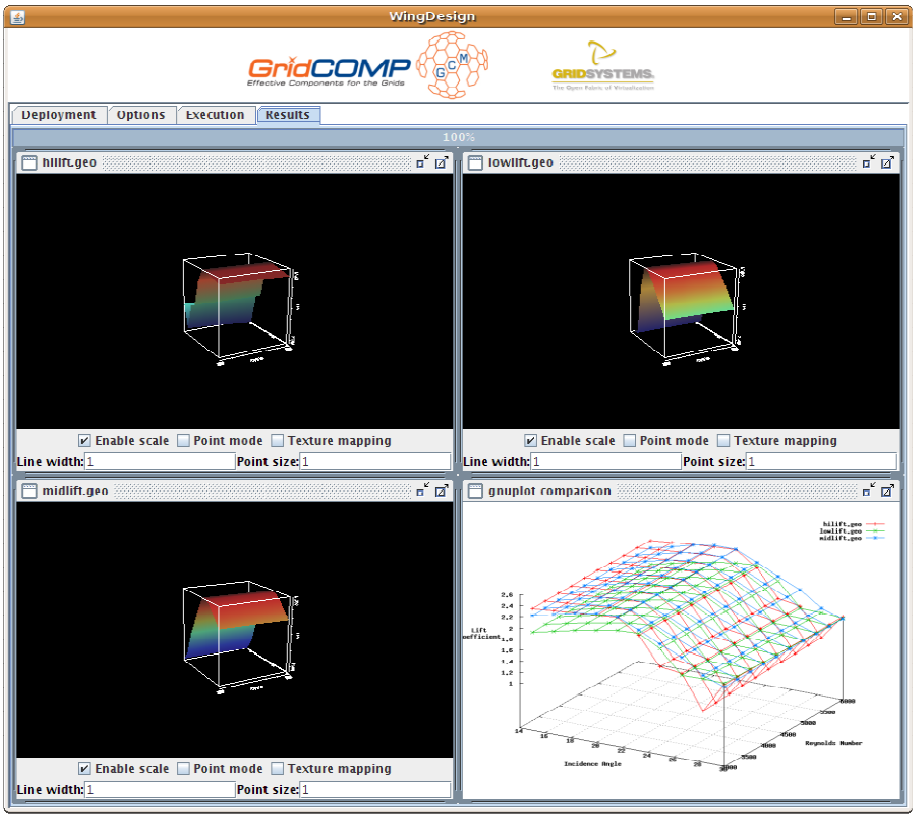
- Wing geometries: all .geo files found in the geometries folder are listed; one or more can be selected (Ctrl + click).
- Range of incidence angle: from, to, and number of samples
- Range of Reynolds number: from, to, and number of samples
- Number of iterations




The “start” button submits the request, when pressed. The “Execution” and “Results” tabs are enabled. The former shows log messages and a progress bar:



The “Results” tab displays the interactive graphs, a progress bar and the gnuplot comparison graph (only when finished):

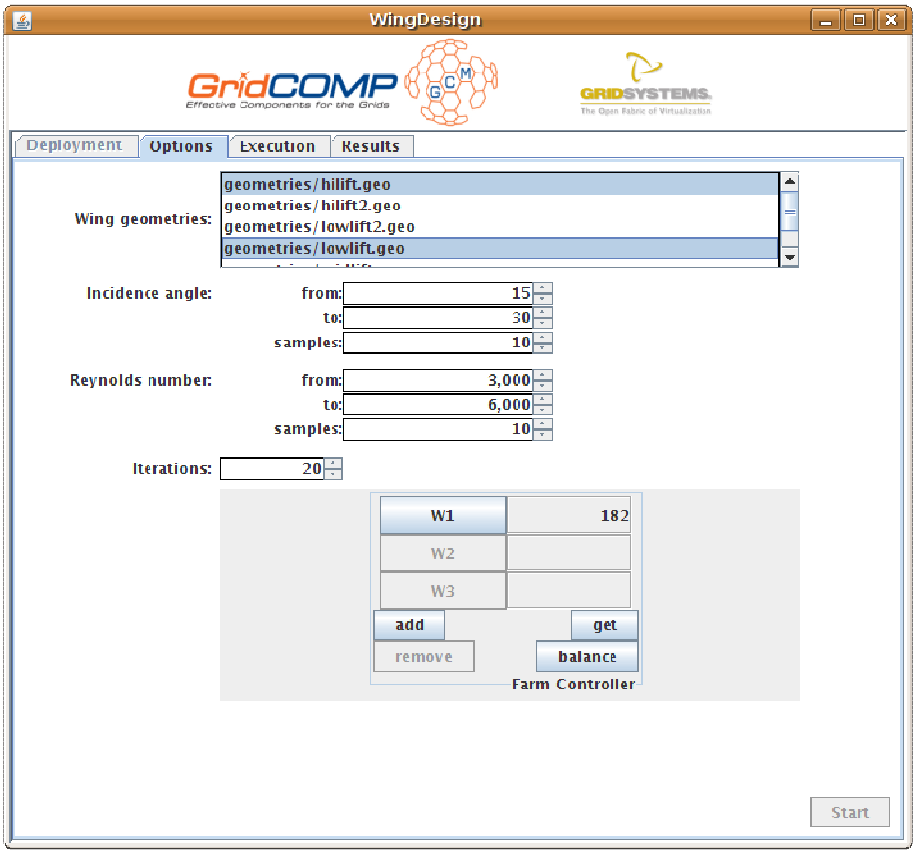



In order to control the interactive graphs:

- Dragging the mouse will rotate the point of view
- Pressing caps while dragging the mouse up or down will zoom in or out.
- Pressing ctrl while dragging the mouse will move the point of view
- The controls at the bottom of each window will change the appearance of the corresponding graph.

#### *5.2.2.1.1 Autonomic version*

In order to test the autonomic version of the application, the “Use autonomic features” check box must be checked in the “Deployment” tab, and one of the specific deployment descriptors must be selected (at the time of this writing only a local deployment descriptor is offered). While running a request, a set of controls will be displayed in the “Options” tab. While using these controls, the autonomic behaviour of the application can be monitored and or altered, adding or removing workers.




### 5.2.3 Examples

The early prototype includes a few wing geometry files to be used for testing (under the geometries folder). The more files that are included or the more samples that are selected for the incidence angle or Reynolds number, the higher the amount of invocations to the legacy application. Changing the amount of iterations will also increase or decrease the time needed to accomplish each invocation (the higher, the longer). Default values should take a few minutes to complete for a local deployment if only a wing geometry is selected.

## 5.3 Next actions

These are the next actions in order to turn the current prototype into the final one:

- Make use of the methods and techniques for legacy code wrapping as Grid Components. Currently, the wrapping is ad hoc and follows no standard.
- Finish the integration with the autonomic controller, providing a way for the user to specify QoS (Quality of Service) requirements for the execution of the experiments.
- Measure the performance of the application, running on different deployments and with different parameters.
- Clean up and document in depth the source code.

# 6 References

[1] T. Weigold, F. Tumiatti, E. Prunés, J. Santacatalina, G Freire. D.UC.03 Use cases description: preliminary architectural design and primitive prototypes. https://bscw.ercim.org/bscw/bscw.cgi/d315688/D.UC.03-final.pdf

[2] T. Weigold, P. Buhler, J. Thiyagalingam, A. Basukoski, V. Getov. Advanced Grid Programming with Components: A Biometric Identification Case Study. Proceedings of COMPSAC 2008, IEEE Digital Library (to appear).

[3] Pentaho Data Integration: http://kettle.pentaho.org/

[4] Java 1.6: http://java.sun.com/

[5] Apache Ant: http://ant.apache.org/

[6] ProActive 3.90: http://proactive.inria.fr/

[7] Visad: http://www.ssec.wisc.edu/~billh/visad.html

[8] Java3D: https://java3d.dev.java.net/

[9] gridcompwp5 project at INRIA's GForge: http://gforge.inria.fr/projects/gridcompwp5gs/