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.NFCF.05 –NFCF tuned prototype and final documentation

Due date of deliverable: December 1st, 2008

Actual submission date: January 19th 2009

Start date of project: 1 June 2006 **Duration:** 33 months

Organisation name of lead contractor for this deliverable: UNIPI

Project co-funded by the European Commission within the Sixth Framework Programme (2002–2006)		
Dissemination level		
PU	Public	PU

Keyword list: autonomic management, component controller, GMC, task farm
Responsible Partner: UNIPI

MODIFICATION CONTROL			
Version	Date	Status	Modifications made by
0	1 December 2008	Draft	Sonia CAMPA
1	5 December 2008	Draft	Marco ALDINUCCI
2	10 December 2008	Draft	Sonia CAMPA
3	10 December 2008	Draft	Marco ALDINUCCI
4	27 December 2008	Draft	Sonia CAMPA
5	8 January 2009	Final	Marco ALDINUCCI

Deliverable manager

- Marco Aldinucci, UNIPI

List of Contributors

- Sonia Campa, UNIPI
- Patrizio Dazzi, ISTI-CNR
- Nicola Tonellotto, ISTI-CNR
- Giorgio Zoppi, UNIPI

List of Evaluators

- Françoise Baude, INRIA
- Rajkumar Buyya, U. MELBOURNE

Executive Summary The current deliverable is intended to 1) describe latest advances of the GCM prototype, which mainly regard advanced autonomic features such as the enhancements to QoS contract expressiveness, the dynamic injection of QoS contracts into the managers, the usage of parametric ADLs and composite component to instance behavioural skeletons; and 2) serve as step-by-step tutorial describing how to develop a GCM autonomic applications by fully exploiting the power of behavioural skeletons. A release of the GCM autonomic component framework is associated with this deliverable (D.NFCF.05-code).

This deliverable concludes the "Non Functional Component Features" deliverable series which have started with the D.NFCF.01 [3] that describes the architectural specification of a framework supporting the GCM autonomic component model [7]. Then, the deliverable D.NFCF.02 [4] has introduced an early prototype implementing the basic features of that specification. This prototype has been enriched with autonomic policies and methodologies to derive performance models in D.NFCF.03 [5] that in turn has led to the refined prototype (with early documentation) that has been described in D.NFCF.04 [6].

Contents

1 Introduction

In this tutorial we present recent advances of the Grid Component Model (GCM)[7, 9] and its reference implementation. Main advances with respect to previous milestone [6] are concerned with:

- the design and implementation of autonomic managers and QoS models driving them;
- the possibility of dynamically inject QoS contracts into the managers;
- the refinement of behavioural skeleton templates and their parametric description with GCM ADL;
- the generalisation of behavioural skeletons, which can now be instanced with both primitive and composite components.
- the introduction of management overlay and distributed management (on-going activity, not fully described in the deliverable).

In the previous tutorial, we gave an initial perspective about the usage of two specific *behavioural skeletons*. They can be used in either a *passive* and *active* fashion, which respectively model user-driven steering and autonomic adaptivity. The reference implementation of GCM has been developed on top of the ProActive 3.9 middleware; the advances presented in this document are related to same version of the middleware. The porting of GCM on top of Proactive 4.0 is currently ongoing.

We recall that behavioural skeletons are high-order, parametric component assemblies. These skeletons can be equipped with specialised management strategies driven by either predefined or user-defined performance goals [1, 2]. The advances described in the rest of the document involve the `farm` and `data parallel` skeletons, which are instances of the *functional replication* skeleton family, i.e. the class of computation patterns that enable the application designer to easily exploit a dynamically variable number of replicas of a guest (primitive or composite) component.

The `farm` skeleton realises the parallel computation of independent tasks parallel pattern, whereas the `data parallel` skeleton exploit the homonym pattern. These skeletons are realised in GCM as particular composite components that exploit a set of *autonomic controllers* implementing the protocols for the correct dynamic adaptation of the skeleton (e.g. life cycle, content, binding, etc.). In particular, adaptation operations are exposed by the *Autonomic Behaviour Controller* (ABC) of the component [2]. These operations can be used to dynamically steer the component behaviour (i.e. passive autonomicity).

Behavioural skeletons (a.k.a. *BeSke*) are obtained by equipping skeletons with an *autonomic manager* (see Fig. 1) exploiting a proper management goal. Truly autonomic components can be obtained as instances of the behavioural skeletons.

The autonomic management happens according to four successive phases. The *monitor* phase provides mechanisms to collect, aggregate, filter and correlate details from managed resource (memory, work load, remote resources, etc.); the *analyse* phase provides mechanisms to observe and analyse behavioural or structural changes (for example a change in the work load of a given machine involved in the application execution); the *plan* phase provides mechanisms to create or select a procedure to change the current status of the observed resources; the *execute* phase applies such procedures by triggering operations directly at the passive level. All these phases are conditioned by a (set of) goal that a user could specify by means of a Quality of Service (QoS) contract.

In [6] we showed how farm and data parallel skeleton can be instanced in a set of use-cases both at the passive and the active level by providing the manager with a set of goals defined by the user in the form of a contract file. In this tutorial we will not focus on specific use-cases but will provide some programming guidelines to describe, extend and instance parametric behavioural skeleton, having also the possibility to change the QoS contract while the application is running.

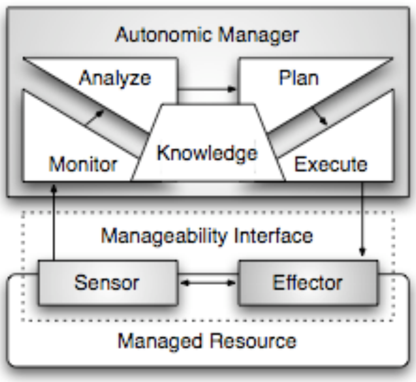

Figure 1: Autonomic computing control loop

1.1 Structure of the code

The framework code is distributed as a zipped archive whose content is summarised in Table 1. The archive contains four main directories (Eclipse projects):

- `GridComp-Core`: contains all the packages representing the core of the framework: interfaces and abstract classes of controllers, managers, autonomic operations, etc.
- `GridComp-Manager`: contains all the classes needed for the implementation of the autonomic manager.
- `GridComp-Farm`: contains all the interfaces, abstract and implementation classes related to the farm behavioural skeleton and its manager implementation.
- `GridComp-DataParallel`: contains all the interfaces, abstract and implementation classes related to the data-parallel behavioural skeleton and its manager implementation.

All these directories are linked to the `GridComp-Libs` directory collecting all the ProActive, Fractal and Java library needed to compile and run the code. The `distribution` directory is used to store all the jar packages produced during the compilation.
The `GridComp-Example` directory contains a set applications exemplifying GCM usage. Those examples show how to develop an application by using farm, data parallel in passive and active fashion, using primitive and composite workers, exploiting centralised or distributed (many-level) management. All the examples are variation of the following base applications:

- *Mandelbrot set:* it's the classic Mandelbrot application exploiting pure task parallelism in which a set of independent tasks are computed at different level of complexity by a set of processing elements represented by the workers of a farm.
- π calculus: as the previous example, this application adopts a task parallelism patterns but use a functional feedback in a cyclic graph fashion.
- *Stateful data parallelism:* it is a data parallel application in which the content of the input data is partitioned and computed by the set of worker components, which may have an internal state provided the worker components expose a method for importing and exporting the internal state. In case of skeleton re-configuration, the state is re-distributed under the coordination of the manager.
- *Stateless data parallelism:* the example is mock-up of the IBM use case [10]. Here, worker components are assumed to be stateless (or including a not disclosed state), thus in the case

GridComp-Core	implementation of the framework core
– `gridcomp.*`	GCM specific classes
– `org.objectweb.proactive.*`	Patch to Proactive classes (overloaded)
GridComp-Manager	abstract classes, interfaces and ADL files for implementing the manager
GridComp-Farm	abstract classes, interfaces and ADL files for implementing the farm behavioural skeleton; prototype implementation of a farm behavioural skeleton supporting active autonomicity
GridComp-DataParallel	abstract classes, interfaces and ADL file for implementing the data parallel behavioural skeleton; prototype implementation of a data parallel behavioural skeleton supporting active autonomicity
GridComp-Example	
– `active.mandelbrot`	evaluation of the Mandelbrot set (autonomic farm)
– `passive.mandelbrot`	evaluation of the Mandelbrot set (adaptive farm)
– `active.pi`	evaluation of the π approximation (autonomic farm)
– `passive.pi`	evaluation of the π approximation (adaptive farm)
– `statelessmap`	data parallel use-case supporting functional reconfiguration (autonomic stateless data parallel)
– `statefulmap`	data parallel use-case with functional reconfiguration (autonomic stateful data parallel)
– `active.twolevels`	evaluation of the Mandelbrot set (autonomic farm with two levels of managements)
– `stateful.twolevels`	data parallel use-case supporting functional reconfiguration (autonomic stateless data parallel with two levels of managements)
– `compworker.farm`	evaluation of the Mandelbrot set (adaptive farm with composite workers)
– `compworker.map`	data parallel use-case with functional reconfiguration (autonomic stateless data parallel with composite workers)
GridComp-Dicom	Dicom use case example for the recognition of X-ray images

Table 1: Content of the framework distribution.

of re-configuration the manager does not provide any automatic way to redistribute the worker state. In case the worker components exhibit a not disclosed state, the redistribution of state can be managed by way of functional interfaces, i.e. providing components with the proper functional ports to manage it.

- *Dicom:* Implements a sequence of (parallel) image filters to highlight possible neoplasms in a stream of x-ray images recorded after the injection of a contrast fluid (e.g. mammography).

2 Getting started

2.1 Platform requirements

In order to compile and run the framework you will need Java 1.5 or upper versions. ProActive 3.9 and all the auxiliary libraries are provided within the zipped archive file.

2.2 Installing the code

In order to install the code, you need to enter your working directory and type:

```
~ > tar -xzvf gridcomp.tar.gz
~ > cd gridcomp/gridcomp
~/gridcomp/gridcomp > ls
Doc            GridComp-DataParallel    GridComp-Farm     build.xml
G5Ktools       GridComp-Dicom-UseCase   GridComp-Libs     commons.xml
GridComp-Core  GridComp-Examples        GridComp-Manager  distribution
```

Opening the archive will lead to the creation into the working directory of a folder called `gridcomp` containing the framework implementation directory, all the files for supporting compilation and running, as well as a `Doc` folder in which a copy of this tutorial can be found.

If you are using Eclipse to work on the code, once you have created your own project into the workspace, you can import the folders through the `Existing Project into Workspace` option.

2.3 Deployment files and compilation

The first step needed to compile the code is to define the correct path in the deployment files. The reference deployment file for a local run is the `local-deployment.xml` file in the `GridComp--Examples` folder, that lists the virtual nodes allocated in the examples and the mapping between virtual nodes and JVMs, JVMs and target architecture.
The reference deployment file for a remote run is the `distrib-deployment.xml` file. This file must be edited in order to customise it to the particular cluster or grid on which it will be used for the run. In particular, the user should update the `NFSHOME` variable in order to point to the `$HOME/gridcomp` folder, the `REMOTE_USER` variable in order to represent the user account name, the `JAVA_HOME` and the `REMOTE_JAVA_HOME` in order to directly point to the Java executable in the system.

Moreover, the user is asked to provide a list of remote hosts and to bind it in the remote deployment file with the `includePropertyFile` tag. Since we have realised a complete independence between the remote deployment file and the target architectures, the `distrib-deployment.xml` file is parametric with respect to the name of the hosts involved in the execution. As a result, in the mapping between each `processDefinition` target and a given host name on which a certain JVM will be activated (via a ssh process), the associated host is provided by the parameter `${HOST_ID_x}`, where $\texttt{x} \in [1, K]$ (being K the number of `processDefinition` tags). At compile time, one of the hosts listed in the file specified by the `includePropertyFile` tag will be assigned to each of these variables. In our distribution, the file listing the hosts is represented by the `cores-pianosa.properties` file and its content is just a trivial list of hosts, one for each line.

The compilation process is supported by the `ant` compiling system. Thus, in order to compile the framework, it suffices to type

```
~/gridcomp/gridcomp > ant
```

which will invoke the compilation steps programmed into the `build.xml` file.

If the user wants to compile one of our example application only, he/she need to enter the directory `GridComp-Examples` and to call the compilation process from the related ant target, thus:

```
~\gridcomp > cd GridComp-Examples
~\gridcomp\GridComp-Examples > ant build-passive-mandelbrot
```

However, when invoked without parameters, `ant` will compile all the provided example, by default.

2.4 Running an example

Once the user has compiled the code, she/he can simply invoke its execution through the `ant` mechanism. We provide two type of deployment descriptors for driving the execution phase: the `local-deployment.xml` file that allows to deploy the application on the user local machine and

RUN_TARGET	::=	DEPLOY_TYPE ∘ ADAPT_MODEL ∘ APPLICATION | DEPLOY_TYPE ∘ ADAPT_MODEL ∘ APPLICATION ∘ EXTRA_CONF
DEPLOY_TYPE	::=	`local-deploy` | `distrib-deploy` | `g5k-deploy`
ADAPT_MODEL	::=	`passive` | `active`
APPLICATION	::=	`mandelbrot` | `pi` | `stateless_map` | `stateful_map`
EXTRA_CONF	::=	`twolevels` | `compworker` | `varcon` | `vardataset`

Table 2: General structure of deployment targets.

local-deploy-active-mandelbrot	distrib-deploy-active-mandelbrot
local-deploy-active-mandelbrot-compworker	distrib-deploy-active-mandelbrot-compworker
local-deploy-active-mandelbrot-twolevels	distrib-deploy-active-mandelbrot-twolevels
local-deploy-active-pi	distrib-deploy-active-pi
local-deploy-active-stateful_map-twolevels	distrib-deploy-active-stateful_map-varcon
local-deploy-active-stateful_map-varcon	distrib-deploy-active-stateful_map-vardataset
local-deploy-active-stateful_map-vardataset	distrib-deploy-active-stateless_map
local-deploy-active-stateless_map-compworker	distrib-deploy-active-stateless_map-compworker
local-deploy-active-statelessmap	distrib-deploy-passive-mandelbrot
local-deploy-passive-mandelbrot	distrib-deploy-passive-pi
local-deploy-passive-pi	g5k-deploy-active-mandelbrot
	g5k-deploy-active-stateful_map-vardataset

Table 3: Predefined deployment targets.

the `distrib-deployment.xml` descriptor that is configured to deploy the application on a cluster. In both cases, she/he simply need to invoke the execution process on the target representing the application she/he is interested in.
Table 2 describes the general grammar of a deployment (and run) targets; Table 3 summarises all the predefined targets for the run of examples provided with the code. As an example, in order to launch the evaluation of the Mandelbrot set at the active autonomicity level, type:

```
~/gridcomp/GridComp-Examples > ant local-deploy-active-mandelbrot
```

3 Non-Functional behaviour: the QoS contract

Behavioural skeletons represent an easy and abstract way to express well-known parallel computation patterns by using predefined composite component. The main advantage of using such kind of component is that the knowledge behind their behaviour specification allows to predefine autonomic operation in order to adapt the component structure and/or behaviour depending on events triggered by the external or internal environment.
All our behavioural skeletons come along with a predefined set of autonomic controllers ensuring the passive level of autonomicity and a version of the autonomic manager related to the specific skeleton. The manager is a component implementing non-functional features. It implements the autonomic cycle (see Fig.1) by evaluating the current behaviour of the application on the basis of a set of requirements described by the user through the QoS contract.

3.1 Structure of a QoS contract

The type of information the manager has to check is provided through the quality of service (QoS) contract. The contract is a set of requirements that the user needs to be satisfied during the application lifetime and includes performance ones as well as other qualitative and/or quantitative information about resource consumption, data sizes, network bandwidth or latency, etc.

The contract is represented by a list of *when <condition> then <action>* JBoss rules [8] structured in the following way:

```
   rule "Symbolic rule name"
     when
       Bean1: <condition_1>
       Bean2: <condition_2>
       ...
       BeanN: <condition_N>
     then
       action_1;
       action_2;
       ...
   end
```

The role of the manager is to periodically evaluate specific monitored conditions (`condition_1, ..., condition_N`) each one encapsulating quantitative information into a (set of) Java Bean (`Bean1, ..., BeanN`), at each `AutonomicLoopCycle` seconds (an attribute of the manager). If one or more hold, the evaluation fires the specific autonomic `action` related to the occurred condition. An example of a QoS contract is given below:

```
package gridcomp.computeTaskMonitor
import gridcomp.manager.beans.*;
import gridcomp.manager.operations.*;
import gridcomp.operation.*;
import gridcomp.example.active.mandelbrot.ManagersConstants;

[methodMonitor="drawLine"]
rule "CheckRateLow"
  salience 1
  when
    $departureBean : DepartureRateBean( value < ManagersConstants.FARM_LOW_PERF_LEVEL )
    $parDegree: NumWorkerBean(value < ManagersConstants.FARM_MAX_NUM_WORKERS)
  then
    $departureBean.setData(new Integer(2));
    $departureBean.fireOperation(ManagerOperation.ADD_EXECUTOR);
end

rule "CheckRateHigh"
  salience 0
  when
    $departureBean : DepartureRateBean( value > ManagersConstants.FARM_HIGH_PERF_LEVEL )
    $parDegree: NumWorkerBean(value > ManagersConstants.FARM_MIN_NUM_WORKERS)
  then
    $departureBean.fireOperation(ManagerOperation.REMOVE_EXECUTOR);
end

rule "StartingParDegree"
  salience 10
  when
    $parDegree: NumWorkerBean(value < 10)
  then
System.out.println("Manager: Starting farm with 10 workers");
$parDegree.setData(new Integer(10));
    $parDegree.fireOperation(ManagerOperation.ADD_EXECUTOR);
    $parDegree.fireOperation(ManagerOperation.BALANCE_LOAD);
end
```

In this contract (see `gridcomp/example/active/mandelbrot/adl/rules.drl`), the manager checks if any of the rules in the QoS contract applies each time the method marked in the contract is evaluated. If a condition holds, the rule is executed. In this example, the "StartingParDegree" rule is applied at the first activation of the manager (i.e. immediately after the start of the applications) since all the farm behavioural skeleton always starts with one worker[1] and this rule exhibits the highest salience. The effect of the rule consist in raising the number of workers up to 10 workers. After the first round, the the two rules "CheckRateLow" and "CheckRateHigh" respectively check that the "DepartureRateBean" value (i.e. the throughput) remain within the `FARM_LOW_PERF_LEVEL` and `FARM_HIGH_PERF_LEVEL`. In the case the first (second) bound is violated, and the farm has not yet reached the `FARM_MAX_NUM_WORKERS` (`FARM_MIN_NUM_WORKERS`), the manager `ADD_EXECUTOR` (`REMOVE_EXECUTOR`) a number of workers to the farm (2 in the example due to the `setData(new Integer(2))`). The `ADD_EXECUTOR` operation should be followed by a `BALANCE_LOAD` operation. The `BALANCE_LOAD` operation is not strictly required after the `REMOVE_EXECUTOR` operation since the tasks queued in the removed worker are anyway distributed to the remaining workers.

3.2 Predefined manager operations

The application developer can exploit the manager functionality by simply writing a QoS contract as a list of JBoss rules. This requires the knowledge of adaptation operations and monitor variables exposed by the particular component. Notice however that, if the component is an instance of a behavioural skeleton (*BeSke*), these operations and variables are typically predefined (and documented) as part of the *BeSke* definition.

The set of autonomic operations (i.e. the actions taken in accordance with given conditions) currently provided are listed in `gridcomp.manager.operation` and are:

- `ADD_EXECUTOR`: adds a new (local or remote) executor to the skeleton configuration (all *BeSke*).
- `REMOVE_EXECUTOR`: removes a (local or remote) executor from the skeleton configuration (all *BeSke*).
- `BALANCE_LOAD`: balances the load of tasks to be executed by a set of executors by reassigning them with respect to the queue distribution defined by the ProActive framework (`farm` only).
- `RAISE_VIOLATION`: signals that a violation contract occurred (all *BeSke*).
- `SETUP`: defines new configuration features of the skeleton (all *BeSke*).
- `GET_EXECUTORS`: provides the number of executors currently involved in the computation (all *BeSke*).
- `PUSH_CONTRACT`: dynamically push a new contract to another manager (used for multi-level management and management overlay, not detailed in this deliverable).
- `PUSH_OBJECT`: dynamically push a new set of bounds to another manager (they are related to a contract installed in another manager; used for multi-level management and management overlay, not detailed in this deliverable).
- `PUSH_LIST`: dynamically push a list of new contracts or bounds to a list of managers (used for multi-level management and management overlay, not detailed in this deliverable).

[1]This restriction is due to the absence of a unicast port in the implementation of Proactive 3.9. This limitation cannot be overcome simply adopting Proactive 4.0 since it implements the unicast port as scheduling policy of the scatter port that can be used only for List data type whereas the farm is supposed to work for any input data type. The current implementation dynamically overload a standard server port with a WP3 implementation of the unicast port thus any farm should start with a standard server port that can bind only one worker (more workers can be added after the overloading is completed, i.e. at run time).

The actions are triggered via the evaluations of Java Beans, as required by the JBoss rules engine. Each time the manager is activated, one or more events (the ones cabled in the implementation of that particular manager) are observed by its monitoring system and for each event a quantitative value is recorded in the related Java bean. As soon as one or more bean values collectively satisfy a *when <condition>*, the related `action` are fired and the operations executed.
The set of currently predefined beans are discussed in Sec. 5.4 and Sec. 4.7. Notice that the user can statically extend the set of beans to access to more monitor data. The introduction of new beans requires a good knowledge of GCM run-time support implementation (see GCM user classification [2]).

3.3 How to change the contract at running time

As the application starts, a contract must be provided to the manager. Each behavioural skeleton accepts a path to the initial contract file among his parameters. Passing such an argument can be statically done by simply setting the proper pair in the hash map used to define the context of the application component. At instancing time, the value of the key will be passed to the `AutonomicManagerAttributeController` of the manager.
In other words, the user will need to define a `java.util.HashMap` and to insert the key `contract-File` associated to the QoS contract path.

```
Map<String, Object> context;
context = new HashMap<String, Object>(1);

// args[0] points to the QoS contract file
context.put("contractFile", args[0]);
String rootADL = "gridcomp.example.active.pi.adl.root";
Factory fact = gridcomp.adl.FactoryFactory.getFactory();
Component root =  (Component) fact.newComponent(rootADL, context);
```

However, a user could need to change a contract at run time and it can be done in two different ways. The first is given by the explicit interaction with the attribute controller responsible for setting the QoS file name. Thus, it suffices the user inserts the following piece of code, where `manager` represents the `Component` instance referencing the manager component:

```
AutonomicManagerAttributeController amac;
amac = (AutonomicManagerAttributeController) manager.getFcInterface("attribute-controller");
amac.setPerformanceContractFile("<path to the new QoS file>");
```

A more conventional strategy is to bound the `commit-contract` interface exposed by the manager and to send the new contract path by calling it. Thus, the preceeding piece of code would be replaced by

```
AutonomicServerManager asm = null;
asm=(AutonomicServerManager) manager.getFcInterface("commit-contract");
asm.commitContract(newcontractPath);
```

4 The data parallel behavioural skeleton (map)

In this section we will show how an application exploiting a data parallelism pattern can be programmed with the related GCM behavioural skeleton into the framework. A data parallel skeleton can be configured in order to exploit a stateful or a stateless computation. A *stateful* computation is a computation in which each worker, that represents the computational unit of the skeleton, manages an internal state that remains consistent between successive reconfigurations of the skeleton. On the other hand, a *stateless* computation is a computation in which the workers do not keep a permanent state among successive reconfigurations.

4.1 ADL description of the skeleton

The data parallel behavioural skeleton is depicted in Fig. 2.

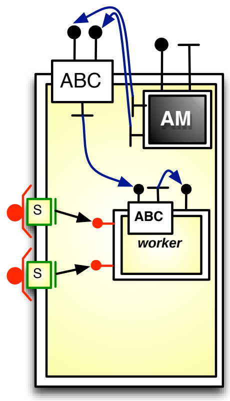

Figure 2: Stateful data-parallel *BeSke* structure.

The composite component representing the skeleton has been designed focusing on the following requirements:

- all the *provide* ports of the composite *must* be multicast ports
- the user interfaces that define the signature of each multicast port can adopt all the distribution policies provided by ProActive (`BROADCAST`, `ONE_TO_ONE`, `ROUND_ROBIN`). However, in order to define a port exploiting the pure data parallel behaviour (i.e. the input list is partitioned among the available processing elements of the skeleton), the port must be configured to adopt the `gridcomp.map.port.MapDispatch` distribution policy.
- all the multicast port belonging to the composite are managed by the manager and the autonomic controller
- it is parametric with respect three parameters provided at instantiation time:
  1. the package name of the worker ADL file
  2. the input QoS contract path
  3. the path to the deployment file

The data parallel behavioural skeleton comes in two versions, both available from the package `gridcomp.map`:

- the ADL file `mapSingle.fractal` describes a data parallel skeleton exposing just a multicast port adopting the `MapDispatch` distribution policy.
- the ADL file `mapInitQuery.fractal` describes a data parallel skeleton exposing two multicast ports both adopting the `MapDispatch` distribution policy. The first port is thought as an initialisation port, since it distributes the input data set to the available workers in a persistent manner, i.e. the partition becomes part of the worker state; the second port is thought as a port to be invoked to query in parallel each worker that will presumably access its data partition (i.e. its state) to answer.

Both versions are parametric with respect to the package name of the worker ADL file, the initial QoS contract that will be given as input to the manager and the path to the deployment file in use.

A data parallel skeleton encapsulates a manager component who is responsible of all the non-functional features of the skeleton. Thus, its influence on the composite behaviour is completely independent from the user knowledge.

4.2 How to use the data parallel skeleton

In order to instantiate a map skeleton, the user has simply to extend the `mapInitQuery.fractal` file (or the `mapSingle.fractal`, depending on his needs), in order to provide the initial number of workers encapsulated into a starting configuration of the skeleton and the bindings between the subcomponents and its internal interfaces. As an example, the following `usermap.fractal` file

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
 "classpath://org/objectweb/proactive/core/component/adl/xml/proactive.dtd">

 <definition name="gridcomp.example.statefulmap.adl.usermap"
    extends="gridcomp.map.mapInitQuery(gridcomp.example.statefulmap.adl.worker,
                    src/gridcomp/example/statefulmap/adl/rulesWithControl1.drl,
                    local-deployment.xml)">

     <!-- the initial set of workers -->
     <component name="server0"
                definition="gridcomp.example.statefulmap.adl.worker"/>
     <component name="server1"
                definition="gridcomp.example.statefulmap.adl.worker"/>

     <!-- bindings between the internal interfaces and workers interfaces -->
     <binding client="this.multicastServerItf-01"
              server="server0.serverItf-worker-01"/>

     <binding client="this.multicastServerItf-01"
              server="server1.serverItf-worker-01"/>

     <binding client="this.multicastServerItf-02"
              server="server0.serverItf-worker-02"/>

     <binding client="this.multicastServerItf-02"
              server="server1.serverItf-worker-02"/>

    <virtual-node name="map-node" cardinality="single"/>
</definition>
```

extends the two port version of the data parallel behavioural skeleton, being parametric with respect to the three arguments mentioned above that are passed to the map instantiation in the line

```
extends="gridcomp.map.mapInitQuery(gridcomp.example.statefulmap.adl.worker,
                src/gridcomp/example/statefulmap/adl/rulesWithControl1.drl,
                local-deployment.xml)"
```

In this example, the user-defined extension simply adds to the basic definition of the skeleton, two inner components that will represent the initial workers at launching time.

Of course, each worker will be a component exposing two interfaces at least, one for each internal port of the composite to which they will be bound, as the example below:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
"classpath://gridcomp/adl/gridcomp.dtd">
```

```
  <definition name="gridcomp.example.statefulmap.adl.worker"
       extends="gridcomp.map.controller.DPWorkerController"
       arguments="vn-id">

  <interface signature="gridcomp.map.controller.operation.ReconfigSupport"
       role="server"
       name="reconfig"/>

  <!-- will be bound to the init multicast port -->
  <interface signature="gridcomp.example.statefulmap.impl.ServerInitItf"
       role="server"
       name="serverItf-worker-01"/>

  <!-- will be bound to the query multicast port -->
  <interface signature="gridcomp.example.statefulmap.impl.ServerQueryItf"
       role="server"
       name="serverItf-worker-02"/>

  <content class="gridcomp.example.statefulmap.impl.ServerImpl"/>

   <!-- parametric id for the virtual node -->
   <virtual-node name="slave-node-${vn-id}" cardinality="single" />
</definition>
```

Each component representing a worker for a data parallel behavioural skeleton requires that two conditions are always satisfied:

1. the definition extends the `DPWorkerController.fractal` file: such extension enriches the component definition with a set of pre-defined controllers and interceptors providing the passive autonomicity level and all the needed hooks with the skeleton manager to interact with the worker;
2. the ADL description takes the destination virtual node as an input parameter through the argument `vn-id`, assuming that the virtual nodes are differentiated in their names by numeric suffixes (they are labelled as `slave-node-1`, `slave-node-2`, etc.).

Thus, from the user point of view, the root application can be written as follows:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
 "classpath://org/objectweb/proactive/core/component/adl/xml/proactive.dtd">

  <definition name="gridcomp.example.statefulmap.adl.testcase"
      extends="gridcomp.CompositeController">

  <interface signature="gridcomp.map.port.MapInitInterface"
             role="server"
             name="runTestItf-01"/>

  <interface signature="gridcomp.map.port.MapQueryInterface"
             role="server"
             name="runTestItf-02" />
  <!-- the map -->
  <component name="multicastComposite"
             definition="gridcomp.example.statefulmap.adl.usermap"/>
  <!-- the bindings -->
  <binding client="this.runTestItf-01"
           server="multicastComposite.multicastServerItf-01"/>
  <binding client="this.runTestItf-02"
           server="multicastComposite.multicastServerItf-02"/>

  <!-- ..... maybe other components and/or interfaces ...... -->
  <controller desc="composite"/>
  <virtual-node name="root-node" cardinality="single" />
</definition>
```

The ADL file `testcase.fractal` represents a composite extending the predefined composite named `gridcomp.CompositeController`: such extension allows the user to automatically configure a set of controllers and interceptors of GCM composite component. Moreover, the root composite encapsulates the data parallel behavioural skeleton, extended by the user through the `usermap.fractal` ADL file, that is directly bound to the two input interfaces provided by the application component as its "entry-point" (in the case of the example, the application provides the same server port provided by the extended data parallel behavioural skeleton).

4.3 How to write a parametric code

The whole application seen in the previous section, can be written as parametric with respect to the ADL worker file name, the path to the initial QoS contract, and the path to the deployment file to be used in dynamic instantiation. In other words, the ADL file can be generic enough to directly actualise these three parameters in the user Java code, instead of cabling them in the fractal files. In order to write a completely parametric code, we need very slight modifications to what we have seen above: the idea is to let the parameters "flow" among the ADL description. Thus, the `testcase.fractal` can be defined as parametric with respect to three arguments that will be passed to the `usermap.fractal` invocation. The `testcase.fractal` file changes in the following way:

```
 <definition name="gridcomp.example.statefulmap.adl.testcase"
    extends="gridcomp.CompositeController"
    arguments="worker,rulespath,padPath">

    .....

    <!-- the map -->
    <component name="multicastComposite"
        definition="gridcomp.example.statefulmap.adl.usermap(${worker},
                                                             ${rulespath},
                                                             ${padPath})"/>
   <!-- the bindings -->
   ....
</definition>
```

On the other hand, these three arguments will be passed to the map instantiation.

```
<definition name="gridcomp.example.statefulmap.adl.usermap"
     extends="gridcomp.map.mapInitQuery(${worker},${rulespath},${padPath})"
     arguments="worker,rulespath,padPath">

    <component name="server0" definition="${worker}(1)"/>
    <component name="server1" definition="${worker}(2)"/>

    <!-- bindings specifications -->
    ...
</definition>
```

Finally, the actual value of such arguments is passed at launching time to the root component as part of the context associated to its instantiation. Thus, as we have seen in 3.3, in the main of the application the user will provide three pairs to the `java.util.Map` representing the context:

```
Map context = new HashMap();
ProActiveDescriptor pad = PADeployment.getProactiveDescriptor(padPath);

context.put("deployment-descriptor", pad);
context.put("worker", "gridcomp.example.statefulmap.adl.worker");
context.put("rulespath",
            "src/gridcomp/example/statefulmap/adl/rulesWithControl1.drl");
context.put("padPath", "local-deployment.xml");

Component testcase = (Component)
                f.newComponent("gridcomp.example.statefulmap.adl.testcase",
                               context);
```

Note that the context is enriched also by the key `deployment-descriptor` to which an object of type `ProActiveDescriptor` is associated. Such pair is needed by the non-functional machinery of the data parallel skeleton to correctly deploy every new instance of a worker dynamically allocated.

4.4 Stateful data parallel skeleton

The stateful computation is naturally related to the `mapInitQuery.fractal` implementation of the data parallel skeleton, since it is thought to provide an *init* interface to partition and distribute an input data set, and a *query* interface to apply functional computation on the distributed partition that became part of the workers state. However, the same skeleton can also been used in a *stateless* computation.

As soon as a worker is added or removed from the actual configuration, a problem arises in a stateful computation: how can be the state redistributed in order to keep it consistent after the new configuration has been established? The workers are the only components functionally responsible for the partition assigned to them; thus, the workers are the only entities that can take care of the state. The mechanism offered by the data parallel skeleton to preserve the state between configuration changes requires that each worker provides a method for serializing and a method for deserialising its internal state by implementing a specific interface. In other words, since the implementation of the state is a functional concern (it's only the user who knows how the state is implemented and it is fully application-dependant), should be the user who "suggests" the framework about how to serialise and how to deserialise it, by offering to the "external" world a `java.util.List` of opaque objects. In this prospect, the role of the skeleton membrane will only be to collect a `List` of objects from the workers belonging to the old configuration and to redistribute it to the new established set of workers.

On the other hand, if the a worker does not provide methods for serialising/deserialising the state, the framework cannot make assumption on its implementation and we will talk about a *stateless* data parallel behavioural skeleton : in case of a reconfiguration, the state will not be automatically redistributed and the user will *functionally* update the data distribution among the workers, for instance by calling the *init* interface again.

4.4.1 Setting up a worker for autonomic reconfiguration

If the user wants to exploit the autonomic reconfiguration features offered by the data parallel skeleton, each worker must provide the autonomic support. In this sense, she/he simply needs to define a worker component that:

- extends the `grid.map.controller.DPWorkerController` as mentioned above;
- provides the following *server* interface

  ```
  <interface signature="gridcomp.map.controller.operation.ReconfigSupport"
             role="server"
             name="reconfig"/>
  ```

- is associated to a content class that implements, among the others, the `ReconfigSupport.java` interface. Such interface provides two methods:
  - `public void setStatus(List<Task> tsl)`: it's called by the membrane after a reconfiguration has occurred for assigning the new partitions to each worker. The partition is represented by a `java.util.List<Task>`.
  - `public List<Task> provideStatus()`: it's called by the membrane before a reconfiguration occurs in order to collect the partitions. The partition must be packed as an object of type `java.util.List<Task>`.

A `gridcomp.map.Task` object encapsulates a `String`, and `Integer` (in these cases the object is tagged as "primitive" thus the method `isPrimitive()` will return `true`) or a `List<Object>`. Its content can be retrieved through the `getObject()` method and `size()` will return 1 in case of primitive object, the size of the `List<Object>`, otherwise.

4.4.2 The reconfiguration process

The reconfiguration is a three-steps process:

1. as first step, the partitions already assigned to the current workers are collected at passive level through the invocation of the `provideStatus()` methods;
2. the reconfiguration proceeds by adding or removing a worker: the new worker will be mapped on to a virtual node that will randomly chosen among the available ones described in the deployment file in use;
3. as last step, the state collected in step 1 is partitioned on the basis of the new cardinality of workers and the partitions are distributed accordingly;

See the examples provided in [6] to have a detailed description of the output offered to the user by these three steps.

4.5 Stateless computation

If the workers provided by the user do not implement the `ReconfiSupport.java` interface, they are intended not to offer the autonomic support for guaranteeing the automatic data distribution in case of a reconfiguration.
See the examples provided in [6] to have a suggestion on how to monitoring the reconfiguration at functional level and on how to functionally manage the redistribution of the data set.

4.6 Composite workers

The package `gridcomp.example.compworker.map` offers a perspective on how a dataparallel application exploiting a composite worker can be written.

In this case, the fractal description of the worker will specify a composite component embedding one or more components, as in the example below:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
"classpath://org/objectweb/proactive/core/component/adl/xml/proactive.dtd">

<definition name="gridcomp.example.compworker.map.adl.pipe"
    extends="gridcomp.CompositeController" arguments="vn-id">

   <interface signature="gridcomp.example.compworker.map.impl.Stadio1Itf"
       role="server"
       name="inputport"/>

   <component name="stage1"
         definition="gridcomp.example.compworker.map.adl.stadio1(${vn-id})"/>
   <component name="stage2"
         definition="gridcomp.example.compworker.map.adl.stadio2(4)"/>

   <binding client="this.inputport"
            server="stage1.input"/>

   <binding client="stadio1.output"
            server="stage2.input"/>

   <virtual-node name="slave-node-${vn-id}" cardinality="single" />
</definition>
```

The composite described by `pipe.fractal` provides a server port, encapsultes two composites and specifies the related bindings. The description of `stadio1.fractal` is nothing surprising and appears as follows:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
"classpath://gridcomp/adl/gridcomp.dtd">

<definition name="gridcomp.example.compworker.map.adl.stadio1"
   extends="gridcomp.map.controller.DPWorkerController" arguments="vn-id">

   <interface signature="gridcomp.example.compworker.map.impl.Stadio1Itf"
       role="server"
       name="input"/>

   <interface signature="gridcomp.example.compworker.map.impl.Stadio2Itf"
       role="client"
       name="output"/>

   <content class="gridcomp.example.compworker.map.impl.Stadio1Impl"/>
   <virtual-node name="slave-node-${vn-id}" cardinality="single" />
</definition>
```

Indeed, `stadio2.fractal` is a simple primitive component. Both components extend the `DPWorkerController` description.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
 "classpath://gridcomp/adl/gridcomp.dtd">

<definition name="gridcomp.example.compworker.map.adl.stadio2"
    extends="gridcomp.map.controller.DPWorkerController" arguments="vn-id">

    <interface signature="gridcomp.example.compworker.map.impl.Stadio2Itf"
         role="server"
         name="input"/>

    <content class="gridcomp.example.compworker.map.impl.Stadio2Impl"/>
    <virtual-node name="slave-node-${vn-id}" cardinality="single" />
</definition>
```

Note that, since the worker description can be parametric with respect to just a parameter, if a user needs to assign different virtual nodes to each component involved in the description above (components `pipe`, `stage1` and `stage2`), in the current version of the framework he will need to cable the virtual nodes in the description, or to decide to which component the argument is related. In the example above, the user decided that the parameters refers to the composite and to the first stage of the pipeline (they will located in pair on the same virtual node), while the second will be loaded on a different virtual node.

Limitations in the use of composite workers In the current version of the data parallel behavioral skeleton, in case of composite workers only the stateless computation is supported. This limitation arises from the fact that, once the mechanism illustrated in Section 4.4.1 has been consolidated, keeping the state between one configuration and another would require a coordinating functionality offered by the autonomic controller of the composite worker. Developing such feature belongs to the set of goals related to the next refinements of the skeleton.

4.7 Observable events in the map behavioral skeleton

Currently, there are two events that have been designed to specifically observe the map behaviour, each represented by a given Java bean:

- the size of a partition assigned to each worker of a map: the bean is implemented by class `gridcomp.manager.map.impl.PartitionSizeBean` and returns the size of the partition assigned to each worker by the `MapDispatch` distribution policy;
- the cardinality of workers in a map: it returns the current number of workers and the bean is implemented by the class `gridcomp.manager.map.impl.NumWorkerBean`.

Moreover, in `gridcomp.manager.beans.CounterBean` a bean functionally equivalent to a counter has been developed . This bean is particularly useful to count the number of evaluation steps or to introduce idle evaluation cycles while the monitoring system collects more reliable performance information.

A counter rule could appear as follows:

```
rule "Count"
  salience -10
  when
    // include some other bean evaluations, if needed
    $count   : CounterBean( value < 10 )
  then
    // take some action, if needed...
    $count.incValue();
end

rule "CountReset"
  salience -20
  when
    // include some other bean evaluations, if needed
    $count   : CounterBean( value >= 20)
  then
    $count.reset();
end
```

The rule `Count` has a minimal salience and if its condition holds, the value of the count is increased. In the `CountReseat` rule, the value of the count is reinitialised.

The example contract below shows how this bean can be used to module the evaluation of the quality of service. Let us assume to have the following rule in our contract:

```
rule "CheckHigherBound"
  when
    $partBean : PartitionSizeBean(value >= Manager2Constants.DP_MAX_PART_SIZE)
    $count : CounterBean( value > Manager2Constants.DP_SKIP_TURN_COUNTER )
  then
    System.out.println( "Inadequate partition size ="+$partBean.getValue());
    $partBean.setData(new Integer(Manager2Constants.DP_ADD_WORKERS));
    $partBean.fireOperation(ManagerOperation.ADD_EXECUTOR);
    $count.reset( );
 end
```

The effects of this contract (that use variables defined in class `Manager2Constants`) is to keep idle until at least `DP_SKIP_TURN_COUNTER` evaluation cycles has been done. After that, `CheckHigherBound` is activated if the partition of each worker is greater than `DP_MAX_PART_SIZE` and the effect is

- to set the number of workers to be added (represented by the `DP_ADD_WORKERS` constant) as value for `Data` field in the `PartitionSizeBean`.
- to fire the `ADD_EXECUTOR` operation
- to reset the counter
- to update the value of the counter

From this point ahead, at least other `DP_SKIP_TURN_COUNTER` idle cycles will be required before another firing of `CheckHigherBound` gets possible.

4.8 Guidelines for using the stateful data parallel skeleton

The steps a user must consider in order to successfully use the stateful data parallel skeleton can be summarised as follows:

1. provide the root component description as an extension of the `gridcomp.CompositeController` (and any other sub-component he/she wants to use);
2. include in his application a component extending the adequate version of the data parallel skeleton among those available in `gridcomp.map`. The extended component should contain (if needed) the definition of the workers included in the initial configuration of the skeleton;
3. provide the worker component description by extending `DPWorkerController` in the package named `gridcomp.map.controller`, and by implemeting the `ReconfigSupport` interface in order to instruct the manager about how serialise/de-serialise the worker state. Define the virtual node identifier on which the worker will be assigned to as an argument (i.e. a parameter) of the fractal description file;
4. populate the `java.util.HashMap` context of the root component with a `ProActiveDescriptor` instance of the related to the deployment file in use.
5. provide the needed parameters (the ADL file of the worker, the path to the QoS contract file and the path to the deployment file) to the skeleton at description or launching time;

4.9 Guidelines for using the stateless data parallel skeleton

The steps a user must consider in order to successfully use the stateless data parallel skeleton can be summarised as follows:

1. provide the root component description as an extension of the `gridcomp.CompositeController` (and any other sub-component he/she wants to use);
2. provide the worker component description as an extension of the `gridcomp.PrimitiveController` or the `gridcomp.map.controller.DPWorkerController`; define the virtual node identifier on which the worker will be assigned to as an argument (i.e. a parameter) of the fractal description file;
3. include in his application a component extending the adequate version of the data parallel skeleton among those available in `gridcomp.map`. The extended component should contain the definition of the workers included in the initial configuration of the skeleton;
4. treating at functional level the state updates of the workers in case of reconfiguration (if needed);
5. populate the `java.util.HashMap` context of the root component with an instance of the ProActive class `ProActiveDescriptor` related to the deployment file in use.
6. provide the needed parameters (the ADL file of the worker, the path to the QoS contract file and the path to the deployment file) to the skeleton at description or running time;

5 The Farm behavioural skeleton

In this tutorial we will detail the progress achieved for the active level of autonomicity; for a detailed discussion about the passive level we refer back to [6].

We recall that a behavioural skeleton exploits an active level of autonomicity if the non-functional strategies related to the reconfiguration of the composite are taken by a manager belonging to the skeleton itself. The decision taken by the manager can be driven accordingly with the user needs thorough a QoS contract.

5.1 ADL description of the skeleton

The farm behavioural skeleton is depicted in Fig. 3

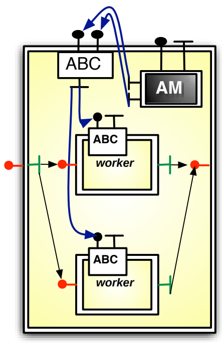

Figure 3: The farm behavioural skeleton logical organisation

The composite component `gridcomp.manager.farm.adl.farm` representing the skeleton has been designed focusing on the following requirements:

- all the *provide* ports must be server ports
- the skeleton description is parametric with respect to three parameters provided at instantiation time:
  1. the input QoS contract file path
  2. the package name of the worker ADL file
  3. the path to the deployment file
- a properties file has to be associated to each skeleton of this type (see next section for further details).

5.2 How to use the farm skeleton

In order to instantiate a farm skeleton, the user has simply to extend the `farm.fractal` file. The extended component will provide the functional interfaces of the skeleton and the initial number of workers encapsulated by the starting configuration of the skeleton. As an example, the following `userfarm.fractal` file offers the description of a farm including just one worker and exposing two interfaces with different roles.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
  "classpath://gridcomp/adl/gridcomp.dtd">

  <definition name="gridcomp.example.active.pi.adl.userfarm"
              extends="gridcomp.manager.farm.adl.farm(${contractFile},
                                                      ${worker},
                                                      ${padPath})"
              arguments="contractFile,worker,padPath">
  <interface name="work"
             role = "server"
             signature = "gridcomp.example.active.pi.Worker" />

  <interface name="send"
             role = "client"
             signature = "gridcomp.example.active.pi.Collector" />

  <component name="worker"
             definition = "${worker}"/>


  <binding client="this.work"   server = "worker.work"/>
  <binding client="worker.send" server = "this.send"/>

  <virtual-node name="master-node" cardinality="single"/>

  <beske xmlFile="src/gridcomp/example/active/pi/farm.properties"/>
</definition>
```

The description extends the `farm.fractal` file passing the three arguments mentioned above. In particular, the worker have to extend the `gridcomp.PrimitiveController` file and offer the functional interfaces to the external world:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
"classpath://gridcomp/adl/gridcomp.dtd">

  <definition name="gridcomp.example.active.pi.adl.worker"
              extends="gridcomp.PrimitiveController">

      <interface signature="gridcomp.example.active.pi.Worker"
             role="server"
             name="work"/>

      <interface signature="gridcomp.example.active.pi.Collector"
             role="client" name="send"/>

      <content class="gridcomp.example.active.pi.WorkerImpl"/>
      <virtual-node name="master-node" cardinality="single"/>
</definition>
```

Since this version of the farm skeleton does not provide a way to parametrically express the virtual node on which the worker will be assigned to, the user is asked for providing a worker ADL description for each virtual node involved in the deployment.[2] In order to facilitate this step, the description of a user extended farm *must* always include the tag

```
<beske xmlFile="path to a properties file"/>
```

where the properties file has the following content:

```
adl.dir  = gridcomp/example/active/pi/adl/slaves
xml.file = current-deployment.xml
```

The first property, called *slaves directory*, contains the path to a directory in which the user collected the ADL descriptions of the set of workers the application could instantiate. The second property contains the name of the deployment file in use. The role of this tag is to pass these two arguments to the autonomic controllers that will drive the reconfiguration of the skeleton at running time .
The `slave` directory contains a set of fractal files like the following one:

[2]In the future, this solution will be replaced with the more light way of passing the virtual node identifiers as arguments to the worker description, already used for the dataparallel skeleton.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADLVbc 2.0//EN"
"classpath://org/objectweb/proactive/core/component/adl/xml/proactive.dtd">

<definition name="gridcomp.example.active.pi.adl.slaves.slave-1"
            extends="gridcomp.example.active.pi.adl.worker">

<virtual-node name="slave-node-1" cardinality="single"/>
</definition>
```

De facto, since the farm represents a functional replication, each worker definition is given as a derivation of the worker/ description given as input to the skeleton but specializing the destination virtual node.

Finally, the root application can be written as follows:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
 "classpath://gridcomp/adl/gridcomp.dtd">

<definition name="gridcomp.example.active.pi.adl.root"
    extends="gridcomp.CompositeController"
    arguments="contractFile">

   <interface signature="gridcomp.example.active.pi.Master"
              role="server"
              name="do"/>

   <!-- a stream generator -->
   <component name="client"
              definition="gridcomp.example.active.pi.adl.master"/>

   <!-- the farm -->
   <component name="farm"
    definition="gridcomp.example.active.pi.adl.userfarm(${contractFile},
                               gridcomp.example.active.pi.adl.worker,
                               current-deployment.xml)"/>

   <binding client="this.do"     server="client.do"/>
   <binding client="client.work" server="farm.work"/>
   <binding client="farm.send"   server="client.send"/>
   <virtual-node name="master-node" cardinality="single"/>
</definition>
```

The ADL file `root.fractal` represents a composite that extends `gridcomp.CompositeController` in order to allow the user to automatically include a set of controllers and interceptors of GCM composite components. Moreover, the composite encapsulates a generator for the stream, and the farm behavioural skeleton extended by the user through the `userfarm.fractal` ADL file.

Notice that, as in the map case, both the `userfarm.fractal` and the `root.fractal` descriptions can be written in a fully parametric way with respect to the arguments needed by the farm skeleton (as shown Sec. 4.3).

5.3 Composite workers

The package `gridcomp.example.compworker.farm` offers an example of how to write a version of the Mandelbrot application (the full code is provided among the examples in the `GridComp-Examples` folder) in which the worker is a composite component. In the example, the worker is a two stages pipeline in which the first stage redirect the method calls to the second stage (do nothing, actually) that really evaluates the Mandelbrot set.

As shown in Sec. 4.6, using a composite component to instantiate farm behavioural skeleton is methodologically identical of using a primitive component. Moreover, by exploiting a parametric ADL file for the composite worker is also possible to avoid the replication of ADLs files of the inner components. As matter of a fact, the only additional complexity due to the usage of composite worker come from the growth of virtual nodes in the deployment file.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
"classpath://org/objectweb/proactive/core/component/adl/xml/proactive.dtd">

<definition name="gridcomp.example.compworker.farm.adl.workerpipe"
    extends="gridcomp.CompositeController" arguments="vn1,vn2">

    <interface signature="gridcomp.example.compworker.farm.impl.LineServer"
         role="server"
         name="lineserver-server"/>

    <interface signature="gridcomp.example.compworker.farm.impl.LineCollector"
         role="client"
         contingency="optional"
         name="collector-client"/>

    <component name="first"
 definition="gridcomp.example.compworker.farm.adl.dummyworker(${vn1})"/>

    <component name="second"
         definition="gridcomp.example.compworker.farm.adl.lineserver(${vn2})"/>

   <binding client="this.lineserver-server"
    server="first.start"/>

   <binding client="primo.input"
    server="second.lineserver-server"/>

    <binding server="this.collector-client"
    client="second.collector-client"/>

<virtual-node name="output-node" cardinality="single" />
</definition>
```

The `workerpipe.fractal` file describing the worker is a regular composite embedding two components and providing the bindings between them and between the internal interfaces. Moreover, the package `gridcomp.example.compworker.farm.slave` will collect the descriptions of the workers dynamically instantiated and extending the definition above by just overriding the virtual-node association.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
"classpath://org/objectweb/proactive/core/component/adl/xml/proactive.dtd">

<definition name="gridcomp.example.compworker.farm.slave.slave-1"
   extends="gridcomp.example.compworker.farm.adl.workerpipe(1,2)">

   <virtual-node name="slave-node-1" cardinality="single" />
</definition>
```

Note that extending the composite description to define a new component worker also allow to differentiate the virtual nodes that will be assigned to the encapsulated components, because their identifiers can be passed as arguments to the `workerpipe.fractal` description. In fact, according to the extended description of the composite, the virtual node with identifier `1` will be assigned to component `first`, while the virtual node with identifier `2` will be assigned to component `second`.

Finally, the description of the two primitive components encapsulated by the composite worker is not different from any other primitive description: the ADL file extends the `gridcomp.PrimitiveController`, it's parametric with respect to the virtual node hosting the component and esposes the functional interfaces needed to provide the computation.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
"classpath://gridcomp/adl/gridcomp.dtd">

<definition name="gridcomp.example.compworker.farm.adl.lineserver"
   extends="gridcomp.PrimitiveController" arguments="vn2">

   <interface signature="gridcomp.example.compworker.farm.impl.LineCollector"
              role="client"
              contingency="optional" name="collector-client"/>

   <interface signature="gridcomp.example.compworker.farm.impl.LineServer"
              role="server" name="lineserver-server"/>

   <content class="gridcomp.example.compworker.farm.impl.LineServerImpl"/>
   <virtual-node name="slave-node-${vn2}" cardinality="single"/>
</definition>
```

Limitations in the use of composite workers While instancing a composite worker for implementing the farm, the user must consider that the `ManagerOperation.BALANCE_LOAD` is not supported and cannot be used in the definition of the QoS contract's actions. In fact, this operation is based on moving the tasks enqueued in the workers' priority queues in order to assign ready jobs to the instanced worker. However, all the tasks passing through the composite worker interfaces move directly to the inner components' queues, making useless the rebalancing action on the composite queue.

5.4 Observable events in the farm behavioral skeleton

Currently, there are three events that have been designed to specifically observe the farm behaviour, each represented by a given Java bean:

- the arrival rate of the tasks belonging to an input stream, being the bean implemented by the class `gridcomp.manager.beans.ArrivalRateBean`;
- the departure rate of the tasks belonging to an input stream, being the bean implemented by class `gridcomp.manager.beans.DepartureRateBean`;
- the cardinality of workers in a farm: the bean returns the current number of workers and is implemented by the class `gridcomp.manager.beans.NumWorkerBean`.

In the description of a QoS contract related to a farm-based application, a counter bean as the one described in Section 4.7 could also be used in order to introduce idle cycles in the evaluation time-line.

5.5 Guidelines for using the farm skeleton at active level

The steps a user must consider in order to successfully use the farm skeleton are the following ones:

1. provide the root component description as an extension of the `gridcomp.CompositeController`
2. include a component extending `farm.fractal`; the extended component should contain the definition of the workers included in the initial configuration of the skeleton, the `beske` XML tag specifying a path to a properties file and any other functional interface;
3. provide a properties file in which the slaves directory path and the name of the deployment file are specified;
4. provide a description of the worker and a set of its extensions (to be collected in the slave directory), one for each virtual node on which a worker could be loaded at running-time
5. provide the needed parameters (the ADL file of the worker, the path to the QoS contract file and the path to the deployment file) to the skeleton at instancing (i.e. through constants) or at running time (i.e. through ADL arguments).

6 Multilevel management

One of the major improvements introduced with the current version of our behavioral skeletons is the possibility to configure a multilevel management between different managers in the same application. Let us assume to have an application like the one depicted in Fig. 4, in which a data parallel skeleton as the one showed in 4 is initialized and queried by a `Producer` component that is in charge of defining the initial data set and to push the queries to the skeleton. We could imagine a scenario in which, as

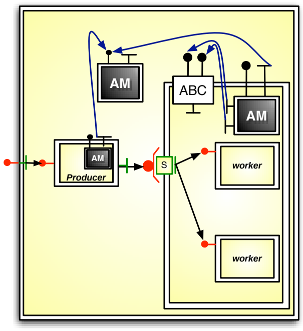

Figure 4: Managers interacting at 2 level of hierarchy

soon as a reconfiguration occurs in the skeleton, the `Producer` corrects its behavior by, for instance, modifying the size of the data set given as input to the skeleton or modifying the arrival time between two successive queries. In order to implement this scenario and, generally speaking, a situation in which sibling components can coordinate each other in order to realize a coherent behavior, we need a two-level management. In this hierarchy, each sibling is responsible for its internal behavior (its not-functional concerns) by means of its manager or it's autonomic controllers, depending on if it exposes a passive or an active autonomicity. The interaction between the sibling components, instead need to be coordinated by a super-manager that monitors not-functional behaviors of the controlled components, plans a behavioral strategy, and execute it by suggesting the siblings how to modify their behavior in order to implement a coherent modification. In order to provide these features to GCM components and, in particular, to *BeSke* component, we have extended the set of non-functional interfaces to a set delegated to treat the inter-relation among managers.

6.1 Not functional interfaces

According to the picture in Fig. 4, the root component embodies a component behaving as a "root manager": this component extends `gridcomp.manager.GenericAutonomicManager` and exposes a set of non-functional interface for interacting with the other components, in particular:

- `commit-contract`: it's a *use* port through which the manager submit a new instance of a QoS contract from an lower level component;
- `commit-value`: it's a *use* port through which the manager submit a new set of requirements (i.e. pairs `<variable, value>`) to a lower level component, so that the controlled component can update its contract;
- `commit-to-all`: it's a *use* port similar to the `commit-contract` but it exposes a multicast cardinality. It's used when for the coordination of two or more controlled managers as a whole
- `raise-violation`: it's a *provide* port through which a root manager receives violation signals from lower level components.

As a result, the protocol activated while a interaction manager-manager occurs is the following:

- a low level manager can signal to the root manager that one or more conditions in its contract have been broken through the `raise violation` port. Note that the signalling phase also occurs if the manager itself is not able to plan a valuable strategy with respect to the QoS goals to be reached.

- the root manager submit a new contract or a new (set of) value to the controlled manager through the `commit-contract`, the `commit-value` and the `commit-to-all`, respectively.

A root manager can push a new contract on the basis of

- the signaling of a violation raised at lower level
- an explicit request pushed by the user
- a planning strategy due to the evaluation of monitored conditions related to the environment and/or QoS information provided by the other components acting in the computation.

6.2 Defining and implementing a top-manager

The application in Fig. 4 has been implemented in `gridcomp.example.stateful.twolevels.*`. The example does not represent a real application since its the goal is to show the potential features of the multilevel mechanism and how to use them.
Before detailing the whole application, we have to point out that we will need to define a manager for the root component, the so called *top-manager*. In this case, it will not be a brand new component, exposing specific interfaces, that the user will implement in order to specify which particular management the application will exploit at higher level of autonomic hierarchy. In the proposed example, the top-manager is defined by the following ADL file:

```
?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
"classpath://org/objectweb/proactive/core/component/adl/xml/proactive.dtd">

<definition name="gridcomp.example.stateful.twolevels.adl.my-top-manager"
    extends="gridcomp.manager.GenericAutonomicManager"
    arguments="contractFile-top,numServed">

    <interface signature = "gridcomp.AutonomicServerManager"
        role = "client"
        name = "commit-contract"/>
    <interface signature = "gridcomp.AutonomicServerManager"
        role = "client"
        name = "commit-value"/>

    <interface signature = "gridcomp.AutonomicMulticastServerManager"
        role = "client"
        name = "commit-to-all"
        cardinality = "multicast"/>

    <interface signature = "gridcomp.AutonomicClientManager"
        role = "server"
        name="raise-violation"/>

    <content class="gridcomp.example.stateful.twolevels.impl.TopManager"/>
    <attributes
          signature="gridcomp.manager.AutonomicManagerAttributeController">
        <attribute name="PerformanceContractFile" value="${contractFile-top}"/>
        <attribute name="LoggerFileName" value="/tmp/top-status.log"/>
        <attribute name="AutonomicLoopCycle" value="4000" />
    </attributes>
</definition>
```

The top-manager exposes the four not-functional interfaces related to the inter-manager communication system and a set of attributes needed to set some important values at start-up time:

1. `PerformanceContractFile`, the path to the QoS contract file on which the top-manager will take decisions and that will passed to it as *first* contract at launching time
2. `LoggerFileName`, the path to a logger file name to track the manager output

3. AutonomicLoopCycle, a numerical value to express the time in milliseconds lasting between each activation of the manager (i.e. the evaluation of the QoS contract) and the successive.

The ADL description is parametric with respect the path to the QoS file.
From the implementation point of view, once the user has extended GenericAutonomicManager class, the top-manager is written as an implementation of the org.objectweb.proactive.RunActive interface. Since the implementation of the methods related to the not-functional interfaces and to the set of attributes is already provided by the class GenericAutonomicManager, the user simply need to implement the method needed to read the QoS contract, as it can be seen in the following Java implementation class:

```
public class TopManager
   extends GenericAutonomicManager
   implements RunActive//, TopManagerAttribute{

private final org.apache.log4j.Logger logger =
             org.apache.log4j.Logger.getLogger(gridcomp.log.Loggers.MANAGER);
private HashMap<String,Double> methodsToMonitor =
             new HashMap<String, Double>();

    public TopManager(){
      super("/tmp/monitorfile");
    }

    protected String readContractFile(String fName) {
       File f = new File(fName);
       if (f.exists()){
           // Reading contract from file fName
           logger.info("Reading contract from file " + fName);
           RulesParser parser = new RulesParser();
           logger.debug("Parsing Contract");
           String parsed = parser.parse(f.getAbsolutePath());
           logger.debug("Parsing Contract "+f.getAbsolutePath());
           if ((parsed!= null) && (parsed.length() > 0)){
               ArrayList<String> methodNames = parser.getMethodsNames();
               for (String method: methodNames){
                  String tmp = method.replace("\"", "");
                  methodsToMonitor.put(tmp, 0.0);
               }
               logger.debug("Committing Contract");
               return(parsed);
            }
        }else {
           logger.warn("File " +fName + " doesn't exist!");
        }
        return (null);
    }
}
```

The constructor requires to setup a file in which the monitoring data will be written during the execution. The only method implementation required is related to readContractFile that will initialize the set of method to monitor and set the file to be evaluated as QoS contract file.

6.3 Controlled components

The root component of the application is described by the following ADL file:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
```

```
"classpath://org/objectweb/proactive/core/component/adl/xml/proactive.dtd">
<definition name="gridcomp.example.stateful.twolevels.adl.testcase"
    extends="gridcomp.CompositeController"
    arguments="worker,rulespath,rulespath-top,padPath">

    <interface signature="gridcomp.example.stateful.twolevels.impl.Producer"
        role="server"
        name="do" />

    <component name="producer"
        definition="gridcomp.example.stateful.twolevels.adl.producer"/>

    <component name="map"
        definition="gridcomp.example.stateful.twolevels.adl.mymap(${worker},
                                                          ${rulespath},
                                                          ${padPath})"/>

     <!-- the top manager! -->
     <component name="top-manager"
     definition="gridcomp.example.stateful.twolevels.adl.my-top-manager(${rulespath-top})"/>

     <binding client="this.do" server="producer.do"/>
     <binding client="producer.init" server="map.multicastServerItf-01"/>
     <binding client="producer.query" server="map.multicastServerItf-02"/>
     <binding client = "map.raise-violation" server = "top-manager.raise-violation"/>
     <binding client="top-manager.commit-contract" server="map.commit-contract"/>
     <binding client="top-manager.commit-value" server="producer.commit-value"/>
     <binding client="top-manager.commit-to-all" server="map.recv-mult"/>
     <binding client="top-manager.commit-to-all" server="producer.recv-mult"/>
     <controller desc="composite"/>
</definition>
```

It defines three inner component: the `producer` (a primitive component), the `map` (i.e. the user extension of the map behavioral skeleton) and the top-manager, the one detailed above.
Before entering the details related to the bindings among them, let us take a look the the `producer` definition, as specified below:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
"classpath://org/objectweb/proactive/core/component/adl/xml/proactive.dtd">

<definition name="gridcomp.example.stateful.twolevels.adl.producer"
    extends="gridcomp.PrimitiveController">

    <interface signature="gridcomp.example.stateful.twolevels.impl.Producer"
         role="server"
         name="do"/>

    <interface signature="gridcomp.map.port.MapInitInterface"
         role="client"
         name="init"/>

    <interface signature="gridcomp.map.port.MapQueryInterface"
         role="client"
         name="query"/>

    <interface signature = "gridcomp.AutonomicServerManager"
         role = "server"
         name = "commit-value"/>
    <interface signature = "gridcomp.AutonomicMulticastClientManager"
         role = "server"
         name = "recv-mult"/>
    <content class="gridcomp.example.stateful.twolevels.impl.ProducerImpl"/>
</definition>
```

The producer exposes three functional interfaces and two not-functional interfaces. Interface `do` represents the "entry-point" to the application and is the one called by the root (through an internal binding) to start the application. Interfaces `init` and `query` are the two interfaces that will be bound to the `gridcomp.map.mapInitQuery` version of the data-parallel behavioral skeleton.
The two not-functional interfaces, instead, are needed to bound the `producer` component to the top-manager. In our case it offers two interfaces:

- `commit-value`, that will be provided by the producer component to receive quality of service values from the top-manager in a one-to-one communication channel.
- `recv-mult`, that will be provided by the producer component to receive quality of service values from the top-manager in a one-to-many communication channel.

From an implementation point of view, the producer will need to implement `org.objectweb.proactive.RunActive`; moreover, `AutonomicServerManager` and `AutonomicMulticastClientManager` need to be implemented in order to define the behavior of the producer after a new QoS value as been committed through its interface. For instance, in the example shown below, the functional behavior of the `producer` is implemented in the method `start()`: for each life cycle, the component creates a `List` of `SIZE` elements to query the map.

```
 public class ProducerImpl
    implements Producer, RunActive, AutonomicServerManager,
    AutonomicMulticastClientManager{

   private int SIZE;      // the size of the input dataset
   MapQueryInterface map; // the reference to the query port of the Map BeSke

   public ProducerImpl( int n ){
      this.SIZE = n;
   }

   public void start(){
      // this method could represent the entry point of the application
      ....
      List inputlist = ...;
      while (true){
        for (int i=0;i<SIZE;i++)
           inputlist.add(...);

        List t = map.searchMatch(inputlist);
     }
      ....
   }

    // this method is required by the AutonomicServerManager interface
    // it's invoked by the top-manager to signal a context change
    public GenericTypeWrapper<Object> commitValue(GenericTypeWrapper<Object> qosValue) {
          // retrieve the value coming from the top-manager
          int val = ((Integer)qosValue.getObject()).intValue();
          if (val>0)
             this.SIZE = this.SIZE * 2;
          else
             this.SIZE = this.SIZE/2;
  return  new GenericTypeWrapper(null)
}
```

A not-functional behavior reacting to the context change could be the one represented by the implementation of the `commitValue` method. Such method is invoked by the top-manager to signal that some variables in the context have been changed. In the example above, depending on the sign of the committed value, the producer will decide if double or half the size of the lists provided as input to the Map *BeSke* at each cycle.

6.4 Father-child bindings

Once we have discussed the interfaces of the root component, the top-manager component, the producer and the map (see Section 4), let us understand how these four entities can be bound in order to have an effective not-functional communication system. The bindings have to be explicated in the root component ADL and is detailed as follows:

```
<!-- ---- bindings between functional interfaces ---- -->
   <binding client="this.do" server="producer.do"/>
   <binding client="producer.init"
            server="map.multicastServerItf-01"/>
   <binding client="producer.query"
            server="map.multicastServerItf-02"/>

<!-- ---- bindings between NOT-functional interfaces ---- -->

 <!-- both components are bound to recv-mult -->
   <binding client="top-manager.commit-to-all"
            server="map.recv-mult"/>
   <binding client="top-manager.commit-to-all"
            server="producer.recv-mult"/>

 <!-- --- the map is bound to raise-violation --- -->
   <binding client = "map.raise-violation"
            server = "top-manager.raise-violation"/>
 <!-- --- map bound to the manager through commit-contract --- -->
   <binding client="top-manager.commit-contract"
            server="map.commit-contract"/>
 <!-- --- map bound to the manager through commit-value --- -->
   <binding client="top-manager.commit-value"
            server="producer.commit-value"/>
```

As it can be seen by the example above, the bindings between the top-manager and the other components could be of different type: the `map` component is bound to the top-manager through a the `recv-mult`, the `raise-violation` and the `commit-contract` port, while the `producer` component is bound through the `recv-mult` and the `commit-value` port. This is due because

In fact, since the not-functional relation between top-manager and controlled-component depends on what they need to communicate each other, all the not-functional port are optional in the top-manager set of interface and each one could be included or not, taking into account the particular pair involved in the binding. In our specific example, the programmed behavior requires that

1. in case of a reconfiguration, the map component proceeds to the adding/removing a component and to signal the change to the top-manager through the `raise-violation` port
2. as soon as the top-manager has received a violation signal from the map component, depending on the action taken, the top-manager could decide to send a not-functional message to both the sub-components through the `recv-mult` port or to just the `producer` component through the `commit-value` port (maybe a code identifying the type of violation raised elsewhere in the component tree).

6.5 Driving the top-manager behavior

As specified by the ADL file describing the top-manager, this entity needs to be provided with a QoS contract. The role of the contract is to detect the behavior of the controlled components and, eventually, to take some decisions (action) in order to keep the performance in an acceptable range. The contract showed below is the one implementing the protocol required to implement the application in Fig. 4.

```
package gridcomp.example.stateful.twolevels.impl;
import gridcomp.manager.beans.*;
import gridcomp.manager.operations.*;
import gridcomp.operation.*;
import gridcomp.manager.util.MessageList;
/*
* QoS contract to be submitted to the Top-Manager
*
*/
[methodMonitor="runActivity"]
rule "detectViolation"
     when
          $exception : ViolationIdentityBean( value < 0 )
     then
          System.out.println("Top-Manager contract: Violation n. "+
                 $exception.getValue()+" from "+$exception.getIdentity());
          $exception.setData(new Integer(-600));
          $exception.fireOperation(ManagerOperation.PUSH_OBJECT);

          $exception.setData((Object)
             new MessageList(
               new ViolationMessage[]{
                  new ViolationMessage($exception.getValue(),$exception.getIdentity()),
                  new ViolationMessage($exception.getValue(),$exception.getIdentity())
               })
           );

          $exception.fireOperation(ManagerOperation.PUSH_LIST);
          $exception.setValue(new Integer(0));
end
```

The monitoring activity is based on the values provided by

`gridcomp.manager.beans.ViolationIdentityBean`

which carries an Integer value representing an error code and a String representing a symbolic name for the original sender of the violation. Once the bean returns a not-null violation code, it sends to the `producer` component a message (an Integer value). The message is sent through the `commit-value` port and the operation, which belongs to the set of pre-defined operation included in the `ManagerOperation` class is called `PUSH_OBJECT`. After that, the Manager could decide to send a message to both the components through its `recv-mult` port that is a multicast client port. Thus, it needs to instantiate a `List` of messages (implemented by `gridcomp.manager.util.MessageList`, an extension of `ArrayList`) and to fire the `PUSH_LIST` operation. At the end, the bean value becomes 0 again.

Let us see how this protocol is activated by the map QoS contract. This contract could be written as follows:

```
package gridcomp.example.stateful.twolevels.impl;
import gridcomp.manager.beans.*;
import gridcomp.manager.operations.*;
import gridcomp.operation.*;
import gridcomp.manager.map.impl.PartitionSizeBean;
import gridcomp.manager.map.impl.NumWorkerBean;
/*
* QoS contract to be submitted to the Map Beske
*
*/
[methodMonitor="eval"]
```

```
rule "CheckHigherBound"
        when
            $arrivalBean : PartitionSizeBean(value > 18 && value < 24)
        then
            $arrivalBean.fireOperation(ManagerOperation.ADD_EXECUTOR);
            $arrivalBean.setData(new ViolationMessage(-100,"map-message"));
            $arrivalBean.fireOperation(ManagerOperation.RAISE_VIOLATION);
            retract($arrivalBean);
 end
```

The decision taken by the manager is based on the partition size returned by the `PartitionSizeBean`: if the size belongs to a given range, the first operation fired by the manager is an `ADD_EXECUTOR` to increase by one the worker cardinality; then a `ViolationMessage` containing an error code and a symbolic identity for the sender component is instantiated and provided to the bean as its data; at the end, a `RAISE_VIOLATION` operation is fired, which means that the instantiation violation message will be sent to the top-manager through the `raise-violation` client port.

6.6 Another example

The package `gridcomp.example.active.twolevels.*` offers an example of multilevel management in the context of a farm *BeSke*. The application is depicted in Fig. 4 and, with respect to the previous example, the top-manager controls three components, two primitives and one farm.
The root application is represented by the following `root.fractal` ADL file:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
"classpath://gridcomp/adl/gridcomp.dtd">

<definition name = "gridcomp.example.active.twolevels.adl.root"
            extends = "gridcomp.CompositeController"
            arguments="contractFile-farm,worker,padpath">

   <!-- ----------- the inner components ------------------ -->
   <component name="input"
             definition = "gridcomp.example.active.twolevels.adl.input"/>
   <component name="farm"
    definition = "gridcomp.example.active.twolevels.adl.userfarm(${contractFile-farm},
                                                                 ${worker},
                                                                 ${padpath})" />
   <component name="output"
             definition = "gridcomp.example.active.twolevels.adl.output"/>
   <!-- ---------------- the top-manager ------------------- -->
   <component name="top-manager"
             definition = "gridcomp.example.active.twolevels.adl.top-manager"/>

   <!-- ---------------- the bindings ------------------- -->
   <binding client = "farm.collector-client"   server = "output.collector-server"/>
   <binding client = "input.lineserver-client" server = "farm.lineserver-server"/>
   <binding client = "top-manager.commit-contract" server = "farm.commit-contract"/>
   <binding client = "farm.raise-violation" server = "top-manager.raise-violation"/>
   <binding client = "top-manager.commit-value" server = "input.set-rate-server"/>

   <virtual-node name="root-node" cardinality="single"/>
</definition>
```

while the top-manager is described as follows:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
```

```
"classpath://gridcomp/adl/gridcomp.dtd">

<definition name = "gridcomp.example.active.twolevels.adl.top-manager"
            extends = "gridcomp.manager.GenericAutonomicManager"
            arguments="contractFile-top">
   <interface signature = "gridcomp.AutonomicServerManager"
              role = "client"
              name = "commit-contract"/>
   <interface signature = "gridcomp.AutonomicClientManager"
              role = "server" name="raise-violation"/>
   <interface signature = "gridcomp.AutonomicServerManager"
              role = "client" name = "commit-value"/>
   <content class="gridcomp.example.active.twolevels.TopManager"/>

   <attributes signature="gridcomp.manager.AutonomicManagerAttributeController">
        <attribute name="PerformanceContractFile" value="${contractFile-top}"/>
        <attribute name="LoggerFileName" value="/tmp/top-status.log"/>
        <attribute name="AutonomicLoopCycle" value="4000" />
   </attributes>

   <virtual-node name="top-manager-node" cardinality="single"/>

</definition>
```

The top-manager exposes a `raise-violation` server port and two client ports, `commit-value` and `commit-contract`, whose role is already been explicated in Section 6.1. These ports are bound to the `input` and the farm components; in particular, the `raise-violation` is bound to the `raise-violation` port belonging to the *BeSke* component since it represents the channel through which the (manager of the) farm signals to the top-manager two possible type of events:

- `notEnoughTasks_VIOL`: the input rate is too low and the *BeSke* is not more able to satisfy its internal QoS contract
- `tooMuchTasks_VIOL`: the input rate is too high and the *BeSke* is not more able to satisfy its internal QoS contract

The farm signals these events by means of specific rules in its contract. As instance, the following one is the rule through which the first input rate violation is signalled to the top-manager (look at `rule.drl` to see the whole contract):

```
rule "CheckInterArrivalRateLow"
    salience 5
     when
         $arrivalBean : ArrivalRateBean( value <  ManagersConstants.FARM_LOW_PERF_LEVEL)
         $count : CounterBean( value > ManagersConstants.FARM_SKIP_TURN_COUNTER )
     then
         $arrivalBean.setData(ManagersConstants.notEnoughTasks_VIOL);
         $arrivalBean.fireOperation(ManagerOperation.RAISE_VIOLATION);
end
```

Adaptive behavior The not-functional behavior of the top-manager is represented by its contract, `rules-top.drl`: as soon as the observed time reaches a minimal threshold, the top-manager starts checking if the input rate falls in a given range and acts in order to keep this range valid. This behavior is kept until there are enough remaining tasks to compute, else no autonomic actions are taken any more. The manager can detect two type of violation, both represented as error code in the class `gridcomp.example.active.twolevels.ManagersConstants` and signalled by the farm *BeSke*:

- `notEnoughTasks_VIOL` if the input rate needs to be increased: in this case the top-manager fires the operation `PUSH_OBJECT` in order to signal to its controlled components to adequate their behavior;
- `tooMuchTasks_VIOL` if the input rate needs to be decreased: the top-manager takes the same action but modifies the error code signalled to its controlled components.

The input component is connected to the top-manager by binding the `commit-value` port. Since the `input` component is a primitive component, it interacts with the top-manager through a not-functional port implementing the `gridcomp.AutonomicServerManager` interface, as documented by its ADL description:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
"classpath://gridcomp/adl/gridcomp.dtd">

<definition name = "gridcomp.example.active.twolevels.adl.input"
 extends = "gridcomp.PrimitiveController">

   <!-- functional interface that will be bound to the farm  -->
   <interface signature = "gridcomp.example.active.twolevels.LineServer"
              role = "client"
              name = "lineserver-client"/>

   <!-- NOT functional interface that will be bound to the top-manager  -->
   <interface signature = "gridcomp.AutonomicServerManager"
              role = "server"
              name = "set-rate-server"/>

   <content class="gridcomp.example.active.twolevels.InputImpl"/>
   <virtual-node name="input-node" cardinality="single"/>
</definition>
```

The methods implementing the `AutonomicServerManager` in the `InputImpl.java` class describes how the component reacts to the top-manager signals, i.e. how it decreases or increases the time spent between sending two successive tasks and here represented by the value `idt` :

```
public GenericTypeWrapper<Object> commitValue(GenericTypeWrapper<Object> qosValue) {

   // get the value sent by the top-manager
   float increment = ((Integer) qosValue.getObject()).floatValue();
   if (increment != 0) {
      Float idtf = new Float(idt);
      System.out.println("Input: reconf - difference "+(increment*idtf/100));
      // the actual increment is given as a percentage of the current idt
      idtf += (increment*idtf/100);
      idt = idtf.intValue();
      if (idt<0) {
         idt=1;
      }
   }
    else
      // no actions are taken
      System.out.println("Input: reconf - difference ZERO");

    System.out.println("Input: reconf - new sleep time (ms) "+idt+
                                        ", (tasks/s) "+(1000.0/((float) idt)));
    Integer ret = new Integer(remaining_tasks);
    return (new GenericTypeWrapper<Object>(ret) );
}
```

Just to provide an example, let us take a look to the rule defining an increasing of the input rate in the top-manager QoS contract:

```
rule "detectViolationLow"
   when
       $remainingtasks : PushObjectResultBean( )
```

```
        $exception : ViolationBean( value == ManagersConstants.notEnoughTasks_VIOL )
    then
        $exception.setData(new Integer(-50)); // decrease by 100%
        $exception.fireOperation(ManagerOperation.PUSH_OBJECT);
        System.out.println("========== TopMan: task remaining "+$remainingtasks.getValue());
        $exception.setValue(ManagersConstants.No_VIOL);
        update($exception);
end
```

The involved beans are `PushObjectResultBean` that carries an estimation of the remaining tasks and `ViolationBean` that collects the violation messages arriving from the farm through the **`raise-violation`** port. Once the **`notEnoughTasks_VIOL`** violation is detected, the manager defines a valuable time decrement for the **`input`** component and sends it by firing a `PUSH_OBJECT` action (we recall that this action sends objects via the **`commit-value`** port to which the **`input`** component is bound).

6.7 Guidelines for writing a top-manager based application

First of all, it must be taken into account that writing a two-level management application means providing a ”special” component with management, not-functional responsibilities. Thus, the first step is to provide the ADL description and the implementation of such component considering that:

1. the ADL description must extend **`gridcomp.manager.GenericAutonomicManager`** and include the **`gridcomp.manager.AutonomicManagerAttributeController`** controller to set some initial attributes (see Section 6.2 for further details);
2. the implementation class must implement **`org.objectweb.proactive.RunActive`** and all the not-functional interfaces needed to support the multi-level management, depending on the not-functional interfaces defined in the ADL description;
3. its behavior will be driven by a *ad hoc* QoS contract that must be provided as input, following all the characteristics and rules of the contracts already detailed for the *BeSke* components in Section 3;
4. the not-functional interfaces provided by the top-manager can be *selected* (i.e. they are all *optional* interfaces) among:
    - **`raise-violation`**: to collect violation messages from the lower level
    - **`commit-contract`**: to send a brand new contract to a component belonging to the lower level
    - **`commit-value`**: to send new values (i.e. QoS performance requirements) referring the contract currently held by a component belonging to the lower level
    - **`recv-mult`**: to send new values to all the components belonging to the lower level in *multicast* mode.
5. the top-manager is bound to its child component in the canonical manner: the *BeSke* components are already equipped with the optional not-functional port supporting the multi-level management, while a primitive component should be explicitly provided with the not-functional interfaces needed to treat the interaction both at ADL and implementation level.

7 Conclusion and work in progress

We presented the autonomic features related to the GCM implementation of the behavioural skeleton `farm` and `map` given on top of ProActive 3.9. As the traditional skeleton paradigm, the basic idea is to provide the user with a set of behavioural skeletons, i.e. components whose non-functional behaviour is pre-defined, while the functional behaviour is parametrically specified by the user.

Since the non-functional behaviour of such component is pre-defined, we are able to provide a manager driving the behaviour with respect to the user's needs specified through a QoS contract at running time.

Successive development of the skeleton will include:

- unifying the programming methodology of the `farm` and the `map`. In particular, we will describe the first in a more parametric way, allowing the definition of just a worker ADL file and not the set currently required;
- extending or overwriting the manager of the provided skeleton in order to add/change manager capabilities while keeping the skeleton structure and behaviour;
- adding new autonomic operation to both the active and the passive level in order to increase the adaptive power of pre-existent or new skeletons;
- adding new controllers in order to wide the spectrum of autonomic operations at passive level as well as monitoring capabilities.
- extending the multi-level mechanism with new operations and monitoring conditions

Besides the new Priority Controller introduced in ProActive3.9 we introduced some new controllers for components to implement the effector and the sensor of the behavioural skeletons (as an example the monitor controller). These controllers strictly depend on the Proactive implementation of components, and they are generic enough to be reusable by other *BeSke* programmers (exploiting GCM/PROACTIVE implementation).
However, we are working for moving the code onto ProActive 4.0, aiming at using the monitoring features directly offered by the framework.

References

[1] Marco Aldinucci, Sonia Campa, Marco Danelutto, Patrizio Dazzi, Peter Kilpatrick, Domenico Laforenza, and Nicola Tonellotto. Behavioural skeletons for component autonomic management on grids. In *CoreGRID Workshop on Grid Programming Model, Grid and P2P Systems Architecture, Grid Systems, Tools and Environments*, Heraklion, Crete, Greece, June 2007.

[2] Marco Aldinucci, Sonia Campa, Marco Danelutto, Marco Vanneschi, Patrizio Dazzi, Domenico Laforenza, Nicola Tonellotto, and Peter Kilpatrick. Behavioural skeletons in GCM: autonomic management of grid components. In Didier El Baz, Julien Bourgeois, and Francois Spies, editors, *Proc. of Intl. Euromicro PDP 2008: Parallel Distributed and network-based Processing*, pages 54–63, Toulouse, France, February 2008. IEEE.

[3] Marco Aldinucci, Sonia Campa, Patrizio Dazzi, and Nicola Tonellotto. D.NFCF.01 – non functional component subsystem architectural design. `http://gridcomp.ercim.org/`, June 2007.

[4] Marco Aldinucci, Sonia Campa, Patrizio Dazzi, and Nicola Tonellotto. D.NFCF.02 – non functional component subsystem architectural design (code). `http://gridcomp.ercim.org/`, June 2007.

[5] Marco Aldinucci, Sonia Campa, Patrizio Dazzi, Nicola Tonellotto, and Giorgio Zoppi. D.NFCF.03 – methodology to derive performance models for component and composite components. `http://gridcomp.ercim.org/`, June 2008.

[6] Marco Aldinucci, Sonia Campa, Patrizio Dazzi, Nicola Tonellotto, and Giorgio Zoppi. D.NFCF.04 – nfcf prototype and early documentation. `http://gridcomp.ercim.org/`, June 2008.

[7] CoreGRID NoE deliverable series, Institute on Programming Model. *Deliverable D.PM.04 – Basic Features of the Grid Component Model (assessed)*, February 2007. `http://www.coregrid.net/mambo/images/stories/Deliverables/d.pm.04.pdf`.

[8] JBoss rules home page. `http://www.jboss.com/products/rules`, 2008.

[9] ObjectWeb Consortium. *The Fractal Component Model, Technical Specification*, 2003.

[10] Thomas Weigold, Peter Buhler, Jeyarajan Thiyagalingam, Artie Basukoski, and Vladimir Getov. Advanced grid programming with components: A biometric identification case study. In *Proc. of the 32nd Intl. Computer Software and Applications Conference (COMPSAC)*, pages 401–408, Turku, Finland, July 2008. IEEE.