Università di Pisa
Dipartimento di Informatica

Technical Report: TR-10-07

LIBERO: a LIghtweight BEhaviouRal skeletOn framework

M. Aldinucci, M. Danelutto, P. Kilpatrick, V. Xhagjika

April 2010

Address: Largo B. Pontecorvo 3, 56127 Pisa, Italy. Tel: +39 050 2212700 Fax: +39 050 2212726

LIBERO: a LIghtweight BEhaviouRal skeletOn framework

M. Aldinucci*, M. Danelutto†, P. Kilpatrick‡, V. Xhagjika§

April 2010

Abstract

We describe a lightweight prototype framework designed for experimentation with behavioural skeletons. A behavioural skeleton is a component implementing a well-known parallelism exploitation pattern *and* a rule-based autonomic manager taking care of some non-functional feature related to the parallel computation. Our prototype supports multiple autonomic managers within the same behavioural skeleton, each taking care of a different functional concern. The different managers in the behavioural skeleton coordinate themselves in such a way that a global, user-provided SLA can be satisfied. We discuss experiments that validate the manager coordination protocol, and the overall prototype functionality. The prototype is built on top of plain Java and employs JBoss rules for management. We present experimental results that demonstrate the operation of our prototype and allow overheads to be evaluated.

Keywords: structured parallel/distributed programming, behavioural skeletons, non functional concerns, performance, security, autonomic management, business rule systems.

1 Introduction

Behavioural skeletons (BS) have been introduced to tackle the problem of efficient, autonomic management of non-functional features of parallel/distributed computations, such as performance, security, fault tolerance, power management, etc. A behavioural skeleton is the result of the co-design of a well-known, efficient parallelism exploitation pattern *and* of a rule-based control loop implementing an autonomic manager of (one or more) non-functional properties related to the parallel computation pattern [1, 2]. On the one hand, the parallelism exploitation pattern makes better use of well-understood and efficient

*Dept. Computer Science, Univ. of Torino, Italy
†Dept. Computer Science, Univ. of Pisa, Italy
‡Dept. Computer Science, Queen's Univ. of Belfast, UK
§Dept. Computer Science, Univ. of Pisa, Italy

techniques used to implement that particular pattern on parallel and distributed target architectures. On the other hand, the autonomic manager executes a classical MAPE[1] control loop. At each iteration, a set of *pre-condition* $\rightarrow$ *action* rules is evaluated and one of the fireable rules–those whose pre-condition evaluates to true–is executed. Pre-conditions use monitored values from the system and actions are defined in terms of a set of pre-defined actions supported by the system.

Behavioural skeletons were originally designed in the framework of GCM, the Grid Component Model [8, 9] developed within the EU NoE project CoreGRID[7] and subsequently implemented in the GCM reference implementation built on top of ProActive [14] in the EU STREP project GridCOMP [11]. In GridCOMP, behavioural skeletons were implemented modelling common stream parallel patterns–such as pipelines and farms–with managers taking care of performance issues. Those BS have been demonstrated to be effective in managing and enforcing user-supplied (best effort) *performance contracts*. In [2] it was shown how contracts requiring a given throughput can be guaranteed when a single BS is used to model the entire application. In [3] we introduced techniques that support the coordination of the different managers in a BS hierarchy used to model complex parallelism exploitation patterns, in such a way that a (single concern) performance contract provided by the user is ensured.

In the general case, however, *multiple* non-functional concerns have to be addressed within the same computation. The BS concept can be easily extended in such a way that multiple managers are associated to the same parallel pattern, each taking care of a different concern. In [4] we identified the need for such managers to interact to achieve consensus before effecting changes to the managed system and identified protocols for achieving such consensus. Here we show how such management may be realised in practice, using a lightweight framework designed to facilitate experimentation with a set of interacting autonomic managers. As an example, consider a computation where both a performance contract and a security contract have been supplied by the user. The performance contract asks for a given throughput. It can be ensured by recruiting increasing numbers of resources up to the point where the required throughput is delivered. New resources may also be dynamically recruited to the computation in the event that existing ones become less effective due to temporary overloads or faults. The security contract demands that, where nodes are recruited from external, possibly unreliable domains, such nodes must be suitably secured by, for example, encrypting data and code communications; nodes internal to the user domain may be considered secure. Thus, if the performance manager identifies failure of the performance contract it will prompt the recruitment of further resources. If some of these are in an external domain the security manager may in turn demand the securing of communications with such potentially unsafe resources.

In this work, we first detail problems related to autonomic management of multiple non-functional concerns (Sec. 2), then we introduce LIBERO [15], a

[1] Monitor, Analyse, Plan, Execute

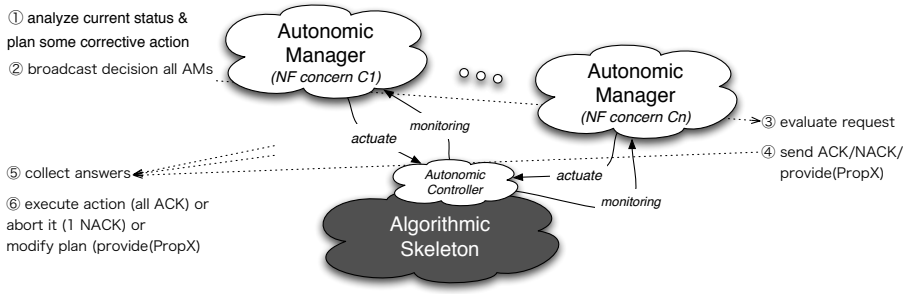


Figure 1: Coordinating activities of distinct autonomic managers in a BS

prototype supporting BS with multiple managers taking care of different non-functional concerns (Sec. 3). LIBERO is a lightweight prototype in that it relies only on the existence of Java on the target machines and on the possibility to place an RMI-based runtime on those machines. Unlike the former implementation of BS–that only supported one manager per skeleton–it does not require the installation of complex "middleware" systems such as ProActive.

Finally we discuss experimental results demonstrating the functionality of LIBERO and its suitability to support a multi-concern use case modelling the performance/security management scenario outlined above and *de facto* assessing the techniques suggested in [4] for the support of multiple concern autonomic manager coordination (Sec. 4)

2 Autonomic manager coordination

Problems such as that outlined in Sec. 1 with respect to performance and security manager coordination may arise when *independent* autonomic managers are run within the same behavioural skeleton. In a scenario such as that depicted in Fig. 1, multiple managers are associated with the same algorithmic skeleton in a single behavioural skeleton. The algorithmic skeleton implements a well-know parallelism exploitation pattern. Through its *autonomic controller* (AC) it provides i) methods to access its internal state (to support monitoring activity) and ii) methods to operate on its internal state (to support implementation of actions modifying its behavior). Each of the associated autonomic managers takes care of a distinct non-functional concern. It periodically executes a control loop monitoring algorithmic skeleton behaviour and, possibly, planning and executing actions aimed at improving system behaviour with respect to the non-functional concern managed by it.

In [4] we proposed some coordination protocols that can be used to coordinate manager activities. In particular, we evaluated the feasibility of a two-phase approach where each action planned by an AM is validated by the other AMs in the behavioural skeleton before being actually executed. As shown in

Fig. 1, the manager taking care of non-functional concern X (e.g. performance), analyzes system behaviour and decides to take some action ①. It informs the other managers of the decision ②. The other managers evaluate ③ the decision–which is given as a modification to the current process graph implementing the parallel pattern modelled by the algorithmic skeleton–with respect to the consequences (if any) for their non-functional concern. Eventually they return ④ one of three answers: `ACK`, meaning the decision can be safely taken by the first manager, `NACK`, meaning the decision is in conflict with the managed non-functional concern and therefore should be aborted, or `provide(property)`, meaning the decision may be actuated provided `property` is ensured (e.g. securing of connections). The manager initiating the process gets answers from all the other managers ⑤ and eventually either actuates its decision (the original plan or the one modified to accomplish `property`) or aborts it ⑥.

This two-phase protocol has not, prior to now, been experimented with, due mainly to the difficulty of embedding a complex management structure in the reference implementation of BS in ProActive/GCM. We decided to implement LIBERO to allow us to assess the feasibility of this protocol as well as to experiment with other protocols regulating autonomic management of multiple concerns in behavioural skeletons. The need for a simple framework for experimentation is made even more apparent when one considers that BS are usually nested and thus autonomic management of even a single non-functional concern involves the interaction of different autonomic managers, as detailed in [3].

3 LIBERO

LIBERO is a prototype supporting behavioural skeletons with multiple autonomic managers implemented using lightweight components. A component, as a constituent element of a system, is an entity entrusted with some activity, part of the system's overall purpose, with a well-defined public interface. In LIBERO the components are either Communication Components (CC), Autonomic Managers (AM) or Behavioural Skeletons (BS).

Each notable component actually implementing some kind of parallel computation has a managing entity–the AM–that deals with the *non-functional* aspects of the parallel computation in a local and autonomic way. LIBERO AMs implement a lightweight management program. The AM management functions operate on the components of the system through the operations provided by the component "membrane"–the AC–that exports its internal computation state and provides a controlled set of operations to modify component state and functioning.

LIBERO implements the behavioural skeletons already investigated in GridCOMP, namely those BS modelling the usual stream parallel patterns, such as *task farm*s and *pipeline*s [6] and equipped with a single autonomic manager, taking care of a single non-functional concern.

In addition, LIBERO also supports a *Multiple Concern Management* concept, implementing a decision coordination algorithm among managers, such as that

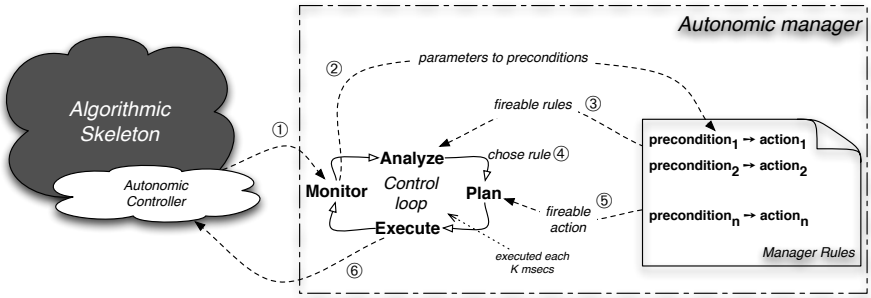


Figure 2: Autonomic manager at work

outlined in Sec. 2. In fact, LIBERO has been designed and implemented precisely to facilitate investigation of problems related to multi-concern management. As all of the components described above are native *Java* objects or POJO[2] in LIBERO, the experiments needed to investigate the multi-concern concepts and implementation turn out to be easier to implement than was the case in the ProActive/GCM BS prototype as i) the ProActive/GCM BS prototype does not support multiple AMs in a single BS, at the moment, and ii) it requires a heavier and more complex runtime. Indeed, LIBERO uses the DROOLS[3] library middle-ware to implement autonomic managers control cycle (see Fig. 2), which is the same middleware supporting AM control loop in ProActive/GCM BS implementation.

In the following subsections a detailed description of the overall framework implementation will be given, starting with the utility components and then discussing all of the objects described up until now.

3.1 Remote node management

LIBERO implements component deployment on remote nodes using a small Java RMI[4] based runtime supporting a lightweight life cycle management of the components. The runtime support allows deployment of LIBERO components on the machine it is running and also management of the life cycle of the deployed component. Management activities access the runtime to check machine dependent parameters unique to the node where the runtime is running, and may also access parameters associated with the other nodes of the system, if needed.

A dual constructor approach ensures smaller network traffic for deployment activities. A *local* constructor constructs the component on the node where it is deployed, while *remote* constructors contact the runtime of the target node and deploy onto the remote node RTS the local constructors that will eventually be used/executed to implement the deployed component. Thus the only

[2]Plain Old Java Objects
[3]DROOLS Expert library is used for rule oriented logic implementation, Knowledge Engine
[4]Remote Method Invocation

information sent between the nodes is that regarding what to construct and not the serialization of the entire object to be deployed.

3.2 Communication components

Separation of functional and non-functional concerns drove the design for their respective interfaces on CC. Considering *functional* concerns we were interested in having generalized, efficient and lightweight interfaces. These interfaces were implemented using permanent Java TCP socket connections, with the use of serialisation for input/output object delivery between BS components. Connection components are associated with a BS and have a straightforward lightweight life cycle[5]. These connection components implement many-to-many communications using either normal or SSL TCP connections, and a three way handshake mechanism for data streaming. Permanent TCP connections for the *functional* interfaces imply the use of a discovery component to locate the distributed components. Implementation of such a discovery mechanism is carried out with the assumption of there being a global naming scheme for the components. A multicast discovery centralised component is used as a *Nameserver*, keeping track of component distribution and relative server ports. This component allows registration, removal and lookup requests using the specified component ID's.

On the other hand, interfaces for the *non-functional* concerns need to have a stronger expressiveness and ease of use. Therefore in LIBERO managers communications are transparent and use RMI as a means of data and control communication. This choice makes it easier to specify the management contract for the management components, so that the user needs only to operate on objects and need not be concerned about communication.

3.3 Supported BS

The LIBERO framework was designed and implemented focussing attention on having a lightweight implementation of Behavioural Skeletons primarily for experimenting with management policies, but also for ease of extendability. Implementation starts with a common abstract class (`BehaviouralSkeleton`) [15] for all of the BS components. This common class implements standard behaviour for manager and sub-component registration/removal, default startup and connection related activities. BS components extend this class and add custom behaviour for the operations and for the functions over the input that they implement. The supported BS components implemented to date are summarized in Table 3.

Implementation of new kinds of BS is supported through extensions of the abstract class `BehaviouralSkeleton`, including the addition of controller operations and supplying a local and a remote constructor. As a simple example we outline the implementation of the Farm BS. The default abstract class behaviour

[5]Data is read from the input, data is processed by the associated BS and then sent to the destination

Sequential	❏ The constructor supports merge and split operations, so that sequential code can be divided/united into smaller/larger pieces. ❏ No parallelism operation is implemented. ❏ Code can be assigned in the constructor as a class implementing a single method that executes some function over some input. ❏ Default measures supported are service time and total task number.
Farm	❏ The constructor accepts as parameters the worker component (in the form of other constructors), the number of initial workers to be allocated and the set of nodes for worker allocation. ❏ It implements increase/decrease parallelism operations such that the number of active workers is increased or decreased to match user supplied contracts. ❏ No code or function is assigned directly to this BS. The function is passed to the constructor of the workers. ❏ Default measures supported are service time, total task number and number of workers.
Pipeline	❏ The constructor accepts stages in the form of other constructors, the order in which these constructors are passed is the order in which the stages will be linked. ❏ No parallelism operation is implemented. ❏ No code or function is assigned directly to this BS. The function is passed to the constructor of the stages. ❏ Default measures supported are service time and total task number.

Figure 3: LIBERO Behavioural Skeletons

is changed so that on component registration/removal the sub-component is registered/unregistered from the farm destination. Counters for worker distribution on the nodes are incremented/decremented in such a way that the farm component may have a clear view of the state of execution of the sub-components. Increase/Decrease parallelism degree operations are also implemented so that a worker can be added/removed while trying to maintain a uniform distribution of workers on the set of possible nodes.

3.4 Autonomic manager implementation

Multiple managers, specialized by their contracts, can be associated with the same LIBERO BS. The actions of these cooperating AM are coordinated by means of a two-phase protocol, such as that proposed in [4]. In the first phase, the AM planning an action seeks consensus of the other managers associated with the same BS. In the second phase, upon receiving unconditional or conditional ACK or a NACK messages from all the other managers, the AM eventually i) commits the planned action, ii) commits the action with a slightly modified execution plan, or iii) aborts the action.

The AM behaviour is expressed in terms of JBoss rules. At runtime the rules are compiled and executed by the DROOLS middleware. The rules are expressed according to a *pre-condition* $\rightarrow$ *action* form, with both parts being expressed in plain Java.

The LIBERO runtime support is used to deploy different managers to the different target nodes. Before entering its life cycle, the manager commits to the DROOLS working memory[6] a reference to the associated BS component and a reference to itself. References to the other managers associated to the

[6]The working memory is the set of all objects known to be true for the knowledge engine.

same BS can be retrieved by calling a proper BS accessor method. The life cycle of the manager is timed by a compile time constant and is divided into the classical MAPE phases: i) in the *Monitor* phase, sampling of the associated component execution state is performed, ii) in the *Analise* phase, the sampled execution state is analyzed and decisions regarding rule applicability are taken, iii) in the *Plan* phase, a decision on which of the applicable rules has to be executed is taken, and finally iv) in the *Execute* phase, the actions specified by the selected rules are executed. The phases described are all delegated to the DROOLS runtime, except the monitoring phase (see Fig. 2). The DROOLS runtime offers a stateless knowledge engine, representing facts with plain Java objects and knowledge in the form of rules.

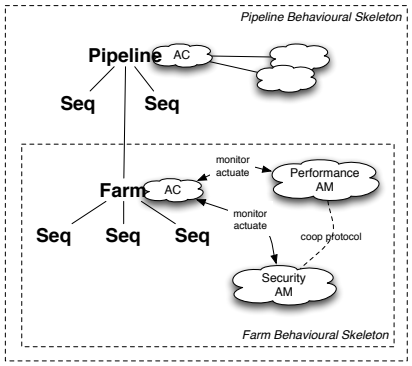
To ensure that the operations planned by an AM after firing a given rule may actually be executed, the mangers implement the two-phase protocol described in Sec. 2. This consensus protocol is implemented using suitable JBoss rules and using runtime values as contract parameters, in such a way that the protocol is not actually embedded in the manager code but in the rule language.

3.5 Autonomic controller implementation

The Autonomic Controller interface extends the normal definition of a component with the means to export the state of the components and allow modification of the internal state. The AC interface is part of the *BehaviouralSkeleton*[15] and as such default behaviour for associated manager and sub-component accessors is already implemented by the abstract BS class.

After inheriting the abstract class the BS component is forced to implement behaviour for the *executeOperation* and *getMeasure* methods. These two methods use enumerations–that represent the measures or the operations that can be applied to the component–as their parameters in order to change/export the internal execution state. The AC also implements methods for accessing machine dependent parameters, fetched from the runtime support of the node. Extendability of these features is as easy as defining the new behaviour for the operations or the new measures to be exported, and then making them available by adding the new descriptors for those activities in the appropriate enumerations.

Machine dependent properties are made accessible through the runtime support; these properties are described in an XML file parsed at startup by the runtime. The configuration file may host properties relative to all of the machines used for program execution. The metadata syntax is simple. At the moment we just use a tag "MACHINE" with property "ip" that specifies the machine the properties apply to, and child tags "PROPERTY" with attributes "name" and "value" that describe the machine dependent properties.




```
rule "FarmPerformanceManagerRuleToAskForConsensus"
 when
  $farm: AutonomicControllerInterface()
  $manager: AutonomicManagerInterface()
  $sample: String() from
      $farm.getMeasure(Measures.NEXT_AVAILABLE_MACHINE)
  $sample_numworker: Integer() from
      $farm.getMeasure(Measures.TOTALWORKERS)

   not(exists(ContractParamValue(name ==
       MulticoncernBroadcastCodes.BCAST_REQUEST_WAIT_ACK)))
   not(exists(ContractParamValue(name ==
       MulticoncernBroadcastCodes.PREPARE_BCAST_COMMAND)))

   eval(((Integer) $sample_numworker) < 8)
then
  $manager.setContractParam(
       MulticoncernBcastCodes.PREPARE_BCAST_COMMAND, "");
  $manager.setContractParam(
       MulticoncernBcastCodes.BCAST_PARAM,
       CommandCode.INCREASE_PARALLELISM);
  $manager.setContractParam(
       MulticoncernBcastCodes.BCAST_SECOND_PARAM,
       $sample);
end
```

Figure 4: Sample use case application (left) and Sample JBoss rule (right)

4 Experimental results

A full set of experiments has been performed, aimed at verifying LIBERO functionality. The architecture used for the tests is a cluster of 25+ dual core Linux machines running Java version 1.5 and supporting file sharing through NFS. Actually, the only requisite for the prototype is presence of Java 1.5+ on the target nodes and the presence of DROOLS 5.0 library. An extensive work on testing was done to assure that the components and the system as a whole exhibit the desired behaviour. At first the actual deployment of the components was tested to assure the distributed nature of the system, and the distribution mechanisms pass the test by allowing every object to be located on their desired destinations. Next the communication primitives and the `Nameserver` component was stressed to see effective usage performance. After multiple runs of complex BS were run the analysis of execution logs showed that all of the features described in this paper had a compliant behaviour, matching all of the expectations made for the system [15].

After functionality tests, we developed a specific use case to demonstrate how the new features of LIBERO perform. This use case is a synthetic application structured as a three stage `Pipeline` component: the first and the third stages are `Sequential` components, while the second stage is a `Farm` component. Each component is placed on a different node in the cluster and 3 machines are assigned as resources for the `Farm` workers. The LIBERO runtime is executed on each of the nodes needed for the execution of the use case application.

Two autonomic managers are associated to the Farm component, one handling security concerns and the other performance ones. The simple contract supplied to the performance manager specifies that a total of 8 workers should be reached and maintained, and that the two-phase broadcast consensus pro-

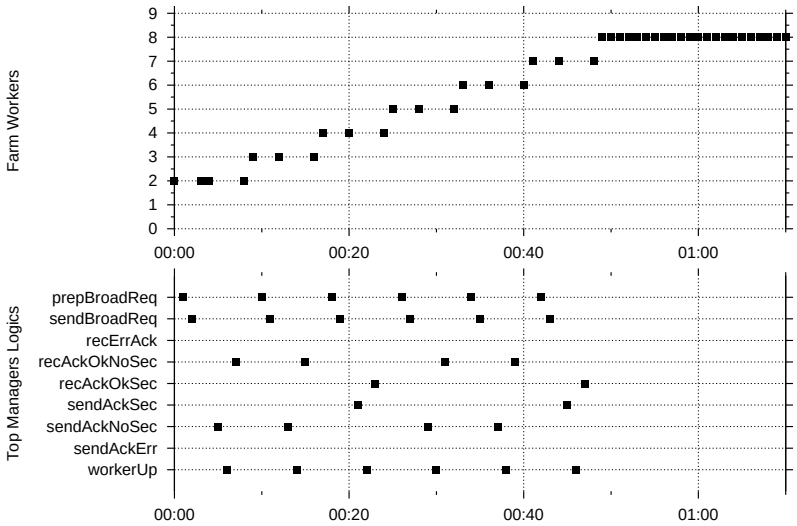


Figure 5: Event distribution over time (ms from system execution startup). *W.r.t. Fig. 1:* prepBroadReq *corresponds to* ①, SendBroadReq *to* ②, recAckOkNoSec/recAckOkSec *to* ③, SendAckSec/SendAckNoSec *to* ④, workerUp *to* ⑥. *Events* ③ *and* ④ *are relative to the Security AM, all the others are relative to the Performance AM*

tocol should be used to implement new worker allocation. The run time nodes host facts stating whether the nodes possibly recruited to implement new `Farm` workers are secure or insecure, that is, if they can be reached using plain TCP or if SSL should be used instead to increase the security level. We used both secure and insecure nodes in the experiment. This allowed to us to check both types of answers from the consensus phase: simple ACK (i.e. accept recruitment of a new node to host a `Farm` worker implemented using plain TCP sockets) and conditional ACK (i.e. accept recruitment of the node provided SSL sockets are used for communications). The life cycle of the managers is set to 500ms so that the plot of the runtime is discrete enough to allow observation of the events, but smaller life cycles are possible down to 100ms.

In our use case application, we start the Farm component with just two workers. The performance manager immediately detects a violation of the contract and asks the other managers for permission to add another worker. The security manager in turn responds with an normal ACK if the machine is secure or with a SECURE_ACK if secure connections are to be used. If other violations

are encountered then the same set of operations is applied again and again, until no further violation is encountered.

The plot in Fig. 5 shows the evolution of the Farm component and the distribution of manager events over the same period of time. As can be seen from the plot the consensus protocol takes an overhead of at most 4 manager life cycles plus the execution time of the rules that depend only on the communication overhead between managers. This gives a total overhead time of $T_{overhead} = 4 * (T_{LyfeCycle} + T_{Com})$, where T_{Com} is the average amount of RMI calls * average RMI latency. In this simple case the entire reconfiguration of the system takes 45s, and reconfiguration time needed for worker allocation on average (accounting also for manager decision making and synchronization) is about 5 secs (including about 2 secs of idle time spent in waiting 4 times for the next iteration of the control loop to take place). These times are of the same order of magnitude as the times spent using in ProActive/GCM BS prototype to achieve an unmediated reconfiguration (e.g. a reconfiguration decided autonomically by a single, uncoordinated manager), which represents quite a good result and underlines the "lightweight" nature of the LIBERO implementation.

5 Related work

The IBM blueprint paper on autonomic computing has already established, in a slightly different context, the need to orchestrate independent autonomic managers [12]. In [10] strategies to handle performance and power management issues by autonomic managers are discussed. However the approach is much more oriented to the generic combination of target functions relating to the two non-functional concerns considered, rather than to the constructive coordination of the actions planned by the two managers.

A framework that can be used to reason on multiple concerns was introduced in [13]. Based on the concepts of state and action (i.e., state transition) adopted from the field of artificial intelligence, this framework maps three types of agenthood concepts (action, goal, utility-function) into autonomic computing policies. Action policies may produce and consume resources, which are used by a *resource arbiter* (i.e. a super manager) to harmonize conflicting concerns. The framework, however, does not provide any specific support for policy design and distributed management overlay.

A similar approach was followed in [5], which also exploits the same policies (action, goal, utility-function) defined on the (Cartesian product of) *state* and *configuration* space of the system. These policies are extended with *resource-definition* policies, which specify how the autonomic manager exposes the system to its environment; this makes it possible to dynamically extend manager knowledge with other resources/parameters, possibly coming from other managers, thus supporting management overlay.

6 Conclusions

We have described a lightweight implementation of a behavioural skeleton framework supporting the implementation of skeletons with multiple autonomic managers, each managing a different non-functional concern. The prototype has been developed to allow investigation of different aspects of autonomic management of non-functional concerns. The lightweight implementation of LIBERO allows us to experiment with various consensus building strategies without being burdened by the complexities of fully-fledged distributed/parallel system implementations. After assessing prototype functionality, we discussed an experiment aimed at validating the two phase manager coordination protocol introduced in [4]. LIBERO successfully ran the protocol with moderate programming effort and notable efficiency, comparable to that achieved when running much simpler behavioural skeleton programs in the ProActive/GCM reference implementation. Having established the efficacy of LIBERO as a test vehicle, we are now in a position to conduct more ambitious experiments with distributed autonomic management.

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, Marco Danelutto, and Peter Kilpatrick. Autonomic management of non-functional concerns in distributed and parallel application programming. In *Proc. of Intl. Parallel & Distributed Processing Symposium (IPDPS)*, Rome, Italy, May 2009. IEEE.

[4] Marco Aldinucci, Marco Danelutto, and Peter Kilpatrick. Autonomic managenemt of multiple non-functional concerns in behavioural skeletons. In *Proc. of the CoreGRID Symposium 2009*, CoreGRID, Delft, The Netherlands, August 2009. Springer.

[5] Radu Calinescu. Resource-definition policies for autonomic computing. In *Proc. of the 5th Intl. Conference on Autonomic and Autonomous Systems (ICAS)*, pages 111–116. IEEE, April 2009.

[6] Murray Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004.

[7] The CoreGRID home page. `http://www.coregrid.net`, 2007.

[8] CoreGRID NoE deliverable series, Institute on Programming Model. *Deliverable D.PM.02 – Proposals for a Grid Component Model*, November 2005.

[9] CoreGRID NoE deliverable series, Institute on Programming Model. *Deliverable D.PM.04 – Basic Features of the Grid Component Model (assessed)*, February 2007.

[10] Rajarshi Das, Jeffery O. Kephart, Charles Lefurgy, Gerald Tesauro, David W. Levine, and Hoi Chan. Autonomic multi-agent management of power and performance in data centers. In *Proc. of the 7th Intl. Conference of Autonomic Agents and Multiagent Systems*, May 2008.

[11] GridCOMP Project. Grid Programming with Components, An Advanced Component Platform for an Effective Invisible Grid, 2008. `http://gridcomp.ercim.org`.

[12] IBM Corp. *An Architectural Blueprint for Autonomic Computing*, 2005. `http://www-01.ibm.com/software/tivoli/autonomic/`.

[13] Jeffrey O. Kephart and William E. Walsh. An artificial intelligence perspective on autonomic computing policies. In *Proc. of the 5th Intl. Workshop on Policies for Distributed Systems and Networks (POLICY'04)*. IEEE, 2004.

[14] ProActive home page, 2009. `http://www-sop.inria.fr/oasis/proactive/`.

[15] Vamis Xhagjika. Implementation of a prototype for experimenting with autonomic hierarchical managers in JAVA. Dept. of Computer Science, Univ. of Pisa, Italy (Thesis, In Italian), December 2009.

A User Manual

A.1 Introduction to the runtime environment

As a conclusion of our presentation of the LIBERO framework we give a full description of the runtime mechanism and user iteration needed to activate and run a sample project. The environment needed to support normal development and running activities is described in the following enumeration. It is worth pointing out that the enumerated entities are the only things needed to develop and run a sample LIBERO application:

- ***Drools Expert*** middle-ware library needed for correct manager usage as mentioned in Sec 3.4. This middle-ware implements a rule oriented knowledge engine using an efficient RETE algorithm for rule satisfiability. The development and execution of LIBERO programs requires version 5.0 of the DROOLS Expert library. This library can either be placed under the lib folder of the framework, or the lib folder can be linked to the library position on the system.

- ***NFS support*** NFS is currently needed to support code distribution and access across the available processing elements.

- ***Java 1.5+ runtime support*** The java runtime is a necessity since the entire program is written in plain Java programming language and as such it needs to be compiled by the Java compiler, and executed by the Java Virtual Machine.

- ***Utility "make"*** This optional utility is needed for automatic compilation of the application. The actual commands supported for this utility are "*make lin*" and "*make win*". Those in term compile the application in a linux or windows based system. Compilation can be done prior to distribution, since the bytecode is Java dependent and not machine dependent, thus making it possible for the code to be executed on every architecture as long as the appropriate Java runtime is present.

Having introduced all of the components needed for normal code execution and operation we outline the fact that documentation is embedded in the source code and can be generated anytime, using the `javadoc` utility. Fig. 6 shows a snapshot of the resulting documentation window. In the next subsection we discuss how to achieve a complete setup and usage of the framework and of the sample application.

A.2 Setup/Compilation/Execution

An extensive step-by-step procedure for example compilation and execution of a small basic LIBERO application should proceed through the following steps:

1. Create a directory on the filesystem containing the framework files.

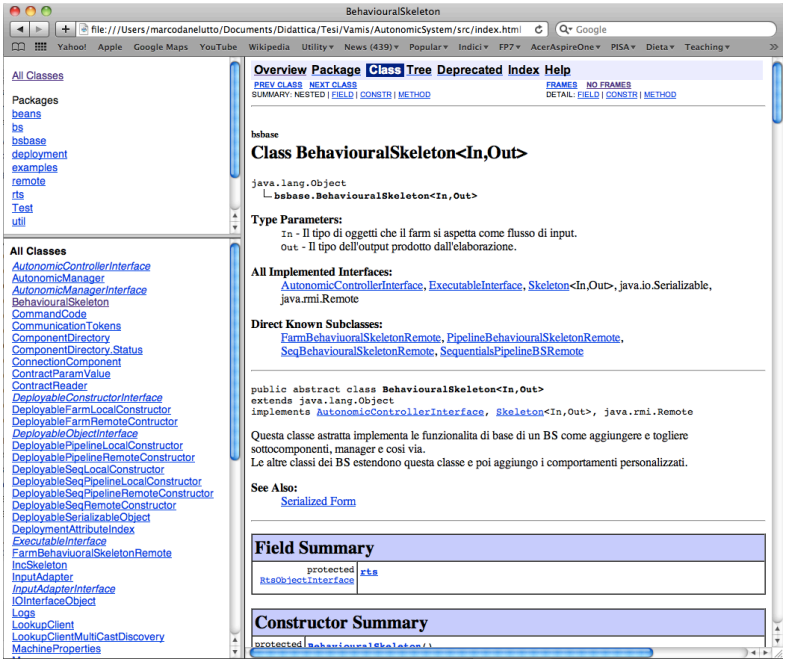


Figure 6: Sanpshot of the javadoc documentation of LIBERO

2. Get a copy of the latest DROOLS Expert runtime(current version is 5.0) and place the unzip-ed content to the "lib" directory of the framework.

3. Browse to the base directory of LIBERO and execute `make lin` or `make win` to compile under Linux or Windows. A base Java `makefile` makes proper compilation possible for the project and the test program; be sure to modify such file to compile additional class implemented by the user. Note that the existence of two different make commands is due to different syntax for path specification between Linux and Windows and not for compilation dependency purposes.

4. If NFS support is not present copy the framework folders to the file systems of the nodes that will be used during the execution of the framework.

5. Select one node to act as a the `Nameserver`. This node can also be used for other purposes but keeping it in a different node from the runtime nodes ensures runtime decoupling of this component from the system measurements. After selection of a suitable node execute the following command[7]:

   ```
   java -cp .:bin:lib/*:lib/lib/*
   ```

[7] This component acts as a centralised logging utility and it is of utmost importance it is launched first. If it is not launched as the first process, the system hangs when as soon as a log operation is tried.

```
remote.NameServerMulticastDiscovery &
```

6. On every node that will be used during the system active lifetime execute the runtime support of the LIBERO framework[8]:

   ```
   java -cp .:bin:lib/*:lib/lib/* rts.RTS &
   ```

7. The main application is executed in one of the free nodes. In the test case implemented with the framework the command is:

   ```
   java -cp .:bin:lib/*:lib/lib/* Test.TestPipeline &
   ```

After the application terminates the directory "etc" holds the log file for the entire system.

B Library Description

In the following subsections we continue our treaty of the LIBERO framework giving a description of the usage of the Behavioural Skeletons implemented in the framework and also of the Autonomic Manager components. We start describing the constructor of the Autonomic Manager and then we continue with the Sequential BS, Farm BS, Pipeline BS.

B.1 Autonomic Manager

Descriptions given in this subsection, cover the explicit declaration of an Autonomic Manager component. During actual BS usage in the sample application also shown in Listing 3, Autonomic Managers are created implicitly by passing the desired contract file names, as a string array to the BS component (Listing 3 line 16,17) constructor.

The constructor, prior to the construction of the actual BS component, parses the string array with AM names and generates an AM for each of the file names in the list. Using this implicit creation process relieves the programmer of the necessary code for AM setup and delegate this activities to the constructor of the BS, so that the programmer only has to care of the BS components and of contract specification. The drawback of this technique is that the AM components are deployed on the same node as the BS component, and the user can't specify destinations other than the BS component node.

Listing 1 shows the code necessary to deploy an AM component onto a different node, to bind it to a BS component and to make it enter the active phase. First we define the id of the component to which the AM will be associated and the file name of the contract to be used (Listing 1 lines 1-2). Then we get a handle at the remote component to which the manager will be associated

[8] If there is successful termination of the program the runtime can be used again without needing a restart of the service. In case of developer introduced bugs the runtime services need to be restarted to ensure correct operation

(Listing 1 lines 5-7), in the code we use a library function that returns a remote reference to the BS object with gobal identifier `id` and located on node whose qualified name is `location`. Lines 9-10 show uses of a static method of the `ContractReader` class called `readContractFromDrl` that reads the contract from a standart Drools DRL file and returns it as a Java string.

The AM contract is passed as a second argument to the `AutonomicManager` constructor. The first argument is the global identifier of the AM component (Listing 1 lines 12-13). We deploy the manager (Listing 1 line 15) using a static function of the utility class `RMIUtils.rtsDeploy(...)` that basically serialises the AM, sends it as a `DeployableObject` using the runtime support and makes sure the runtime exports it for remote activity. Once the component is deployed it is executed (Listing 1 line 17) and a remote reference of the component is retrieved (Listing 1 line 19-20). The only thing left to do is to associate the AM to the BS component using the `commitManager` method of the BS component and tell the manager to pass into an active life cycle execution mode (Listing 1 line 22-23).

Listing 1: Explicit Manager Creation

```
1  String id = "NomeComponenteBS";
2  String nome_manager = "NomeFileContratto";
3  String location = "axth1.cli.di.unipi.it";
4
5  AutonomicControllerInterface refBs =
6    (AutonomicControllerInterface)
7      RMIUtils.lookupResource(location, id);
8
9  String mContract =
10   ContractReader.readContractFromDrl("rls/" + nome_manager + ".drl");
11
12 AutonomicManager myManager =
13   new AutonomicManager("DEFMAN_" + id + nome_manager, mContract);
14
15 RMIUtils.rtsDeploy(location, "DEFMAN_" + id + nome_manager, myManager)
16
17 rts.runComponent("DEFMAN_" + id + nome_manager)
18
19 man = (AutonomicManagerInterface) RMIUtils.lookupResource(location,
20          "DEFMAN_" + id + nome_manager);
21
22 refBs.commitManager(man);
23 man.enterActive();
```

B.2 Autonomic Behavioural Skeletons

After discussing the creation of Autonomic Managers we conclude our outline of the framework with a small description on the procedures necessary to create BS

Listing 2: BS Remote Contructors

```
1  public DeployableSeqRemoteConstructor(
2    String id, String location, String destination,
3      String[] managerContract, Skeleton<?, ?>[] code,
4        SecurityDescriptor security);
5
6  public DeployableFarmRemoteContructor(
7    String id, String location, String[] workerLocations,int initial,
8      String[] manager, DeployableConstructorInterface<In, Out> code,
9       SecurityDescriptor security);
10
11 public DeployablePipelineRemoteConstructor(
12   String id, String location, String[] managerContract,
13     DeployableConstructorInterface<?, ?>[] code,
14       SecurityDescriptor security);
```

components. Listing 2 shows the signatures of the BS component constructors for the BS implemented in the LIBERO framework.

The first BS component constructor we describe is the Sequential BS. Listing 2 lines 1-4 enumerate the BS parameters for the Sequential BS. The first parameter is the global identifier of the component, then we have the location (the node where the component will be deployed), the identifier of the destination component (where to send the result of the computation), a list of strings with contract file names[9] to create managers for this component, an array of objects implementing the functional behaviour of this BS component[10], and finally a security description for the connection components.

Having discussed the Sequential BS we move on to the Farm BS (Listing 2 lines 6-9). The constructor, as in the Sequential BS case, expects as first parameters the global identifier of the component and the location where such component will be deployed, then a list of node names where the workers will be distributed (round robin), the initial number of workers, a list of manager contract file names, a remote constructor of some kind of BS (the worker BS, actually), and finally a security description for the connection components.

Last but not least, we'll introduce the remote constructor for the Pipeline BS. Actually, we will limit our description of this component to the description of the code parameter, since the other parameters are the same of the other BS components. The code parameter of this constructor accepts a list of Remote Constructors that will be used to generate the stages of the pipeline in order of appearance (the first element in the array will be used for the first stage, the second for the second stage and so on...). The elements of this array can be any kind of remote constructor implemented in the LIBERO framework.

[9] This is the implicit way of creating managers associated with the component being created.

[10] In case the functional code is expressed as an aggregation of some sequential code, the composition of the functions described by the objects in order of appearance will constitute the BS functional code.

The next subsection of this Appendix describe a sample application implemented using the LIBERO framework to show how the framework can be used to create BS components as well as the components can be used.

B.3 Sample application

Listing 3 shows a simple framework usage. We give a quick overview of the construct used in the example and delegate a detailed explanation to [15]. This sample code describes the construction of a Pipeline with three stages. The first stage is a Sequential BS with global ID `Skeleton1` to be deployed on the machine `axth1.cli.di.unipi.it` and having as destination the component with global ID `Farm1`. `Skeleton1` has no manager and as function uses the `code` object, that takes an `Integer` as parameter and increases it (in order of appearance Listing3 line 15-21).

The second stage is a Farm component. The only important difference with the previous stage construct, is that some additional parameters are passed to the constructor that decide the Farm topology. The set of nodes to use for worker placement is given as a fixed set[11], 2 workers are included in the initial configuration, and two autonomic managers[12] are set to be created. Finally as described by the security descriptor, this component sends data to a non secure node (in order of appearance Listing 3 line 28 - 33).

The third stage is again made of a Sequential construct, as the first stage, and introduces no new structural concept.

Finally the last appearing element represents the Pipeline with the newly created remote constructors described above.

Listing 3: Example usage

```
1  IOInterfaceObject<Integer, Integer> con =
2          new IOInterfaceObject<Integer, Integer>("Temp");
3
4  DeployableConstructorInterface<?, ?> worker[] =
5          new DeployableConstructorInterface<?, ?>[3];
6
7  DeployablePipelineRemoteConstructor<Integer, Integer> pipe = null;
8
9  AutonomicControllerInterface remoteBS = null;
10 Skeleton<?, ?>[] code = new Skeleton<?, ?>[] { new IncSkeleton() };
11
12 DeployableSeqRemoteConstructor<Integer, Integer> farmWorker =
13         new DeployableSeqRemoteConstructor<Integer, Integer>(...);
14
15 worker[0] = new DeployableSeqRemoteConstructor<Integer, Integer>(
16                 "Skeleton1",
17                 "axth1.cli.di.unipi.it",
18                 "Farm1",
```

[11] (`axth3.cli.di.unipi.it`, `axth3.cli.di.unipi.it`, `axth3.cli.di.unipi.it`)
[12] (`FARM_MULTICONCERN_MANAGER`,`FARM_MULTICONCERN_SECURITY_MANAGER`) respectively

```
19                  null,
20                  code,
21                  SecurityDescriptor.SECURE_NO_SECURITY);
22
23  worker[1] = new DeployableFarmRemoteContructor<Integer, Integer>(
24                  "Farm1",
25                  "axth2.cli.di.unipi.it",
26                  new String[] {  "axth3.cli.di.unipi.it",
27                                  "axth4.cli.di.unipi.it",
28                                  "axth5.cli.di.unipi.it"},
29                  2,
30                  new String[] {  "FARM_MULTICONCERN_MANAGER",
31                                  "FARM_MULTICONCERN_SECURITY_MANAGER"},
32                  farmWorker,
33                  SecurityDescriptor.SECURE_OUT);
34
35  worker[2] = new DeployableSeqRemoteConstructor<Integer, Integer>(
36                  "Skeleton3",
37                  "axth6.cli.di.unipi.it",
38                  "Temp",
39                  null,
40                  code,
41                  SecurityDescriptor.SECURE_NO_SECURITY);
42
43  pipe = new DeployablePipelineRemoteConstructor<Integer, Integer>(
44                  "Pipe",
45                  "axth7.cli.di.unipi.it",null,
46                  worker,
47                  SecurityDescriptor.SECURE_NO_SECURITY);
48
49  remoteBS = (AutonomicControllerInterface) pipe.deploy();
50
51  con.send(new Integer(2));
52  Integer temp = con.receive();
```

This example only illustrates the usage of the LIBERO component constructors since we only specify how detail the system building. The actual component deployment is done when the `deploy()` method that is called on the last constructor (the `pipe` object, Listing 3 line 49).

In order to send data to the system a connection component is created with global ID `Temp`. This component is set to have as a destination the Pipeline BS and receives data from the last stage of the pipeline. In this naif example, we initiate the computation by sending the data to the pipeline and we wait for a response with a blocking call to `con.receive()` on Listing 3 line 52.