ADVANCES IN AUTONOMIC COMPONENTS & SERVICES*

Marco Aldinucci, Marco Danelutto, Giorgio Zoppi
Dept. Computer Science – Univ. Pisa
{aldinuc,marcod,zoppi}@di.unipi.it

Peter Kilpatrick
Dept. Computer Science – Queen's Univ. Belfast
p.kilpatrick@qub.ac.uk

Abstract Hierarchical autonomic management of structured grid applications can be efficiently implemented using production rule engines. Rules of the form "precondition → action" can be used to model the behaviour of autonomic managers in such a way that the autonomic control and the application management strategy are kept separate. This simplifies the manager design as well as user customization of autonomic manager policies.

We briefly introduce rule-based autonomic managers. Then we discuss an implementation of a GCM-like behavioural skeleton – a composite component modelling a standard parallelism exploitation pattern with its own autonomic controller – in SCA/Tuscany. The implementation uses the JBoss rules engine to provide an autonomic behavioural skeleton component and services to expose the component functionality to the standard service framework. Performance results are discussed and finally similarities and differences with respect to the ProActive-based reference GCM implementation are discussed briefly.

Keywords: Behavioural skeletons, autonomic computing, Service Component Architecture, task farm.

*This research is carried out under the FP6 Network of Excellence CoreGRID and the FP6 GridCOMP project funded by the European Commission (Contract IST-2002-004265 and FP6-034442).

Proc. of the CoreGRID symposium, Springer, Aug 2008. To appear.

1. Introduction

Autonomic management is increasingly attracting attention as a means of handling the non-functional aspects of grid applications. Several research groups are investigating various ways to associate adaptive behaviour with distributed/grid programs [15, 19, 10, 18, 9].

Within the CoreGRID Programming Model Institute a component based grid programming model is being developed (the Grid Component Model, GCM) [12] which introduces the possibility of associating autonomic managers with grid application components. GCM allows hierarchical composition of components. This means that composite components can be perceived by the users as normal, primitive components. Thus GCM system designers can capitalize on composition to provide grid application programmers with composite components that encapsulate common Grid programming patterns such as pipes, farms, etc. [13]. Then, application programmers can simply use appropriately parameterized instances of these composite components to implement complete, efficient grid applications that exploit these patterns or nested arrangements of them.

Autonomic managers have been introduced into GCM to take care of performance concerns of composite components without requiring explicit/significant application programmer involvement [12]. The combination of well-known grid/distributed programming patterns together with an autonomic manager taking care of the pattern performance is represented by the concept of a *behavioural skeleton* [4–5].

Autonomic management of typical grid programming patterns is a complex activity *per se*. It requires the ability to monitor composite pattern execution, suitable policies capable of handling "irregular" executions as perceived via the monitoring activity and, last but not least, suitable mechanisms to implement the corrective actions described within the policies and triggered in response to monitoring of irregular execution activity.

Further complexity arises when the autonomic manager activities are not considered in isolation but as a part of more global autonomic management activities as happens when composite patterns are nested to model increasingly complex grid applications. In this latter case, complex autonomic management policies and strategies have to be identified that allow combination of the actions performed by the single autonomic managers in the application in such a way as to implement a more general, application-wide autonomic strategy.

In this work we build on previous work concerning behavioural skeletons and hierarchical autonomic management in grid applications [6] and we define a general principle that allows combination of autonomic behaviour of different, nested behavioural skeletons in a single grid application (Sec. 2). Then we discuss a prototype implementation *de facto* demonstrating the feasibility of the

approach. The prototype implementation is built on top of the Tuscany/SCA (Service Component Architecture) [8] infrastructure rather than on top of the existing reference implementation of GCM under development within the GridCOMP STREP project (Sec. 3). Finally, we outline how the whole methodology based on autonomic management within behavioural skeletons can be exported to plain service users. The result is a seamless integration of GCM behavioural skeleton concepts into the SOA/WS framework (Sec. 4).

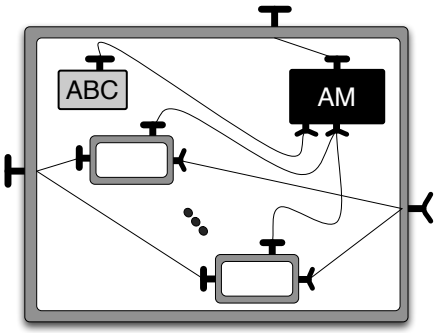

Figure 1. Sample behavioural skeleton structure.

2. Autonomic management using rules

We introduced autonomic managers enforcing user provided performance contracts within a single behavioural skeleton in [4–5]. The performance contracts enforced by behavioural skeletons currently include only service time (basically the inverse of throughput) and constant parallelism degree (i.e. the ability to keep constant the number of resources used to implement the application, in the presence of (temporary or permanent) resource faults).

In this section we discuss *hierarchical* management of grid applications. In particular, we make the assumptions used in [6] to discuss autonomic management of grid applications, that is:

- We assume that grid applications are developed using GCM components.
- We assume that behavioural skeletons modelling common parallel patterns are available. A behavioural skeleton is a parametric composite component modelling a commonly useful, efficient parallel grid pattern under the control of an internal autonomic manager responsible for guaranteeing a user-provided performance contract. Figure 1 outlines the structure of a behavioural skeleton. In the behavioural skeleton ABC is the *Autonomic Behavioural Controller*, the passive component responsible for providing probes for inspecting the status of a behavioural skeleton

and mechanisms to implement autonomic actions. AM is the *Autonomic Manager*, the active component responsible for behavioural skeleton autonomic management (see [5] for a fuller description of both ABC and AM in behavioural skeletons). The inner components are the ones managed by the behavioural skeleton, in this case according to a functional replication/data parallel pattern.

- We assume that behavioural skeletons may be arbitrarily nested and therefore that a grid application can be abstracted as a skeleton tree. Each node in the tree is labelled with the pattern represented by the corresponding behavioural skeleton and each node has a number of descendant nodes representing the functional parameters of the behavioural skeleton.

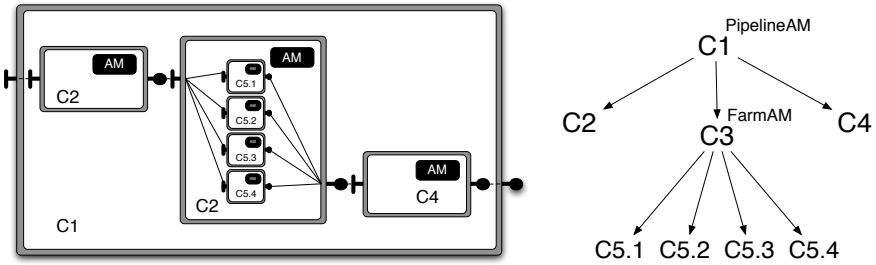

Figure 2. Sample application structure: component view (left) and skeleton view (right)

As an example, Fig. 2 depicts a grid application built as a three-stage pipeline. The first stage pre-processes the input and the last post-processes the results. The inner stage takes as input the output of the first stage and computes its result *in parallel* as the programmer recognizes that this is a highly demanding computation.

Autonomic managers in the behavioural skeleton components of the application enforce a performance contract that can either be provided by the user or agreed to by interacting AMs without any user intervention. For instance, in the sample application of Fig. 2 the contract of C1, the top level pipeline behavioural skeleton, is provided by the user, while the contracts of C2, C3 and C4 are derived from the contract of C1 and sent to the corresponding managers by the manager of C1.

We summarize the autonomic contract management activities in our nested behavioural skeleton context by the following abstract perspective, which was partially developed in [6].

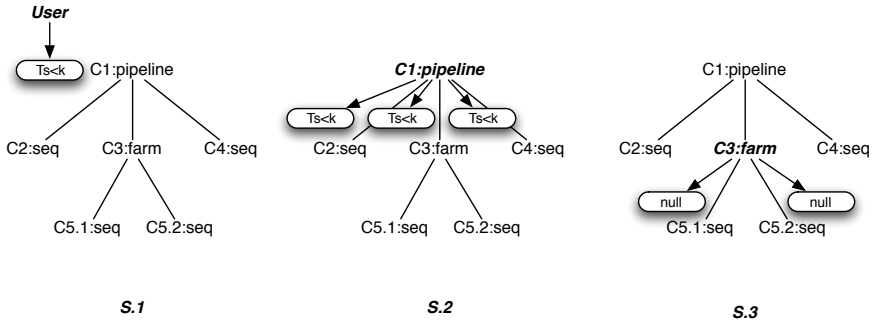

Figure 3. Contract propagation

2.1 Abstract perspective

The application is represented by means of a skeleton tree, such as the one of Fig. 2 right. The top level contract is provided by the application user, using the appropriate non-functional interfaces/ports. Contracts of managers in inner nodes come from parent nodes. The *propagation* of contracts from root to leaves happens either at compile time or at run time, depending on when the user provides the top level contract. In general, this is a non-trivial process. Sub-contracts for the inner component managers can be determined from the contract of the top level component manager only due to the fact that we are considering behavioural *skeletons*, that is, we are limiting the form of parallelism exploited within the top level component to a well known pattern. Figure 3 shows how a pipeline manager propagates contracts to the inner stage managers (steps S.1 and S.2). In this case, the same contract of the pipeline manager is passed to the stage managers, as pipeline service time is given by the maximum of the stage component service times ($T_{S_{pipeline}} = max\{T_{S_{stage_1}}, \ldots, T_{S_{stage_n}}\}$). In the case of task farms, contract propagation is quite different. Farm service time is given by the aggregate service time of the inner worker components. In particular, in a farm with n_w workers, the service time can be approximated as $T_{S_{farm}} = (\sum_{i=1}^{n_w} T_{S_{worker_i}})/{n_w}^2$. Therefore a farm manager propagates to the worker components a null contract, basically stating worker components should do their best to exploit the available resources and then the farm manager will take care of ensuring the farm contract by varying the number of inner worker components (see Fig. 3, step S.3).

Once the application has been started, and the contracts have been propagated to the inner managers, the autonomic managers in the nodes determine whether the current contract is satisfied and, if it is not, they start an autonomic corrective action aimed at enforcing once again contract satisfaction. In this abstract perspective, verification of a contract basically requires three steps.

Step 1 The inner component autonomic managers are queried and the status of their contracts is obtained. Each inner manager provides both a boolean value (contract satisfied or not satisfied) together with a set of parameters concerning its monitoring status (e.g. the measures used to evaluate the contract, as provided by the component ABC). In this phase, the top level manager behaves as a master with respect to the slave inner components in the context of a *monitor* activity.

Step 2 The contract of the behavioural skeleton is evaluated making use of the values given by the inner managers (*monitor*). These values are periodically used to instantiate variables in the terms of a formula that represent the QoS contract (currently a first order logic formula). If the formula evaluates to `false` the contract is considered broken; otherwise it is considered satisfied.

Step 3 If the local contract is no longer satisfied, either a local action is taken aimed at reestablishing the existing contract or a failure is reported to the manager of the parent behavioural skeleton in the skeleton tree. The execution of a local action may involve distribution of new contracts to the inner components, as well as changing the current configuration of the behavioural skeleton component. The choice between performing local actions and reporting failure is driven by the *rules* embedded in the manager. These rules represent the AM knowledge base. Each rule is composed of a *precondition* (if satisfied the rule can be used), an *action* (if the rule is used the action states what steps have to be performed), a *cost* (the overhead incurred if the rule is applied) and finally an *expected benefit* (the benefit, in terms of the contract, that the AM can expect following rule application) [6].

The rules considered in the Step 3 above are related to the performance contract formulas. If the contract is violated, the formula representing the contract itself can be *analysed* to derive (one or more) assignments of the variables that may satisfy the formula and therefore the contract. Only variables that are likely to be altered due to a reconfiguration *plan* are considered in this process, and the plans suitably altering these variable values are considered for execution. The execution of a reconfiguration plan by a manager may consist in changing the assembly of inner components (e.g. adding a replica of a component) and/or enforcing a new contract on some inner component (via its manager). This corresponds to the inclusion in the AM knowledge base of a rule that has as a precondition the formula modelling plan feasibility and as an action the plan itself.

In the event that no plan is likely to induce the satisfaction of the formula at some point in the future, a broken contact event has to be propagated to the parent manager (to the user, if the top level AM is considered). This corresponds

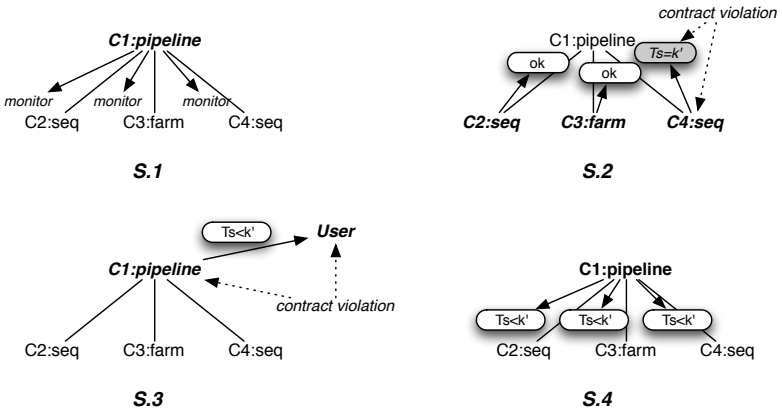

Figure 4. Sample inter-manager interactions: scenario 1

to the inclusion in the AM knowledge base of a (lowest priority) rule that has no precondition and has as action the report of the contract violation to the upper level manager.

Notice that in the general case the co-ordination of management plans is a difficult activity for several reasons. On the one hand, the satisfaction of a contract cannot be always guaranteed by the satisfaction of all the contracts of the inner components (for example, the interaction among components is usually not captured by any of the inner contracts in isolation, and the expected effect of reconfiguration plans is a forecast and its precision may be very coarse). On the other hand, starting from a contract it is not always easy to split it into sub-contracts (to be propagated to the inner components) in such a way that satisfaction of sub-contracts is likely to satisfy the contract (in this regard we are currently investigating an alternative logic that may easily support the projection of contract formulas into sub-contract formulas [7]). The proposed approach aims to ameliorate both problems via the behavioural skeleton concept since in these parametric components the general structure of contracts (formulas and plans) is pre-defined (up to parameterization).

2.2 Managers at work: sample scenarios

To illustrate how the whole process above works, consider again the application of Fig. 2 and let us assume that the user has provided a service time contract stating that service time should be less that k msecs ($T_{S_{application}} = T_{S_{pipeline}} < k$) and that contract propagation has already been performed as shown in Fig. 3. Figures 4 and 5 illustrate some typical contract management scenarios within related autonomic managers.

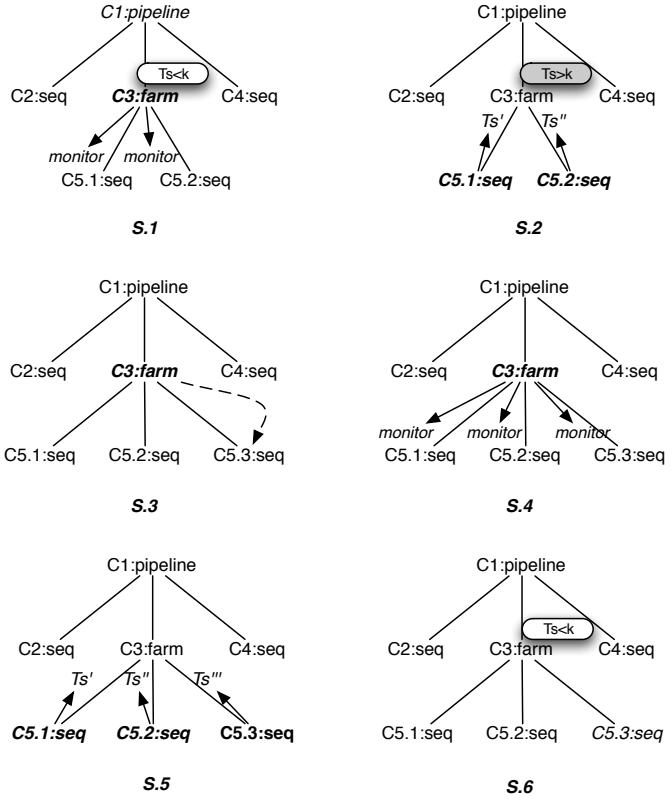

Figure 5. Sample inter-manager interactions: scenario 2

In the first scenario (Fig. 4) the pipeline manager requests from the inner components the status of their contracts (this is the Step 1 in the abstract view above, *S.1* in the figure) and receives back two "contract satisfied" and one "contract violation" responses (*S.2*). The contract violation ($T_S = k'$ with $k' > k$) is raised by a sequential component manager (the manager of C4) that has no way to improve the performance (service time) of the controlled component. The pipeline manager has no means to ensure the user supplied contract and therefore reports a contract violation to the user console (*S.3*). If some "best effort" behaviour is requested by default, the pipeline manager may propagate a new, less strict contract ($T_S < k'$) to the inner stages, which possibly results in the release of resources previously required by the inner stages running with $T_S < k$.

In the second scenario (Fig. 5) the farm manager has a $T_S < k$ contract and requests contract values (service times) from the inner worker components (*S.1*). It receives two values that together make its contract false ($T_S = (T'_S + T''_S)/4 >$

k (*S.2*)). A rule with precondition $T_{S_{monitored}} > T_{S_{contract}}$ and action "add a fresh worker component instance" is applied (*S.3*). After the time needed to implement the rule (as estimated by the farm manager), the contracts of the inner components are monitored again (*S.4*, *S.5*) and this time the contract turns out to be satisfied (*S.6*).

3. Prototype rule based autonomic management

A reference implementation of GCM is being developed on top of ProActive middleware [17] in the framework of the GridCOMP project [16]. Here, behavioural skeletons and autonomic managers within behavioural skeletons are implemented as described above. To date, however, the reference implementation of GCM does not explicitly use rules as described in Sec. 2. Rather, plain Java code is used within the manager to implement the rule concept. This was mainly due to implementation issues and the incremental nature of the design and implementation of the behavioural skeleton concept.

Recently, we implemented a single behavioural skeleton (one modelling the embarrassingly parallel computation pattern) on top of the Tuscany [3] SCA framework [1]. We wished to implement the behavioural skeleton concept as conceived in GCM without the restrictions and constraints of the ProActive-based reference implementation. At the same time, we wished to export GCM concepts to the service world and investigate the feasibility of implementing them on top of services. Tuscany looked like a viable proposition, being an open source component platform using state of the art, service based mechanisms.

The general design of the SCA implementation of the GCM task farm behavioural skeleton was introduced in [20, 14]; in the current work we address in more detail the implementation of the rule-based autonomic manager. SCA allows programmers to make use of the component concept in the service framework. SCA components are perceived as plain services from the user viewpoint. We therefore developed an SCA service (the *WorkPoolService*) implementing a task farm behavioural skeleton according to the GCM specification as introduced in Sec. 2. The Workpool Service is outlined in Fig. 8. Two basic sets of services are provided: to submit tasks to be computed (this is the service functional interface, `WorkpoolService` in the figure) and to interact with the *WorkpoolService* manager (this is the non-functional one, `WorkpoolManagerService` in the figure).

The autonomic manager (*WorkpoolManager Component*) uses JBoss Rules, a "framework that provides an open source and standards-based business rules engine and business rules management system (BRMS) for easy business policy access, change, and management" [2]. The JBoss engine supports dynamic addition and removal of rules. The Drools Rule Language (DRL) implemented in JBoss uses Java to express field constraints, functions, and consequences in

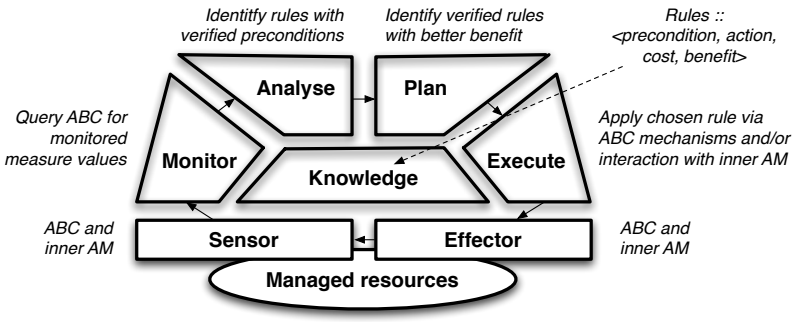

Figure 6. Autonomic cycle revisited

rules. In particular, Java beans are used to implement the getter methods needed to access variable values and the methods implementing functions (actions) used in the rules. A JBoss rule can be defined as a rule having a name, a condition enabling its application and an action to be taken if that condition holds. An example of a JBoss rule is the following:

```
rule "AdaptUsageFactor"
  when   $workerBean:WorkpoolBean(serviceTime > 0.25)
  then   $workerBean.addWorkerToLeastUsedNode();
end
```

The rule named "AdaptUsageFactor" can be used `when` the condition stating that the managed component `serviceTime` is more than 0.25 holds and, in this case, an `addWorkerToLeastUsedNode` is performed.

JBoss rules rely on the existence of a Java Bean (the one referenced by `$workerBean` in the example) to access the required values (e.g. the `service Time` instance variable of the bean) and `then` to implement the rule action (e.g. to invoke the `addWorkerToLeastUsedNode()` method on the same bean). To retain the possibility of using fully-fledged JBoss rules, we implemented the `WorkpoolManager` component in such a way that it uses an internal bean to support JBoss rules. The bean instance fields are set up periodically through the bean setter methods by the `WorkpoolManager`. In turn, the `WorkpoolManager` retrieves the relevant data through the methods exposed via the `WorkpoolService` interface. With respect to the GCM model (as outlined in Fig. 1), these methods (services) correspond to the non-functional, passive interface implemented by the ABC controller.

Our *WorkpoolManager Component* runs the JBoss rule engine. The rules (such as the one given above) constitute the manager knowledge base (see Fig. 6) and can be dynamically configured (added, deleted) through the `Workpool ManagerService` non-functional interface. For example, rules in component C3 of Fig 2 will be initially configured to include the sample rule shown above

if the user contract requires from C1 a service time of at most 0.25 secs. If the C1 manager, while testing for contract integrity, discovers that the service time provided by the task farm is higher than both T_{s_1} and T_{s_3} (the service times of C1 and C3, respectively) it should interact with the C2 manager and send it a new `AdaptUsageFactor` differing only in the when clause

```
when $workerBean:WorkpoolBean(serviceTime > max(TS1,TS2))
```

that will eventually substitute the old `AdaptUsageFactor` rule.

To date we have experimented only with the SCA behavioural skeleton implementation alone (i.e. not in a behavioural skeleton nesting). However, the mechanism discussed above enables manager interaction via the submission of new contracts, in the form of rules. Submission of new rules can take place either during `Workpool` startup, to implement the initial propagation of the user-supplied top level contract, or at run time, during autonomic management actions reconfiguring the inner components of the behavioural skeleton. The mechanism has been proven effective by running a set of experiments that separately measured the scalability of the Tuscany/SCA task farm behavioural skeleton, and the overhead introduced by a typical, single reconfiguration enacted by its autonomic manager. We measured scalability of synthetic applications with variable computational grain. The computational grain $g = T_{seq}/T_{comm\_in\_out}$ is the ratio of the time spent to compute a task on the remote resource (T_{seq}), to the time spent to deliver the input data to the remote node plus the time spent to retrieve the results from the remote node ($T_{comm\_in\_out}$). The definition of scalability, $S(n)$, is the classical one: $S(n) = T(1)/T(n)$, where $T(n)$ represents the completion time of the application run with parallelism degree equal to n. Typical results are shown in Fig. 7 (left). Considering the high overhead in serializing (deserializing) service parameters with SOAP XML (we used no optimization), this represents a fairly good result.

Concerning the overhead related to reconfiguration of the behavioural skeleton, we measured the time spent in computing a set of 1K tasks, including a forced reconfiguration that doubled the number of farm workers ($4 \rightarrow 8$) when a given number of tasks had already been computed. The results are shown in Fig. 7 (right). The Exp1 (Exp2) line refers to an experiment where the workers were doubled after half (quarter) of the tasks were computed. In both cases the overhead involved is negligible, considering it includes both the time spent to activate (upon a timer) the JBoss rule engine and the time spent to perform the "add worker" rule four times.

4. Behavioural skeletons in SCA and interoperability

As stated at the beginning of Sec. 3, our implementation of GCM behavioural skeletons on top of SCA was also aimed at demonstrating the suitability of SCA

	4 PE	8 PE	16 PE
g = 10	0.96	0.89	0.6
g = 24	0.98	0.97	0.77
g = 40	0.99	0.97	0.87

	Measured	Estimated	ε
Exp 1	255.07 s	252.07 s	0.99
Exp 2	217.33 s	209.76 s	0.97

Figure 7. Scalability (left) and reconfiguration (right) efficiency results.

to support GCM concepts and the interoperability we were able to achieve with the wider (i.e. beyond the GCM and grid community) service world.

SCA offers most of the mechanisms needed to implement a GCM behavioural skeleton. One facility missing is the means to change composite component assemblies at run time via XML composite component descriptors. For instance, when a new worker component has to be added to the `WorkpoolService`, we cannot simply produce a new composite descriptor to tell the framework the composite assembly has changed. Consequently, we implemented a component to deal with this kind of assembly change. The component provides means to instantiate a new (worker) component and to create the appropriate `connections` as defined by the schema of Fig. 8. The component uses the Tuscany API which, in turn, provides the mechanisms required to support new component integration with (as well as old component removal from) a component assembly. The SCA implementation of the task farm behavioural skeleton directly mirrors the GCM/ProActive implementation. The GCM/ProActive ABC is implemented via operations exported by the `Workpool` Service and the AM is implemented by the SCA component `WorkpoolManager` Service. All the components in Fig. 8 (the `WorkpoolService`, the `WorkpoolManager`, the `WorkerManagerNode` and the `WorkerService`) are exposed as services. They can be accessed through the automatically generated WSDL as plain services and, more importantly, they can be re-used to implement different behavioural skeletons in exactly the same way that the ABC and AM components may be re-used within the GCM/ProActive framework to implement other behavioural skeletons.

The overall design of the Workpool service (and of the associated support mechanisms) has been judged interesting by the Tuscany developers and our code has been included in the SCA `svn` as a Tuscany sample application.

Concerning interoperability, we verified that accessing a behavioural skeleton is as easy as accessing any other type of service on the network, as expected. Fig. 9 sketches the code needed to submit tasks to the `WorkpoolService` behavioural skeleton. The first part of the code (on the left) is that needed to set up a reference to the service (`args[0]` is the url of the service WSDL file). Here a

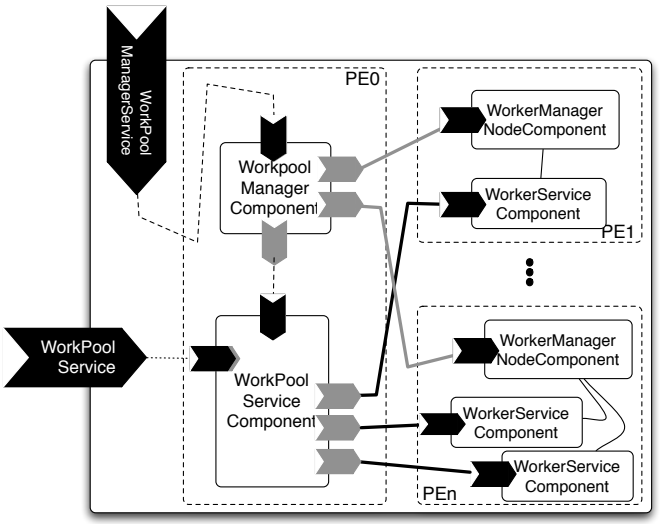

Figure 8. Workpool Service structure

service that will be invoked to post-process the results produced is passed to the `WorkpoolService`. The second part of the code (on the right) is that needed to submit the single task (in a `Job`) to the `WorkpoolService`. This code is of the same form as that required to access any other type of service from a Java program. Normal service application programmers require no additional effort to benefit from the advanced management supported in the `WorkpoolService`. Thus our implementation satisfies the requirement to *propagate the concept with minimal disruption* as stated by Murray Cole in his skeleton "manifesto" [11]. Service users may have the benefit of a fully-fledged autonomic implementation of embarrassingly parallel computations within a single service incorporating the best of the relevant GCM methodology and concepts.

5. Conclusions

We introduced rule-based autonomic management techniques for structured grid applications implemented using GCM Behavioural Skeletons. The general mechanism of rule exploitation for performance contract monitoring together with a significant sample case have been discussed. We then described a prototype implementation in SCA/Tuscany. We presented preliminary experimental results demonstrating the feasibility of the approach as well as the portability of GCM autonomic management aspects into the Service framework. The prototype implementation makes available a GCM task farm behavioural skeleton to service application programmers and thus helps broaden the applicability of

```
...
// creates the workpool service stub
WorkpoolServiceStub wstub =
  new WorkpoolServiceStub(workpoolServiceWSDLuri);
// sets up services processing the results computed
WorkpoolServiceStub.AddTrigger sink = new
    WorkpoolServiceStub.AddTrigger();
WorkpoolServiceStub.CallableReferenceImpl callableReference =
    new WorkpoolServiceStub.CallableReferenceImpl();
WorkpoolServiceStub.EndpointReference endpoint =
     new WorkpoolServiceStub.EndpointReference();
endpoint.setURI(resultPostProcessServiceURI);
callableReference.setEndpointReference(endpoint);
sink.setParam0(callableReference);
wstub.addTrigger(sink);
// create a Job
MyJob j = new MyJob();
// set up serialization stuff
Serializer s = new Serializer();
OMElement element = s.serialize(j);
// create a submit request
WorkpoolServiceStub.Submit submit= new
    WorkpoolServiceStub.Submit();
// create the task
WorkpoolServiceStub.Job task= new
    WorkpoolServiceStub.Job();
// set up task and submit
task.setData(element);
submit.setParam0(task);
wstub.submit(submit)
...
```

Figure 9. Sample client code for the `WorkpoolService`

CoreGRID results. As the intended target audience of the prototype is the service community, this also makes a bridge between the component and service worlds. The design of the prototype, fully exploiting component technology, allows reuse of its different parts to implement different behavioural skeletons. We are currently integrating the rule based implementation of behavioural skeletons into the GCM reference implementation being developed on top of ProActive in the GridCOMP project.

References

[1] Service component architecture, 2007. `http://www.ibm.com/developerworks/library/specification/ws-sca/`.

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

[3] Tuscany home page, 2008. `http://incubator.apache.org/tuscany/`.

[4] M. Aldinucci, S. Campa, M. Danelutto, P. Dazzi, P. Kilpatrick, D. Laforenza, and N. 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.

[5] M. Aldinucci, S. Campa, M. Danelutto, M. Vanneschi, P. Dazzi, D. Laforenza, N. Tonellotto, and P. Kilpatrick. Behavioural skeletons in GCM: autonomic management of grid components. In *Proc. of Intl. Euromicro PDP 2008: Parallel Distributed and network-based Processing*, Toulouse, France, pages 54-63, Feb. 2008. IEEE.

[6] M. Aldinucci, M. Danelutto, and P. Kilpatrick. Towards hierarchical management of autonomic components: a case study. Technical Report TR-0127, CoreGRID, 2008. Available at `http://www.coregrid.net/mambo/images/stories/TechnicalReports/tr-0127.pdf`.

[7] S. Bistarelli, U. Montanari, F. Rossi, Semiring-Based Constraint Logic Programming: Syntax and Semantics, ACM TOPLAS, Vol. 23, 2001

[8] M. Beisiegel, H. Blohm, D. Booz *et al.* Service Component Architecture Building Systems using a Service Oriented Architecture, A Joint Whitepaper by BEA, IBM, Interface21, IONA, Oracle, SAP, Siebel, Sybase. 2000, available at `http://www.iona.com/devcenter/sca/SCA_White_Paper1_09.pdf`

[9] P. Boinot, R. Marlet, J. Noyé, G. Muller, and C. Cosell. A declarative approach for designing and developing adaptive components. In *Proc. of the 15th Intl. Conference on Automated Software Engineering*, pages 111–119. IEEE, 2000.

[10] J. Buisson, F. André, and J.-L. Pazat. Afpac: Enforcing consistency during the adaptation of a parallel component. *Scalable Computing: Practice and Experience*, 7(3):83–95, 2006

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

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

[13] CoreGRID NoE deliverable series, Prog. Model Institute. *D.PM.11 – GCM experience: inside the single component and beyond components*, Feb. 2008. `http://www.coregrid.net/mambo/content/view/428/292/`.

[14] M. Danelutto and G. Zoppi. Behavioural skeletons meeting Services. In *Proceedings of PAPP'08*. Springer Verlag, LNCS No. 5101, pages 146–153, June 2008. Krakow, Poland.

[15] H. González-Vélez. Self-adaptive skeletal task farm for computational grids. *Parallel Comput.*, 32(7):479–490, 2006.

[16] GridCOMP. GridCOMP web page, 2007. `http://gridcomp.ercim.org`.

[17] ProActive home page, 2006. `http://www-sop.inria.fr/oasis/proactive/`.

[18] S. S. Vadhiyar and J. J. Dongarra. Self adaptivity in grid computing: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(2-4):235–257, 2005.

[19] G. Wrzesinska, J. Maassen, and H. E. Bal. Self-adaptive applications on the grid. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 121–129, New York, NY, USA, 2007. ACM.

[20] G. Zoppi. Componenti Avanzati GCM/SCA, 2008. Dept. Computer Science, Univ. of Pisa. 2nd level graduation thesis, in Italian. `http://etd.adm.unipi.it/theses/available/etd-01302008-103715/`