Behavioural Skeletons Meeting Services*

M. Danelutto[1,2] and G. Zoppi[1]

[1] Dept. Computer Science – Univ. Pisa
[2] CoreGRID Programming Model Institute

Abstract. Behavioural skeletons have been introduced as a suitable way to model autonomic management of parallel, distributed (grid) applications. A behavioural skeleton is basically a skeleton with an associated autonomic manager taking care of non-functional issues related to skeleton implementation. Here we discuss an implementation of a task farm behavioural skeleton exploiting SCA, the Service Component Architecture recently introduced by IBM. This implementation is meant to provide plain service/SCA users some efficient skeleton modelling common parallel application pattern and also to investigate the advantages and the problems relative to skeletons in the service world. Experimental results are eventually discussed.

1 Introduction

Algorithmic skeleton concepts have been around since the initial works by Murray Cole in late '80 [9]. They initially spread through the parallel computing community and they fertilized some independent research activities in different research groups [12,8]. Several distinct systems have been developed [11,15,18,19] exploiting different implementation technologies that have been used as proof of concept of user friendly, efficient, completely transparent parallel programming models basically targeting parallel/distributed architectures such as workstation clusters and networks.

More recently algorithmic skeleton concepts moved to the *grid* scenario and were used to program several different kind of super/meta/skeleton components modelling common grid programming paradigms [13,7,4]. Components embedding of skeletons somehow filled the gap among skeletons concepts and software engineering concepts. As a matter of fact, component technology allowed to embed skeletons in well know programming paradigms (the components) and to hide non-relevant implementation parameters to final skeleton users (by exploiting hierarchical component composition facilities). Eventually this allowed more and more high level programming abstraction to be provided to application programmers. In component frameworks, skeletons are usually implemented through composite components and basically provide the very same programming pattern that classical, non component skeleton programming environments

* 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).

M. Bubak et al. (Eds.): ICCS 2008, Part I, LNCS 5101, pp. 146–153, 2008.

usually provide to the application programmers. Skeleton code parameters were provided as components, allowing hierarchical and incremental program development. Component technology simplified somehow code deployment on remote processing elements leveraging onto component framework facilities rather then requiring explicit and consistent programming efforts in the design and implementation of the skeleton compiler and run time tools.

After successfully migrating to the grid scenario via component technology a further step was made: skeletons were used to combine powerful parallel application pattern abstraction with typical grid related autonomic management features. Behavioural skeletons were thus introduced in 2007 [5,6] in the framework of the CoreGRID and GridCOMP projects[1]. A behavioural skeleton is basically a skeleton integrated with an "autonomic manager" item taking care of all the non-functional aspects related to skeleton implementation, such as performance optimization, fault tolerance and security. The autonomic managers, in this case, can be understood as the *locus* where well-known self-optimization and self-healing autonomic [14] features are implemented.

In the meanwhile, grid programming environments evolved more and more through (Web) Services and web services [20] become a de facto standard in several related scenarios: grid programming, enterprise applications, software interoperability, incremental application development, etc.

In this paper we basically discuss the results achieved by implementing a specific GCM behavioural skeleton (the functional replication/task farm skeleton as described in [6]) on top of SCA (the Service Component Architecture introduced by IBM [1]). By this experiment we aimed to follow Cole's "manifesto" suggestion to *propagate the concept with minimal disruption* [10]. The task farm skeleton discussed in this works provides service application programmers with a very high level programming paradigm that can be easily used to program most of the typical *embarrassingly parallel* grid/distributed applications actually leaving to the component implementing the programming paradigm[2] the hard task to implement self optimization and self healing features. We also followed Cole's recommendation to *accommodate diversity* by providing user-friendly ways of extending and modifying autonomic management features and policies. Last but not least, implementation of behavioural skeletons on top of SCA allowed a comparison to be performed with the already existing similar implementation of behavioural skeletons on top of ProActive/Fractal [17] developed within GridCOMP [16].

The rest of the paper is structured as follows: in Sections 2 and 3 we briefly introduce behavioural skeletons and Service Component Architecture. Section 4 details current implementation of a task farm behavioural skeleton in SCA and Section 5 presents the results we achieved and compares SCA implementation with GridCOMP ProActive based one.

[1] The former being an FP6 NoE and the latter being an FP6 STREP project, both funded by EU.

[2] To its autonomic manager, actually.

2 Behavioural Skeletons

As stated in [5,6] behavioural skeletons model common, reusable patterns of parallel computations as traditional algorithmic skeletons do. In addition, however, they provide self management facilities taking care of several distinct non functional[3] issues related to skeleton implementation. Sample behavioural skeleton component is depicted in Fig. 1. This is a simplified version of a composite component modelling a task farm skeleton. The component only has three *provide* ports: two are functional and they are used to ask task computation (submit initial data and retrieve computation results) and to specialize the task farm component with a proper worker component. The other one is non functional and it is used to submit to the component a performance contract to be satisfied on the behalf of the task farm component user. This is all what the final user of the composite component perceives of the task farm component. However, internally the component hosts a `Manager` component and one or more `Worker` components. The worker component string actually computes the submitted tasks under the supervision of the manager. The manager, in turn, takes care of implementing the performance contract provided by the user according to some kind of best effort strategy.

We do not want to enter a complete description of behavioural skeletons implemented as components here. The interested reader may refer to [5,6] in case. What we want to point out is the kind of management provided by the autonomic manager embedded in the composite task farm behavioural skeleton component. The autonomic manager continuously monitors a set of execution parameters (e.g. service time of the task farm). In case a performance contract violation is observed, it figures out a strategy suitable to recover the situation, possibly looking up a set of predefined "repair" strategies. Once the strategy has been individuated, a plan is computed to apply the strategy with the minimum overhead and eventually the plan is executed. Plan execution possibly requires to temporary stop normal functional component activities (task computation) and to reconfigure the internal composite component components (e.g. adding one more worker to the current worker component string). In case no strategy can be found to repair the performance contract violation, the user is simply informed and nothing else happens. The computation prosecutes as initiated, fulfilling the best effort criteria we stated for the autonomic manager.

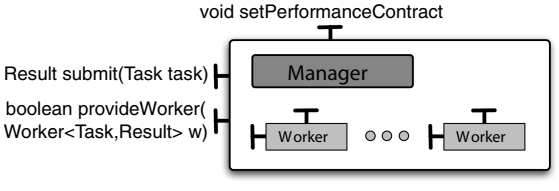

Fig. 1. Behavioural skeleton component (simplified)

[3] "Non functional" features are the features related to *how* a given computation is performed, opposite to "functional" features that are related to *what* it has to be computed.

3 SCA

Service Component Architecture [1] provides the user with a programming framework supporting application development based on Service Oriented Architecture. Applications programmers may build their application re-using existing services embedded in service components and specifying composite components through proper XML files. Eventually, SCA applications can be run on several distinct kinds of distributed platforms exploiting existing technologies such as web services or RMI. SCA bindings are provided for different programming languages in such a way programmers can use (among the others) Java and C++ code to program the primitive SCA components.

Typical SCA component assembly is the one of Fig. 2 (the Figure is taken from SCA documentation). In this case, the composite itself is specified using an XML file detailing all the internal components and wires as well as all the external use/provide interfaces. The composite can be run on the SCA runtime as well as on plain web services runtimes. Several alternative methods can be used to export and reference the component ports.

The big advantage of SCA framework relies in the fact that it provides very handy ways to build applications from existing services (which is the very same thing you can do using BPEL in other context) and allows the service composition to be (re-) used within applications as a primitive component.

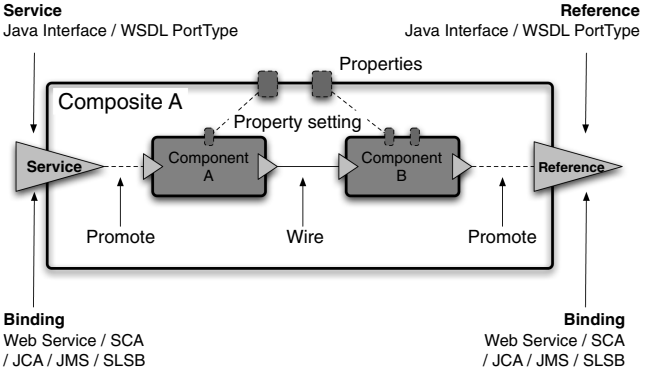

Fig. 2. SCA component assembly

In order to write a Java SCA program, programmers must basically perform the following steps. First, interfaces declaring the component interface should be provided. In the interface services (methods) as well as properties (instance variables) provided by the component are declared. Then Java code/classes implementing these interfaces should be provided. Eventually, XML files describing composition of components have to be written, declaring the component name, what the component exports (provide ports), uses (use ports) and how the component is implemented (java classes used). Once these steps have been performed, the program may be launched by instantiating a `SCADomain` and passing it as a parameter the XML `.composite` file hosting the component composition specification. At this point the component can be accessed by clients querying the SCADomain a reference to the component (this is achieved using the name provided in the XML file) and accessing the component exported features. Again,

the interested reader will find all the details relative to SCA on the web site hosting all the documentation [1]. We used here the Tuscany open source implementation of the SCA service component framework [3].

4 Implementation

Our implementation of the task farm behavioural skeleton was designed as shown in Figure 3. A `WorkpoolService` composite SCA component has been implemented and it is provided to the user that completely takes care of implementing a task farm, with respect to both functional and non-functional behaviour. The `WorkpoolService` component exposes interfaces (provides methods) to submit jobs (and this is a functional concern) as well as to start autonomic management and to submit rules affecting autonomic management behaviour (this is instead a non-functional concern). Autonomic management is implemented exploiting JBoss rules [2]. Each JBoss rule includes a precondition as well as the actions to be executed in case the precondition is satisfied. Both preconditions and actions use proper Java beans associated to the entities managed within the `WorkpoolService`. Methods of these beans may be invoked within a JBoss rule in order to achieve some (part of an) autonomic self-* behaviour. The JBoss rules associate to a SCA task farm behavioural skeleton component are activated only in case the user explicitly asks to start autonomic control management. Some predefined rules, such as increase the parallelism degree in case the task pool service time happens to be larger than user defined service time contract, are predefined in the task farm component. Other rules can be defined on the fly by the task farm component user and inserted via its non-functional interfaces.

The `TaskFarmManager` component inside the `WorkpoolService` actually takes care of tasks submit requests and uses one of the `WorkerManager` components to execute the submitted task, in such a way parallel execution of the submitted task *stream* is achieved. Each one of the `WorkerManager`s take care of the `Worker` components allocated on the same resource (processing element) used to run the `WorkerManager`. In turn, `Worker` components in the resource are allocated instantiating copies of the `Worker` components provided by the user through the `WorkpoolService` functional interface. Each `Worker` is able to compute a single submitted task at a time. The `WorkerManager` component provides a functional `submitTask` interface as well as non functional `addWorker` and `deleteWorker`

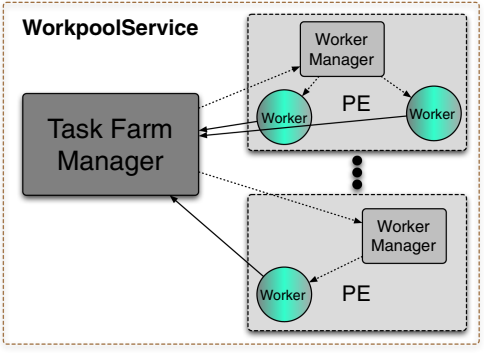

Fig. 3. WorkpoolService skeleton

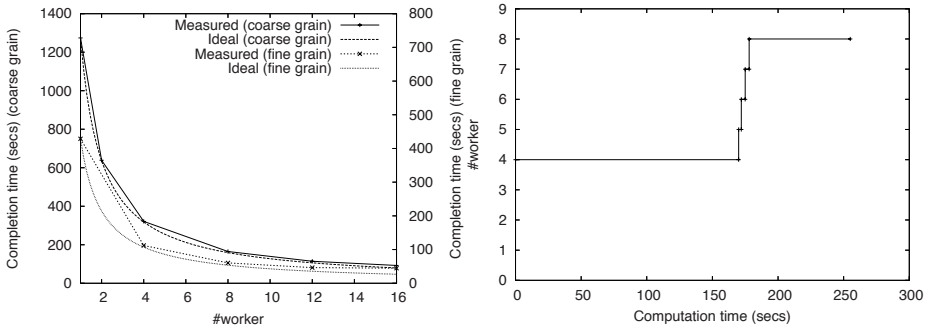

Fig. 4. Scalability (left) and effect of dynamic management (right) in task farm behavioural skeleton

interfaces that can be used, upon `WorkpoolService` requests, to allocate and deallocate computing components on the resource. The `WorkpoolService` basically includes the autonomic management of the task farm behavioural skeleton as described above.

5 Experimental Results

We run some experiments on a cluster of Linux workstations interconnected through a Fast Ethernet network to verify the functional features of our SCA task farm behavioural skeleton as well as the non-functional features of its autonomic management.

First of all, scalability has been measured. Fig. 4 left shows typical scalability curves we got when running coarse grain tasks thought the farm SCA behavioural skeleton. The grain of a task is the ratio between the time spent to compute the task and the time spent to deliver the input data and to retrieve the result data to and from the remote processing element actually computing that task. In this case the computation time refers to the overall time spent to compute 1K tasks on the farm SCA behavioural skeleton. It is not surprising that coarse grain tasks achieve almost perfect scalability as the tasks are independent (task farm implements an embarrassingly parallel computation pattern). In case the task computational grain is lower (O(10) ("fine grain" in the Figure) instead of O(100) ("coarse grain" in the Figure)) the task farm behavioural skeleton stops scaling at about 8 workers.

We then provided JBoss rules to adapt the task farm behavioural skeleton to varying performance achieved in the execution of the user tasks and we explicitly activated autonomic management. Fig. 4 right shows what happened when rules were used stating the service time of the task farm should be kept smaller than a given value and additional load was put on the resources used to compute the tasks. The autonomic manager inside the `WorkpoolService` detected a decrease in the service time and started increasing the parallelism degree until the required

service time was obtained again. In this case, additional load was deployed on the first four worker resources when about 500 tasks (out of 1K tasks) were computed. The task farm behavioural skeleton reacted autonomically and started new workers on four additional computing nodes, whose `WorkerManager` had no `Worker` allocated up to that moment. The whole thing happened without any explicit programmer intervention, but supplying the proper JBoss rule stating that in case the service time goes down under a given threshold, new workers should have been added, such as

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

After implementing the task farm behavioural skeleton on top of SCA Tuscany, we verified that all the necessary mechanisms are presents in SCA. The only thing we had to emulate programmatically was dynamic composite component modification, by generating proper `.composite` files that are then used to instantiate the new (modified) composite components.

6 Conclusions

We implemented a task farm behavioural skeleton such as the one described in [5,6] using SCA. A version of the very same behavioural skeleton on top of ProActive/GCM has already been implemented in the framework of the GridCOMP project [16]. We succeeded demonstrating that SCA can be usefully exploited to implement behavioural skeletons. Implementation scales and autonomic management is actually achieved exploiting proper JBoss rules modelling autonomic manager policies. In the ProActive/GCM implementation, however, autonomic behaviour of the task farm has to be programmed directly in Java, whereas in this case much more simpler JBoss rules can be used to implement the very same politics and strategies. Being the task farm behavioural skeleton component be implemented in a fully compliant SCA way, plain Web Services as well as Java component can be used to provide task farm workers and to implement `WorkpoolService` clients, in such a way interoperability is greatly enhanced (with respect to the ProActive/GCM implementation) and Cole recommendation to "propagate the concept with minimal disruption" was further enforced. Overall, we think this represents another step in the direction of permeating existing, state of the art parallel/distributed programming environments with the algorithmic skeleton concept. Autonomic bahaviour implementation through a separate manager within the task farm skeleton/component, in turn, further separates functional and non-functional concerns, as advocated in the behavioural skeleton framework.

The experience discussed in this work is a first step towards a full integration of behavioural skeletons in the service framework. The prototype presented here is being refined, to support further skeletons and to refine the mechanisms used to implement the autonomic rules.

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. Aldinucci, M., Bertolli, C., Campa, S., Coppola, M., Vanneschi, M., Veraldi, L., Zoccolo, C.: Self-Configuring and Self-Optimising Grid Components in the GCM model and their ASSIST implementation. In: Proceedings of HPC-GECO/Compframe Workshop, Paris, HPDC-15 associate workshop (2006)
5. Aldinucci, M., Campa, S., Danelutto, M., Dazzi, P., Kilpatrick, P., Laforenza, D., Tonellotto, N.: 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 (June 2007)
6. Aldinucci, M., Campa, S., Danelutto, M., Vanneschi, M., Kilpatrick, P., Dazzi, P., Laforenza, D., Tonellotto, N.: Behavioural skeletons in gcm: autonomic management of grid components. In: Proc. of Intl. Euromicro PDP 2008: Parallel Distributed and network-based Processing, Toulouse, France, IEEE, Los Alamitos (2008)
7. Aldinucci, M., Coppola, M., Danelutto, M., Tonellotto, N., Vanneschi, M., Zoccolo, C.: High level grid programming with ASSIST. Computational Methods in Science and Technology 12(1), 21–32 (2006)
8. Bacci, B., Danelutto, M., Orlando, S., Pelagatti, S., Vanneschi, M.: P^3L: A Structured High level programming language and its structured support. Concurrency Practice and Experience 7(3), 225–255 (1995)
9. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computations. Research Monographs in Parallel and Distributed Computing. Pitman (1989)
10. Cole, M.: Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. Parallel Computing 30(3), 389–406 (2004)
11. Cole, M., Benoit, A.: The Edinburgh Skeleton Library home page (2005), http://homepages.inf.ed.ac.uk/abenoit1/eSkel/
12. Darlington, J., Field, A.J., Harrison, P.G., Kelly, P.H.J., Sharp, D.W.N., Wu, Q., While, R.L.: Parallel Programming Using Skeleton Functions. In: Reeve, M., Bode, A., Wolf, G. (eds.) PARLE 1993. LNCS, vol. 694. Springer, Heidelberg (1993)
13. Gorlatch, S., Dünnweber, J.: From grid middleware to grid applications: Bridging the gap with HOCs. In: Getov, V., Laforenza, D., Reinefeld, A. (eds.) Future Generation Grids. Springer, Heidelberg (2006)
14. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer 36(1), 41–50 (2003)
15. Kuchen, H.: A skeleton library. In: Monien, B., Feldmann, R.L. (eds.) Euro-Par 2002. LNCS, vol. 2400, pp. 620–629. Springer, Heidelberg (2002)
16. GridComp: Effective Components for the Grids (2007), http://gridcomp.ercim.org/
17. ProActive home page (2007), http://www-sop.inria.fr/oasis/proactive/
18. Serot, J., Ginhac, D., Derutin, J.P.: SKiPPER: A Skeleton-Based Parallel Programming Environment for Real-Time Image Processing Applications. In: Malyshkin, V.E. (ed.) PaCT 1999. LNCS, vol. 1662, Springer, Heidelberg (1999)
19. Vanneschi, M.: PQE2000: HPC tools for industrial applications. IEEE Concurrency 6(4), 68–73 (1998)
20. W3C. Web services home page (2006), http://www.w3.org/2002/ws/