Co-design of Distributed Systems Using Skeleton and Autonomic Management Abstractions*

M. Aldinucci[1], M. Danelutto[1], and P. Kilpatrick[2]

[1] Dept. Computer Science, Univ. of Pisa, Italy
[2] Dept. Computer Science, Queen's Univ. of Belfast, UK

Abstract. We discuss how common problems arising with multi/many-core distributed architectures can be effectively handled through co-design of parallel/distributed programming abstractions and of autonomic management of non-functional concerns. In particular, we demonstrate how restricted parallel/distributed patterns (or skeletons) may be efficiently managed by rule-based autonomic managers. We discuss the basic principles underlying pattern+manager co-design, current implementations inspired by this approach and some results achieved with a proof-of-concept prototype.

Keywords: Algorithmic skeletons, design patterns, distributed programming abstractions, autonomic computing, grids, clouds, multi/many core.

1 Introduction

The development of parallel and distributed programs is recognized to be a challenging task. The management of the concurrent activities and communications together with the non-functional concerns, such as performance, security, fault tolerance, all require substantial efforts during both the design and implementation phases and in debugging, tuning and maintenance of the application.

The sustained evolution in parallel and distributed architectures makes application development even harder, as technological improvements and architectural model changes must be catered for. On the one hand the increasing prevalence of multi- and many-core systems necessitates the use of some kind of parameterisation of the code to support hundreds or even thousands of parallel activities, as it is inconceivable that a programmer may design, implement and manage hundreds or thousands of different activities. On the other hand, the emergence of first grid and now cloud architectures, with their inherent heterogeneity and dynamicity, has thrown into stark relief the burden of handling non-functional concerns.

Researchers in two distinct areas have tried separately to tackle these issues, but to date there is not a comprehensive methodology to attack the distributed

* This work has been partially supported by EU FP6 NoE CoreGRID, EU FP6 STREP GridCOMP and Italian FIRB Insyeme projects.

E. César et al. (Eds.): Euro-Par 2008 Workshops, LNCS 5415, pp. 403–414, 2009.

application problem in general. On the one hand, *algorithmic skeletons* provide programmers with higher-level abstractions that can be used as building blocks for complex parallel and distributed applications [6]. This addresses the need for structuring. On the other hand, *autonomic computing* has, with some success, provided means to manage some non-functional aspects important in parallel and distributed applications, such as those related to performance (self-configuration and self-optimization) and fault tolerance (self-healing) [13].

In this paper we present an approach which is based on combining algorithmic skeletons and autonomic computing. It advocates a structure/management co-design approach to system development. Skeleton and autonomic management abstractions are given, and the required interfaces allowing interaction between the two are discussed. Refinements to component-based and services-based implementations are then described briefly and results presented.

The proposed approach also provides an attractive separation of concerns between system and application programmers. System programmers have responsibility for providing suitable skeleton frameworks taking care of issues such as process communication, etc., *and* also for ensuring appropriate management of non-functional concerns. This frees the application programmer to focus on selecting a suitably parameterized skeleton and supplying the core functional code; and, for specifying non-functional concern requirements via, for example, some sort of service level agreement (SLA) (although this latter remains a considerable challenge and is currently only achievable to a modest degree).

The rest of the paper is as follows: Section 2 introduces abstractions for the basic skeleton and autonomic computing concepts, Section 3 proposes a methodology for the co-design of computation structure and autonomic management, Section 4 describes an implementation derived using this methodology and presents experimental results achieved within the GridCOMP project. Section 5 explores the challenge of multi-concern management and Section 6 concludes the paper.

2 Programming Abstractions

We introduce here two "generic" programming abstractions: one to capture the structure of a parallel/distributed application and one to deal with non-functional concern management. The co-design of these two abstraction will eventually lead to a much more powerful and effective programming abstraction, whose preliminary implementation and results are briefly outlined in Sec. 4.

2.1 Structuring Abstractions

Successful parallel and distributed applications usually implement some well-known and efficient parallel or distributed computation *design pattern* [16]. These patterns can be recognized as useful distributed programming abstractions to be implemented and optimized once and for all, and then provided to the application programmers, in such a way that the effort of developing efficient distributed applications is factorized across several similar application designs. A natural choice to provide such abstractions to the application programmer

is the *algorithmic skeleton* concept [7,6]. An algorithmic skeleton is a parametric, reusable, efficient implementation of a commonly used parallel/distributed computation pattern. The application programmer can pick up an algorithmic skeleton, instantiate it with suitable code and data parameters and obtain immediately a working application. Depending on the skeleton framework available, algorithmic skeletons can be (more or less) arbitrarily nested to obtain increasingly complex parallel and distributed applications.

Consider a classical skeleton, often used to model distributed computations running on classical distributed architectures, such as COW/NOWs and grids: the divide and conquer pattern. This pattern can be abstracted as a higher order function:

$$(D\&C\ t\ b\ d\ c)\ x = \texttt{if}(t(x))\ \texttt{then}\ b(x)\ \texttt{else}\ c((\texttt{map}(D\&C\ t\ b\ d\ c)(d(x)))$$

where the parameters represent: the function deciding if a termination case has been reached ($t : \alpha \rightarrow \texttt{boolean}$[1]), the function computing the base case ($b : \alpha \rightarrow \beta$), the function splitting a non-base case into sub-cases ($d : \alpha \rightarrow [\alpha]$) and the function combining the results of sub-cases ($c : [\beta] \rightarrow \beta$). By providing the appropriate parameters the user can obtain the working divide&conquer function $(D\&C\ t\ b\ d\ c) : \alpha \rightarrow \beta$. For example, in order to get a working D&C sort function the user should provide a t indicating when the sorting has to be performed sequentially, a b sequentially computing the sort for base cases, a d for splitting (long) lists into a list of (shorter) sublists and finally a c function for combining ordered sublists into an ordered list. All the details relating to the actual computation of the sort according to the divide&conquer pattern are hidden (or "embedded") within the $D\&C$ higher order function.

More generally, complex and richer sets of skeletons are provided that allow the user to express a computation as a skeleton/pattern composition. The following core (abstract) skeleton set has been defined (in slightly different forms) in a large number of skeleton frameworks, including P3L [4], Muesli [14], Lithium/muskel [3,8], SkeTo [15], ASSIST [2] and Calcium [5]:

$$S = seq(C) \mid farm(S) \mid pipe(S, S) \mid map(S) \mid reduce(S)$$

$$C = \langle some\ function\ code\ in\ any\ suitable\ host\ language \rangle$$

In this case, $farm$ denotes the embarrassingly parallel, stream apply-to-all pattern, *pipe* denotes the usual stream parallel computation in stages, *map* and *reduce* model the corresponding data parallel collective operations, and, finally, *seq* just wraps sequential code in such a way it can be used as a parameter in another skeleton.

These higher-order functions can be provided in a way suitable for to the programming model adopted by the user. For example, as library objects for OO programmers, or as composite components or plain services for component and service-oriented programmers.

[1] We denote with $f : \alpha \rightarrow \beta$ the type of a function processing items of type α to produce results of type β.

Algorithmic skeletons may be used to raise the level of abstraction presented to the programmer of parallel and distributed applications by abstracting (and confining in the implementation level) all those aspects not directly related to the function the programmer wants to compute and to the *kind* of parallel/distributed patterns to be exploited. Those details are dealt with in the implementation of the algorithmic skeleton and thus do not directly concern the application programmer. Furthermore, some quantitative aspects that have an impact on the skeleton implementation and, as a consequence, on its performance, can be abstracted through parameters, thus allowing easy skeleton tuning by the application programmer or, as alternative, viable ways to control skeleton behaviour in the implementation (i.e. in the compiling tools and/or in the run time support). For example, consider the parallelism degree. This could either be one of the parameters provided by the application programmer as a kind of SLA when instantiating the skeleton, or it could be a parameter completely managed by the implementation. In the former case, the programmer may make several test runs before identifying the "optimal" parallelism degree for his application, possibly without the need of recompiling the application[2]. In the latter case, the run time system may adjust the parallelism degree upon recognition that the performance of the application does not fit that predicted by the abstract performance model.

2.2 Management Abstractions

When managing parallel and distributed applications, several non-functional concerns such as performance, fault tolerance, security and adaptivity may require consideration on an on-going basis with little or no input from the user. These concerns can be handled at two different levels: either directly at the application code level or within some autonomic manager interacting with the application code. In the former case, the burden lies completely with the application programmer; in the latter case, it becomes a system programmer concern. Furthermore, in the latter case many more autonomic aspects can be included in the manager, making it potentially even more effective, leveraging on the fact that it is implemented as an independent activity. In both cases, however, what typically has to be implemented is a control loop:

$$(\texttt{AM}\ m\ a\ p\ e)(C) = (\texttt{AM}\ m\ a\ p\ e)(e\ (p\ (a\ (m\ C)))\ C)$$

where m represents the monitoring actuated on the current computation, a represents the analyse activity identifying a suitable policy to be executed, p is the function providing plans to implement a given policy and finally, e is the execute function, applying plans to computations (C) in order to adapt the computation according to the chosen policy.

As in the case of structuring abstractions, if users are provided suitable abstractions modelling this kind of autonomic controller, they can obtain running,

[2] Assuming the compiled application will run with some kind of `-np`, MPI-like parameter.

optimized autonomic manager by specializing the general, second order, recursive function with appropriate parameters.

For example, consider performance tuning in an embarrassingly data parallel computation. In this case, the user may provide a monitor function computing current throughput (time spent computing a single data item and time spent to retrieve input data and to deliver (partial) results), an analyse phase that will consider whether the grain limit for this computation has been reached (i.e. the grain such that the time spent to deliver input data to and retrieve results from remote computing elements equals the time spent to compute the data item locally), a plan phase determining either to increase or to decrease the allocated computational resources and finally an execute function applying the planned activities on the current computation to implement policy decisions.

Autonomic managers may be used to raise the level of abstraction presented to the programmer of parallel and distributed applications by abstracting all those aspects directly related to management of their non-functional concerns. To enhance further the abstraction level presented to the autonomic manager designers/implementors, we found it beneficial to express the autonomic cycle behaviour through *business rule*s rather than via the functions mentioned above. In this case, the system programmer is given a set of monitoring and actuation actions, that can be used to get measures about the current computation and to implement adaptive actions, respectively. Then he may completely customize the autonomic manager behaviour by providing *if-then* rules where the *if* part is a first order predicate on the monitored values and the *then* part corresponds to the plan/execute component of the control loop. The autonomic manager periodically scans the (prioritized) rules available, identifies the *fireable* ones (those whose *if* predicate evaluates to *true*) and finally applies the adaptive actions specified in the corresponding *then* part.

3 Structuring and Management Co-design

In this section we propose a co-design approach to developing the structure and management of distributed systems. The aim is to devise a methodology for identifying and implementing distributed programming abstractions which model parallel/distributed computation patterns and handle non-functional features. This methodology may be used to make available programming abstractions that i) can be used (i.e. instantiated according to the general programming model chosen[3]) to implement applications matching exactly the particular parallel/distributed pattern defined by the abstraction, and ii) take care of relevant non-functional aspects via autonomic managers, where the non-functional requirements are specified by the user via a SLA.

In order to be able effectively to co-design distributed application structuring and non-functional concern management, we must identify first suitable interaction patterns between the two. In other words, we have to establish which plays an active role and which a passive role, and how the active actor may impact upon the

[3] OO, component based, service based, ...

passive counterpart. It seems natural to consider the managers as being the active entities and the skeletons the passive ones as autonomic managers are devoted to taking decisions that have to be applied in the execution of an application.

The second step concerns identification of the kind of malleability supported by the passive actor. This requires definition of those parameters which may be controlled from outside an algorithmic skeleton, and thus, as a consequence, which adaptive actions can be ordered by the manager. Triggering of the adaptive actions requires definition of the observable measures of the skeleton that can be sampled via monitoring and, in turn, definition of the policies and plans to be considered in the manager. If the skeleton does not provide suitable mechanisms to monitor relevant parameters then, no matter how good the abstract performance model we have in the manager, it is not possible to perceive in the manager that the computation is not performing as expected and thus trigger some adaptation action. Similarly, if the skeleton implementation does not provide effective actuation mechanisms, then the manager cannot implement any kind of corrective policy. Therefore the skeleton implementation *must* provide appropriate monitoring and actuation interfaces for the manager.

Skeleton malleability can be achieved via appropriate parameters. Thus we will assume that each of the skeletons considered has more that just the function parameters outlined in Sec. 2.1. In particular, we will consider that each of the skeletons has a parameter specifying the *parallelism degree* of the skeleton itself. In addition, a skeleton may have a boolean parameter stating whether the communications involving that skeleton should be secured or not. Whether this should be an actual parameter or some kind of meta-data (e.g. provided via annotations) is beyond the scope of this work. With these parameters available, one can easily envisage managers interacting with running skeletons by setting/resetting those parameters via appropriate setter/getter methods provided by the skeleton interface.

The factors listed determine the nature of the co-design that can be used for structuring skeletons and autonomic managers. It is clear that the more effects we want to control via the manager, the more monitoring and actuators methods should be implemented in the managed skeletons. It is equally clear that the better interaction we have among manager and managed entities, the better autonomic management policies we can implement.

The methodology mentioned at the beginning of this section, can thus be summarized as follows.

1. First, the skeleton set used to structure our application is identified.
2. Then the malleable skeleton interface is designed and implemented allowing i) monitoring of the measures of interest for implementing the autonomic management policies and ii) actuation of the decisions taken by the autonomic manager.
3. Finally, autonomic manager control loop is implemented that i) gathers the relevant monitoring values from the malleable skeleton, ii) activates the rule engine and iii) finally executes the fireable rules through the actuation mechanism interface of the malleable skeleton.

The result will be a programming framework where application programmers build applications by instantiating the skeleton/manager programming abstractions with suitable parameters and devote to the implementation[4] most (all) of the cumbersome activities needed to develop efficient, autonomically managed parallel/distributed applications.

4 Implementation and Results

The approach described in Section 3 has been experimented with in several projects, and the table in Fig. 1 recalls the distinguishing features of the corresponding prototypes. The more important experiments have been made in the framework of the CoreGRID FP6 NoE while designing the GCM (Grid Component Model) and then within GridCOMP, the spin-off FP6 STREP aimed at providing an open source reference implementation of GCM.

	User programming model	Skeletons supported	Manager implemen-tation	Handled non functonal concerns	Policies	User contracts	Skeleton monitoring/ actuator interface
GCM BS	*Component based*	*Farm. Pipe, Dataparallel*	*Inner component*	*Performance*	*Java code then drools*	*Java code then drools*	*AC controller*
SCA service	*Service based*	*Farm, Pipe (partial)*	*Inner service*	*Performance*	*Drools*	*Drools*	*Bean*
muskel	*OO (library)*	*Farm, Pipe (user expandable)*	*Object with callbacks*	*Fault tolerance*	*Java code*	*Java object*	*RMI + local objects*

Fig. 1. Different features of several co-designed skeleton+autonomic manager frameworks (GCM Behavioural skeleton framework [1], SCA service autonomic task farm experiment [9], muskel full Java skeleton library [8])

In this context, the *behavioural skeleton* concept has been developed [1] within the reference GCM implementation built on top of ProActive middleware [17]. A behavioural skeleton (BS) is a component modeling a common parallelism exploitation pattern on parallel and distributed architectures and providing an autonomic manager taking care of the performance non-functional aspects related to the parallelism exploitation pattern considered. In GCM, task farm and data parallel behavioural skeletons have already been implemented and the implementation of a pipeline behavioural skeleton is undergoing. The task farm BS models embarrassingly parallel computations, the data parallel BS models several kinds of data parallel computations, including those sharing a state among their parallel activities, and the pipeline BS models computation in stages. All the current behavioural skeletons handle only performance issues in their autonomic managers.

[4] To the system programmers, but this activity is needed just once, when the skeleton/manager pairs are designed and implemented.

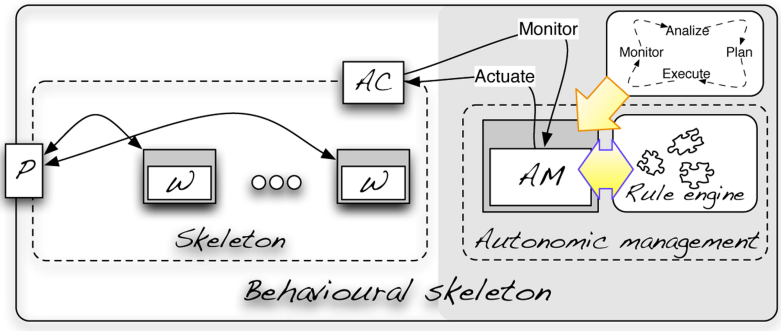

Fig. 2. Abstract schema of Behavioural skeleton: P represents the functional interface, the grey part represents component "membrane", i.e. the non-functional part of the composite GCM component, AC is the the autonomic controller providing a monitoring and actuator interface to the manager. The W are the inner components whose parallel/distributed interaction is managed by the skeleton.

Autonomic managers in behavioural skeletons are implemented using a JBoss rule engine [12][5]. Rules establish manager policies. The precondition part uses methods provided by an *autonomic controller* (AC) bean associated with the implemented skeleton (see Fig. 2). Actuation mechanisms are provided also as methods of the same AC bean, and they are called while executing the action part of fireable rules. The rules are evaluated in a control loop: once the execution of the currently (higher priority) fireable rule action part is terminated, the evaluation of the precondition part starts again. Rules currently included in the autonomic managers allow increase and decrease of the resources allocated to a BS in such a way that a user supplied *performance contract* (SLA) is ensured in the presence of variations in the load and availability of the computing and internetworking resources used to run the application. Performance contracts, in turn, are expressed in terms of throughput via JBoss rules submitted (statically, at the beginning of the application execution, or dynamically, while the execution progresses) to the autonomic manager of the BS.

Separation of concerns is achieved as proposed in Section 3 as behavioural skeletons are implemented by system programmers and application programmers need only choose one of the available BS and provide the appropriate parameters to get a fully working, performance optimized application.

Using the GCM BS prototype we developed several synthetic applications and GridCOMP partners developed more realistic use cases, including biometric identification and fluid-dynamic parameter sweeping applications [11]. Typical results achieved with the GCM BS are shown in Fig. 3. The plot relates to an

[5] Jboss uses *Drools*, that is "a business rule management system (BRMS) with a forward chaining inference based rules engine, more correctly known as a production rule system, using an enhanced implementation of the Rete algorithm" according to Wikipedia (`http://en.wikipedia.org/wiki/Drools`)

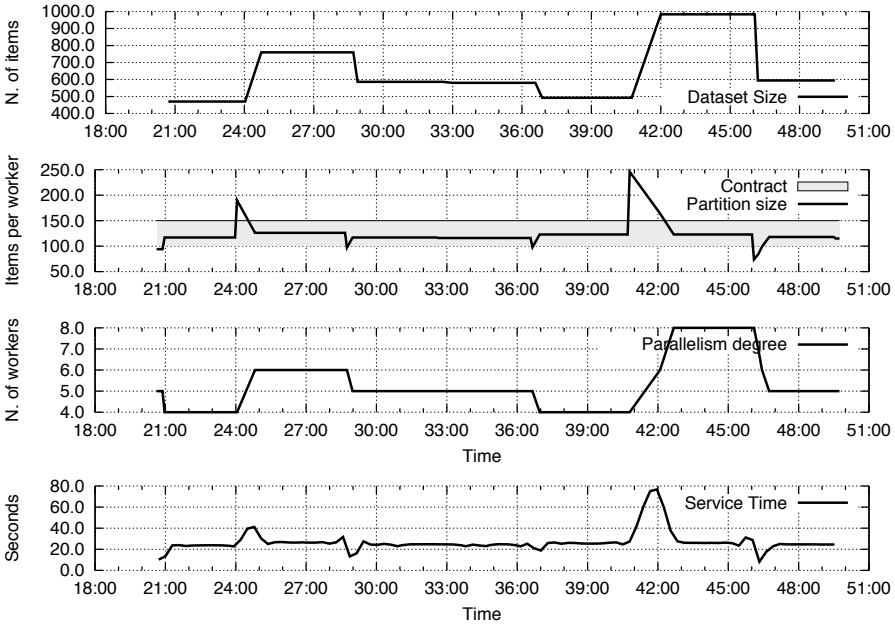

Fig. 3. GCM behavioural skeletons at work

application looking up items in a databases. Several user provided databases are supplied, each with its own stream of items to be searched. Comparing input items with an item in the database takes a non-negligible time. The user-specified contract requires that each worker should have between 100 and 150 database entries to compare, in order to get a reasonable response time (grey bar in the second graph). The dimensions of the databases supplied are plotted on the first graph and the actual database partition size in workers is the line plot of the second graph. The autonomic manager of the data parallel BS reacts by adding and removing workers (third graph) in such a way that the requested partition size varies within the contract range and an acceptable service time is achieved (fourth plot). These results have been achieved by running the application on three different architectures: a Fast Ethernet NOW, GRID 5000 [10] and an SMP multicore core architecture (up to 8 cores). In all cases, the BS autonomic managers reacted as expected and performance has been adapted to the varying load conditions of the target architecture. Experiments are ongoing that demonstrate that the same results can be achieved when heterogeneous NOWs of single and multi-core machines are targeted.

These results assess the concepts and methodologies discussed here. Furthermore, in [9] we discussed a similar implementation providing a *WorkPool* service computing independent tasks according to a task farm skeleton, whose execution is managed by a *WorkpoolManager* service using a JBoss drools engine and interfacing (monitor and actuator interfaces) the managed skeleton to the manager

via a suitable interface bean. With this prototype[6] we demonstrated that the co-design methodology described in Sec. 3 can be easily exported to other programming frameworks (the service framework in this case) while preserving the programmer's investment and fulfilling the "minimal disruption" requirement stated by Cole in his skeleton "manifesto" [6].

5 Multiple Non-functional Concern Management

While the proposed approach has been shown to be effective for a range of underlying programming paradigms, there remain significant challenges, not least in addressing systems where multiple non-functional concerns are to be managed simultaneously. For example, the manager may be required to handle performance, security and fault tolerance aspects of the pattern/skeleton at hand. Thus, in the general case, monitoring may involve different, possibly independent values, independent policies may exist relative to the different non-functional concerns and, finally, decisions taken in relation to different policies may be somehow inconsistent or even conflicting.

Therefore some *meta* policy may be needed to handle autonomic management of different non-functional concerns. The simplest such policy is the weighted one. Different non-functional concerns are given a weight (or a priority) and either those with higher weight/priority are considered first (i.e. the corresponding policies are considered and the corresponding actions taken) or, in the case of policies whose effects can be somehow "scaled" a weighted policy effect is considered (i.e. policy i actions are executed with weight w_i). However this strategy cannot be applied in the general case, as in the general case it makes no sense to execute an action "with weight w_i". We need more complex strategies, and these strategies can probably best be implemented using some business rule engine such that used to implement the rules relating to autonomic management of a single concern.

In this case, we have to distinguish rules used to implement *intra*-non-functional concern policies (*ground* rules) from those implementing *inter*-concern policies (*meta* rules). In addition to normal priority-based handling of rules, system programmers should be able to exploit meta-rules *before* actually actuating fireable ground-rules, but *after* knowing which exact ground rules are fireable and the relative priorities.

For example, consider the case where both performance and security are being managed. Suppose we have a rule stating that we can add more resources to the current computation if it is under performing, and another stating that a resource can be managed without the need to use secure communications and that both are fireable. Application of either rule will probably increase the performance of the application and so there should be some meta-rule stating how they should be applied: both, and if so which one first, or just one, and then which one.

[6] The prototype is implemented on top of the Tuscany [19] implementation of SCA, the Service Component Architecture [18] and is referred to as "SCA service" in Fig. 1.

This simple example indicates how complexity escalates when even straightforward concerns are combined. The challenge of determining policies for dealing with multiple non-functional concerns in concert is huge but we believe the idea of meta-rules will at least provide a framework in which these issues can be addressed.

6 Conclusions

We discussed how co-design of parallel/distributed computation structuring and autonomic management can facilitate the development of distributed systems by enforcing a separation of concerns at two levels. First, suitable abstractions may be provided to the application programmer, ensuring that he can concentrate on the core functional code and on specifying non-functional requirements as a SLA. In turn, many of the more challenging aspects of the distributed system development are left in the hands of the system programmer who is well placed to deal with these challenges. Second the system programmer is further aided by the separation of structure from (non-functional concern) management together with clear guidelines as to how the two should interface.

Preliminary results indicate that the approach is reasonable and feasible, both in the case COW/NOWs and of multi- many-core networks.

The proposed programming abstractions and co-design approach appears also to be suitable for implementing cloud programming environments, as they decouple programming effort from specific knowledge of the target architecture while, at the same time, preserving those positive aspects deriving from efficient implementation of both structuring and management patterns. Indeed, in the case of non-functional aspects in cloud computing, an approach similar to that proposed is essential, as the application programmer typically will have no possibility of directly managing such concerns.

References

1. Aldinucci, M., Campa, S., Danelutto, M., Vanneschi, M., Dazzi, P., Laforenza, D., Tonellotto, N., Kilpatrick, P.: Behavioural skeletons in GCM: autonomic management of grid components. In: Baz, D.E., Bourgeois, J., Spies, F. (eds.) Proc. of Intl. Euromicro PDP 2008: Parallel Distributed and network-based Processing, Toulouse, France, Feburary 2008, pp. 54–63. IEEE, Los Alamitos (2008)
2. Aldinucci, M., Coppola, M., Danelutto, M., Vanneschi, M., Zoccolo, C.: ASSIST as a research framework for high-performance grid programming environments. In: Cunha, J.C., Rana, O.F. (eds.) Grid Computing: Software environments and Tools, ch. 10, pp. 230–256. Springer, Heidelberg (2006)
3. Aldinucci, M., Danelutto, M., Teti, P.: An advanced environment supporting structured parallel programming in Java. Future Generation Computer Systems 19(5), 611–626 (2003)
4. 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)

5. Caromel, D., Leyton, M.: Fine tuning algorithmic skeletons. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641, pp. 72–81. Springer, Heidelberg (2007)
6. Cole, M.: Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. Parallel Computing 30(3), 389–406 (2004)
7. Danelutto, M.: On skeletons and design patterns. In: Joubert, G.R., Murli, A., Peters, F.J., Vanneschi, M. (eds.) Parallel Computing: Advances and Current Issues (Proc. of Intl. ParCo 2001), Naples, Italy, pp. 425–432. Imperial College Press, London (2001)
8. Danelutto, M.: QoS in parallel programming through application managers. In: Proc. of Intl. Euromicro PDP: Parallel Distributed and network-based Processing, Lugano, Switzerland, pp. 282–289. IEEE, Los Alamitos (2005)
9. Danelutto, M., Zoppi, G.: Behavioural skeletons meeting services. In: Bubak, M., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2008, Part I. LNCS, vol. 5101, pp. 146–153. Springer, Heidelberg (2008)
10. Grid 5000 home page (2008), `http://www.grid5000.fr`
11. GridCOMP Use case home page (2008), `http://gridcomp.ercim.org/content/view/41/39/`
12. JBoss rules home page (2008), `http://www.jboss.com/products/rules`
13. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer 36(1), 41–50 (2003)
14. Kuchen, H.: A skeleton library. In: Monien, B., Feldmann, R.L. (eds.) Euro-Par 2002. LNCS, vol. 2400, pp. 620–629. Springer, Heidelberg (2002)
15. Matsuzaki, K., Iwasaki, H., Emoto, K., Hu, Z.: A library of constructive skeletons for sequential style of parallel programming. In: InfoScale 2006: Proceedings of the 1st international conference on Scalable information systems. ACM, New York (2006)
16. Mattson, T.G., Sanders, B.A., Massingill, B.L.: Patterns for Parallel Programming. Addison-Wesley Professional, Reading (2005)
17. ProActive home page (2006), `http://www-sop.inria.fr/oasis/proactive/`
18. Service component architecture (2008), `http://www.ibm.com/developerworks/library/specification/ws-sca/`
19. Tuscany home page (2008), `http://incubator.apache.org/tuscany/`