BEHAVIOURAL SKELETONS FOR COMPONENT AUTONOMIC MANAGEMENT ON GRIDS*

Marco Aldinucci, Sonia Campa, Marco Danelutto
Computer Science Department, University of Pisa
Pisa, Italy
{aldinuc, campa, marcod}@di.unipi.it

Patrizio Dazzi, Domenico Laforenza, Nicola Tonellotto
ISTI – CNR
Pisa, Italy
{patrizio.dazzi, domenico.laforenza, nicola.tonellotto}@isti.cnr.it

Peter Kilpatrick
Computer Science Department, Queen's University
Belfast, UK
p.kilpatrick@qub.ac.uk

Abstract We present behavioural skeletons for the CoreGRID Component Model, which are an abstraction aimed at simplifying the development of GCM-based self-management applications. Behavioural skeletons abstract component self-managent in component-based design as design patterns abstract class design in classic OO development. As here we just wish to introduce the behavioural skeleton framework, emphasis is placed on general skeleton structure rather than on their autonomic management policies.

Keywords: Components, grid, code adaptivity, autonomic computing, skeletons.

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

1. Introduction and Related Work

While developing grid applications neither the target platforms nor their status are fixed, statically or dynamically [12]. This makes application adaptivity an essential feature in order to achieve high performance and to exploit efficiently the available resources [1].

In recent years, several research initiatives exploiting component technology [9] have investigated the possibilities related to component adaptation, i.e. the process of changing the component for use in different contexts. This process can be conceived as either a static or dynamic process.

The basic use of static adaptation covers straightforward but popular methodologies such as *copy-paste* and *OO inheritance*. A more advanced usage covers the case in which adaptation happens at run-time. These systems enable dynamically defined adaptation by allowing adaptations, in the form of code, scripts or rules, to be added, removed or modified at run-time [7]. Among them it is worth distinguishing the systems where all possible adaptation cases have been specified at compile time, but the conditions determining the actual adaptation at any point in time can be dynamically changed [4]. Dynamically adaptable systems rely on a clear separation of concerns between adaptation and application logic. This approach has recently gained increased impetus in the grid community, especially via its formalisation in terms of the *Autonomic Computing* (AC) paradigm [15, 5, 3]. The AC term is emblematic of a vast *hierarchy* of self-governing systems, many of which consist of many interacting, self-governing components that in turn comprise a number of interacting, self-governing components at the next level down [13]. An autonomic component will typically consist of one or more managed components coupled with a single autonomic manager that controls them. To pursue its goal the manager may trigger an adaptation of the managed components to react to a run-time change of application QoS requirements or to the platform status.

In this regard, an assembly of self-managed components implements, via their managers, a distributed algorithm that manages the entire application. Several existing programming frameworks aim to ease this task by providing a set of mechanisms to dynamically install reactive rules within autonomic managers. These rules are typically specified as a collection of `when-`*event*`-if-`*cond*`-then-`*act* clauses, where *event* is raised by the monitoring of internal or external component activity (e.g. the component server interface received a request, and the platform running a component exceeded a threshold load, respectively); *cond* is an expression over internal component attributes (e.g. component life-cycle status); *act* represents an adaptation action (e.g. create, destroy a component, wire, unwire components, notify events to another component's manager). Several programming frameworks implement variants of this general idea, including ASSIST [20, 1], AutoMate [17], SAFRAN [10],

and finally the forthcoming CoreGRID Grid Component Model (GCM) [9]. The latter two are derived from a common ancestor, i.e. the Fractal hierarchical component model [16]. All the named frameworks, except SAFRAN, are targeted to distributed applications on grids.

Though such programming frameworks considerably ease the development of an autonomic application for the grid (to various degrees), they rely fully on the application programmer's expertise for the set-up of the management code, which can be quite difficult to write since it may involve the management of black-box components, and, notably, is tailored for the particular component or assembly of them. As a result, the introduction of dynamic adaptivity and self-management might enable the management of grid dynamism and uncertainty aspects but, at the same time, decreases the component reuse potential since it further specialises components with application specific management code.

In this work, we propose *behavioural skeletons* as a novel way to describe autonomic components in the GCM framework. Behavioural skeletons aim to describe recurring patterns of component assemblies that can be (either statically or dynamically) equipped with correct and effective management strategies with respect to a given management goal. Behavioural skeletons help the application designer to 1) design component assemblies that can be effectively reused, and 2) cope with management complexity by providing a component with an explicit context with respect to top-down design (i.e. component nesting).

2. Grid Component Model

GCM is a hierarchical component model explicitly designed to support component-based autonomic applications in grid contexts. GCM allows component interactions to take place with several distinct mechanisms. In addition to classical "RPC-like" use/provide ports (or client/server interfaces), GCM allows data, stream and event ports to be used in component interaction. Furthermore, collective interaction patterns (communication mechanisms) are also supported. The full specification of GCM can be found in [9].

GCM is therefore assumed to provide several levels of autonomic managers in components, that take care of the non-functional features of the component programs. GCM components thus have two kinds of interfaces: functional and non-functional ones. The functional interfaces host all those ports concerned with implementation of the functional features of the component. The non-functional interfaces host all those ports needed to support the component management activity in the implementation of the non-functional features, i.e. all those features contributing to the efficiency of the component in obtaining the expected (functional) results but not directly involved in result computation. Each GCM component therefore contains an *Autonomic Manager* (AM), interacting with other managers in other components via the component non-

functional interfaces. The AM implements the autonomic cycle via a simple program based on the reactive rules described above. In this, the AM leverages on component controllers for the *event* monitoring and the execution of reconfiguration *actions*. In GCM, the latter controller is called the *Autonomic Behaviour Controller* (ABC). This controller exposes server-only non-functional interfaces, which can be accessed either from the AM or an external component that logically surrogates the AM strategy. We call *passive* a GCM component exhibiting just the ABC, whereas we call *active* a GCM component exhibiting both the ABC and the AM.

3. Describing Adaptive Applications

The architecture of a component-based application is usually described via an ADL (Architecture Description Language) text, which enumerates the components and describes their relationships via the *used-by* relationship. In a hierarchical component model, such as the GCM, the ADL describes also the *implemented-by* relationship, which represents the component nesting.

Typically, the ADL supplies a static vision of an application, which is not fully satisfactory for an application exhibiting autonomic behaviour since it may autonomously change behaviour during its execution[1]. Such change may be of several types:

- *Component lifecycle.* Components can be started or stopped.
- *Component relationships.* The used-by and/or implemented-by relationships among components are changed. This may involve component creation/destruction, and component wiring alteration.
- *Component attributes.* A refinement of the behaviour of some components (which does not involve structural changes) is required, usually over a pre-determined parametric functionality.

In the most general case, an autonomic application may evolve along adaptation steps that involve one or more changes belonging to these three classes. In this regard, the ADL just represents a snapshot of the launch time configuration.

The evolution of a component is driven by its AM, which may request management action with the AM at the next level up in order to deal with management issues it cannot solve locally. Overall, it is a part of a distributed system that cooperatively manages the entire application.

In the general case, the management code executing in the AM of a component depends both on the component's functional behaviour and the goal of the management. The AM should also be able to cooperate with other AMs, which are unknown at design time due to the nature of component-based design. Currently, programming frameworks supporting the AC paradigm (such as the

[1] However, note that with GCM the ADL provides a hook to accommodate a *behavioural specification*.

ones mentioned in Sec. 1) just provide mechanisms to implement management code. This approach has several disadvantages, especially when applied to a hierarchical component model:

- The management code is difficult to develop and to test since the context in which it should work may be unknown.
- The management code is tailored to the particular instance of the management elements (inner components), further restricting the component possible reusability.

For this reason, we believe that the "ad-hoc" approach to management code is unfit to be a cornerstone of the GCM component model.

4. Behavioural Skeletons

Behavioural skeletons aim to abstract parametric paradigms of GCM component assembly, each of them specialised to solve one or more management goals belonging to the classical AC classes, i.e. configuration, optimisation, healing and protection.

Behavioural skeletons represent a specialisation of the algorithmic skeleton concept for component management [8]. Algorithmic skeletons have been traditionally used as a vehicle to provide efficient implementation templates of parallel paradigms. Behavioural skeletons, as algorithmic skeletons, represent patterns of parallel computations (which are expressed in GCM as graphs of components), but in addition they exploit the inherent skeleton semantics to design sound self-management schemes of parallel components.

Due to the hierarchical nature of GCM, behavioural skeletons can be identified with a composite component with no loss of generality (identifying skeletons as particular higher-order components [11]). Since component composition is defined independently from behavioural skeletons, they do not represent the exclusive means of expressing applications, but can be freely mixed with non-skeletal components. In this setting, a behavioural skeleton is a composite component that

- exposes a description of its functional behaviour;
- establishes a parametric orchestration schema of inner components;
- may carry constraints that inner components are required to comply with;
- may carry a number of pre-defined plans aimed at coping with a given self-management goal.

Behavioural skeleton usage helps designers in two main ways: the application designer benefits from a library of skeletons, each of them carrying several predefined, efficient self-management strategies; and, the component/application designer is provided with a framework that helps the design of new skeletons and their implementations.

The former task is achieved because (1) skeletons exhibit an explicit higher-order functional semantics, which delimits the skeleton usage and definition domain; and (2) skeletons describe parametric interaction patterns and can be designed in such a way that parameters affect non-functional behaviour but are invariant for functional behaviour.

5. A Basic Set of Behavioural Skeletons

Here we present a basic set of behavioural skeletons for the sake of exemplification. Despite their simplicity, they cover a significant set of parallel computations in common usage.

One class of behavioural skeletons springs from the idea of *functional replication*. Let us assume the skeletons in this class have two functional interfaces: a one-to-many stream server S, and a many-to-one client stream interface C (see Fig. 1). The skeleton accepts requests on the server interface; and dispatches them to a number of instances of an inner component W, which may propagate results outside the skeleton via C interface. Assume that replicas of W can safely lose their internal state between different calls. For example, the component has just a transient internal state and/or stores persistent data via an external data-base component.

Farm. A stream of tasks is absorbed by a *unicast* S, each task is computed by one instance of W and sent to C, which collect tasks *from-any*. This skeleton can be equipped with a self-optimising policy because the number of Ws can be dynamically changed in a sound way since they are stateless. The typical QoS goal is to keep a given limit (possibly dynamically changing) of served requests in a given time frame. The AM just checks the average time tasks need to traverse the skeleton, and eventually reacts by creating/destroying instances of Ws, and wiring/unwiring them to/from the interfaces.

Data-Parallel. A stream of tasks is absorbed by a *scatter* S; each task is split in (possibly overlapping) partitions, which are distributed to replicas of W to be computed. Results are *gathered* and assembled by G in a single item. As in the previous case, the number of Ws can be dynamically changed (between different requests) in a sound way since they are stateless. As in the previous case, the skeleton can be equipped with a self-configuration goal, i.e. resource balancing and tuning (e.g. disk space, load, memory usage), that can be achieved by changing the partition-worker mapping in S (and C, accordingly).

Active-Replication. A stream of tasks is absorbed by a *broadcast* S, which sends identical copies to the Ws. Results are sent to G, which reduces them. This paradigm can be equipped with a self-healing policy because it can deal with Ws that do not answer, produce an approximate or wrong answer by means

of a result reduction function (e.g. by means of averaging or voting on results).

The presented behavioural skeletons can be easily adapted to the case that S is a RPC interface. In this case, the C interface can be either a RPC interface or missing. Also, the functional replication idea can be extended to the stateful case by requiring the inner component Ws to expose suitable methods to serialise, read and write the internal state. A suitable manipulation of the serialised state enables the reconfiguration of workers (also in the data-parallel scenario [1]).

In order to achieve self-healing goals some additional requirements on the GCM implementation level should be enforced. They are related to the implementation of GCM mechanisms, such as component membranes and their parts (e.g. interfaces) and messaging system. At the current level of interest, they are primitive mechanisms, in which correctness and robustness should be enforced ex-ante, at least to achieve some of the described management policies.

The process of identification of other skeletons may benefit from the work done within the software engineering community, which identified some common adaptation paradigms, such as *proxies* [18], which may be interposed between interacting components to change their interaction relationships; and dynamic *wrappers* [19]. Both of these can be used for self-protection purposes. For example, a pair of encrypting proxies can be used to secure a communication between components. Wrapping can be used to hide one or more interfaces when a component is deployed into an untrusted platform.

5.1 GCM implementation of Behavioural Skeletons

In terms of the GCM specification [9], a behavioural skeleton is a particular composite component exhibiting an autonomic conformance level strictly greater than one, i.e. a component with passive or active autonomic control. The component exposes pre-defined functional and non-functional client and server interfaces according to the skeleton type; functional interfaces are usually collective and configurable. Since skeletons are fully-fledged GCM components, they can be wired and nested via standard GCM mechanisms. From the implementation viewpoint, a behavioural skeleton is a partially defined composite component, i.e. a component with placeholders, which may be used to instantiate the skeleton. As sketched in Fig. 1, there are three classes of placeholders:

1 The functional interfaces S and C that are GCM membrane controllers.
2 The AM that is a particular inner component. It includes the management plan, its goal, and exported non-functional interfaces.
3 Inner component W, implementing the functional behaviour.

The orchestration of the inner components is implicitly defined by the skeleton type. In order to instantiate the skeleton, placeholders should be filled with

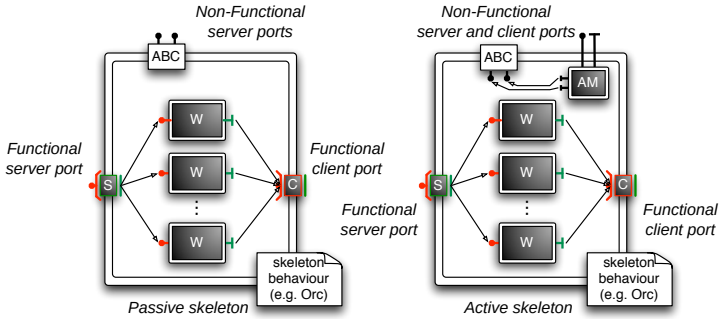

Figure 1. GCM implementation of functional replication. ABC = Autonomic Behaviour Controller, AM = Autonomic Manager, W = Worker component, S = Server interface (one-to-many communication e.g. broadcast, data-parallel scatter, unicast), C = Client interface (many-to-one communication e.g. from-any, data-parallel gather, reduce, select).

suitable entities. Observe that only entities in the former two classes are skeleton specific.

In addition to a standard composite component, a behavioural skeleton is further characterised by a formal (or semi-formal) description of the component behaviour. This description can be attached to the ADL component definition via the standard GCM ADL hook. In this work we propose a description based on the Orc language, which appears suitable for specification of orchestration in distributed systems [2].

6. Specifying Skeleton Behaviour

Autonomic management requires that, during execution of a system, components of the system are replaced by other components, typically having the same functional behaviour but exhibiting different non-functional characteristics. The application programmer must be confident about the behaviour of the replacements with respect to the original. The behavioural skeleton approach proposed supports these requirements in two key ways:

1 The use of skeletons with its inherent parametrisation permits relatively easy parameter-driven variation of non-functional behaviour while maintaining functional equivalence.
2 The use of Orc to describe component behaviour gives the developer a firm basis on which to compare the properties of alternative realisations in the context of autonomic replacement.

In the following we present an Orc specification of the functional replication example depicted in Fig. 1 followed by several alternative formulations of the client and server interface behaviours. First, a brief overview of the Orc lan-

guage is presented. A formal description of management plans is not presented here. The skeleton designer can use the description to prove rigorously (manually, at present) that a given management strategy will have predictable or no impact on functional behaviour. The quantitative description of QoS values of a component with respect to a goal, the automatic validation of management plans w.r.t. a given functional behaviour are interesting related topics, which are the subject of ongoing research but outside the scope of the present work.

6.1 The Orc notation

The orchestration language Orc of Misra and Cook [14] is targeted at the description of systems where the challenge lies in organising a set of computations, rather than in the computations themselves. Orc has, as primitive, the notion of a site call, which is intended to represent basic computations. A site, either returns a *single* value or remains silent. Three operators (plus recursion) are provided for the orchestration of site calls:
Sequential composition: $E_1 > x > E_2(x)$ evaluates E_1, receives a result x, calls E_2 with parameter x. If E_1 produces two results, say a and b, then E_2 is evaluated twice, once with argument a and once with argument b. The abbreviation $E_1 \gg E_2$ is used for $E_1 > x > E_2$ when evaluation of E_2 is independent of x.
Parallel composition: $(E_1 \mid E_2)$ evaluates E_1 and E_2 in parallel. Both evaluations may produce replies. Evaluation of the expression returns the merged output streams of E_1 and E_2.
Asymmetric parallel composition: E_1 where $x :\in E_2$ begins evaluation of both E_1 and $x :\in E_2$ in parallel. Expression E_1 may name x in some of its site calls. Evaluation of E_1 may proceed until a dependency on x is encountered; evaluation is then suspended. The first value delivered by E_2 is returned in x; evaluation of E_1 can proceed and the thread E_2 is terminated.

Orc has a number of special sites: "0" never responds; "if b" returns a signal if b is true and remains silent otherwise; "let" always publishes its argument.

Finally, while Orc does not have an explicit concept of "process", processes may be represented as expressions which, typically, name channels that are shared with other expressions. In Orc a channel is represented by a site [14]. *c.put(m)* adds m to the end of the (FIFO) channel and publishes a signal. If the channel is non-empty *c.get* publishes the value at the head and removes it; otherwise the caller of *c.get* suspends until a value is available.

6.2 The Description of Skeletons in Orc

Assume that *data* is sent by an interface S along a number, N, of channels in_i to N workers W_i. Each worker processes its data and sends the result along a channel out_i to interface C (see Fig. 1). The distribution of data by S to the

channels may be based on various algorithms depending on the nature of the overall task: see below.

Assume that *data* is a *list* of items to be processed; *#data* is the number of items in *data*; *in* is a *list* of N channels connecting the port S with the workers. *out* is a *list* of N channels connecting the port C with the workers. For a list, l, *head(l)* returns the head of (non-empty) l; *tail(l)* returns the tail of (non-empty) l. Denote by l_i the i^{th} element of the list l. The skeleton system depicted in Fig. 1 may be defined in Orc as follows:

$$system(data, S, G, W, in, out, N) \triangleq$$
$$\quad S(data, in) \mid (|i : 1 \leq i \leq N : W_i(in_i, out_i)) \mid C(out)$$
$$W_i(in_i, out_i) \triangleq$$
$$\quad in_i.get > tk > process(tk) > r > (out_i.put(r) \mid W_i(in_i, out_i))$$

Server Interface S. The interface S distributes the *data* in sequence across the channels, *ch*, according to a *distribution* policy that can be substituted by the expression *broadcast, unicast*, or *DP*. The auxiliary expression *next* is used for synchronisation.

$$S(data, ch) \triangleq \ \mathsf{if}\ data = [] \gg 0$$
$$\quad \mid \mathsf{if}\ data \neq [] \gg distribution(head(data)) \gg S(tail(data), ch))$$

$$next(h_1, \ldots, h_N) \triangleq \ let\ 1$$

The *broadcast* sends each item of data to all of the workers.

$$broadcast(item) \triangleq \ next(h_1, \ldots, h_N)\ \mathsf{where}\ h_1 :\in ch_1.put(item)$$
$$\ldots$$
$$h_N :\in ch_N.put(item)$$

The *unicast* sends each item to a single worker $\mathsf{W}_{f(i)}$ where the index i is chosen in a list $[1 \ldots N]$. The function f is assumed to be stateful (e.g. successive calls to f can scan the list).

$$unicast(data) \triangleq \ ch_{f(i)}.put(x) \gg let\ 1$$

The *DP* describes the data-parallel scatter. Assume that *#data* is a multiple of N (for simplicity), and the *slice(data,i)* returns the i^{th} slice of *data*, where each slice is of length *#data*$/N$. The actual definition of "i^{th} slice" may vary, but is abstracted here in the function *slice*. For example, if the first slice corresponds to the first *#data*$/N$ items in *data*, etc. then the distribution is round-robin. In the specification given, the data is divided into N slices and each slice is sent on one of the channels.

$$DP(x) \triangleq \ next(h_1, \ldots, h_N)\ \mathsf{where}\ h_1 :\in ch_1.put(slice(x, 1))$$
$$\ldots$$
$$h_N :\in ch_N.put(slice(x, N))$$

Client interface C. Here interface C receives an item from each worker W_i along channel ch_i and, when it has an item from every worker, applies a *collection* policy. We exemplify here *reduce* and *select* policies.

$$C(ch) \triangleq collection(ch) \gg C(ch)$$

The *reduce* function may take an average, select the median, etc. (Note; it is assumed here that all workers supply results; otherwise timeouts could be used to avoid starvation.)

$$collection(ch) \triangleq reduce(h_1, \ldots, h_N) \text{ where } h_1 :\in ch_1.get \quad \ldots \quad h_N :\in ch_N.get$$

Alternatively, the port C may non-deterministically *select* a single data item from one worker and discard the rest.

$$collection(ch) \triangleq select(r) \text{ where } r :\in (|\ i : 1 \leq i \leq N : ch_i.get)$$

In all of the presented cases it is easy to verify that the functional behaviour is independent of N, provided W is stateless. Therefore, all management policies that change the value of N do not alter the functional behaviour, and can thus be considered correct.

7. Conclusion

The challenge of autonomicity in the context of component-based development of grid software is substantial. Building into components autonomic capability typically impairs their reusability. We have proposed behavioural skeletons as a compromise: being skeletons they support reuse, while their parametrisation allows the controlled adaptation needed to achieve dynamic adjustment of QoS while preserving functionality. We have presented a significant set of skeletons, together with their formal Orc functional behaviour description and self-management strategies. We have described how these concepts can be applied and implemented within the GCM. The presented behavioural skeletons have been implemented in GCM-ProActive [6], in the framework of the Grid-COMP project[2] and are currently under experimental evaluation. Preliminary results, not presented in this work, confirm the feasibility of the approach.

References

[1] M. Aldinucci and M. Danelutto. Algorithmic skeletons meeting grids. *Parallel Computing*, 32(7):449–462, 2006. DOI:10.1016/j.parco.2006.04.001.

[2] M. Aldinucci, M. Danelutto, and P. Kilpatrick. Management in distributed systems: a semi-formal approach. In A.-M. Kermarrec, L. Bougé, and T. Priol, editors, *Proc. of 13th Intl. Euro-Par 2007*, LNCS, Rennes, France, Aug. 2007. Springer. To appear.

[2] `http://gridcomp.ercim.org`, an EU STREP project aimed at providing an open source, reference implementation of the GCM.

[3] M. Aldinucci, M. Danelutto, and M. Vanneschi. Autonomic QoS in ASSIST grid-aware components. In B. D. Martino and S. Venticinque, editors, *Proc. of Intl. Euromicro PDP 2006: Parallel Distributed and network-based Processing*, pages 221–230, Montbéliard, France, Feb. 2006. IEEE.

[4] F. André, J. Buisson, and J.-L. Pazat. Dynamic adaptation of parallel codes: toward self-adaptable components for the Grid. In *Proc. of the Intl. Workshop on Component Models and Systems for Grid Applications*, CoreGRID series. Springer, Jan. 2005.

[5] A. Andrzejak, A. Reinefeld, F. Schintke, and T. Schütt. On adaptability in grid systems. In *Future Generation Grids*, CoreGRID series. Springer, Nov. 2005.

[6] F. Baude, D. Caromel, and M. Morel. On hierarchical, parallel and distributed components for grid programming. In V. Getov and T. Kielmann, editors, *Proc. of the Intl. Workshop on Component Models and Systems for Grid Applications*, CoreGRID series, pages 97–108, Saint-Malo, France, Jan. 2005. Springer.

[7] J. Bosch. Superimposition: a component adaptation technique. *Information & Software Technology*, 41(5):257–273, 1999.

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

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

[10] P.-C. David and T. Ledoux. An aspect-oriented approach for developing self-adaptive fractal components. In W. Löwe and M. Südholt, editors, *Proc of the 5th Intl Symposium Software on Composition (SC 2006)*, volume 4089 of *LNCS*, pages 82–97, Vienna, Austria, Mar. 2006. Springer.

[11] S. Gorlatch and J. Dünnweber. From grid middleware to grid applications: Bridging the gap with HOCs. In V. Getov, D. Laforenza, and A. Reinefeld, editors, *Future Generation Grids*, CoreGRID series. Springer, Nov. 2005.

[12] K. Kennedy et al. Toward a framework for preparing and executing adaptive Grid programs. In *Proc. of NSF Next Generation Systems Program Workshop (IPDPS)*, 2002.

[13] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[14] J. Misra and W. R. Cook. Computation orchestration: A basis for a wide-area computing. *Software and Systems Modeling*, 2006. DOI 10.1007/s10270-006-0012-1.

[15] Next Generation GRIDs Expert Group. *NGG3, Future for European Grids: GRIDs and Service Oriented Knowledge Utilities. Vision and Research Directions 2010 and Beyond*, Jan. 2006.

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

[17] M. Parashar, H. Liu, Z. Li, V. Matossian, C. Schmidt, G. Zhang, and S. Hariri. AutoMate: Enabling autonomic applications on the Grid. *Cluster Computing*, 9(2):161–174, 2006.

[18] S. M. Sadjadi and P. K. McKinley. Transparent self-optimization in existing CORBA applications. In *Proc. of the 1st Intl. Conference on Autonomic Computing (ICAC'04)*, pages 88–95, Washington, DC, USA, 2004. IEEE.

[19] E. Truyen, B. Jørgensen, W. Joosen, and P. Verbaeten. On interaction refinement in middleware. In J. Bosch, C. Szyperski, and W. Weck, editors, *Proc. of the 5th Intl. Workshop on Component-Oriented Programming*, pages 56–62, 2001.

[20] M. Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12):1709–1732, Dec. 2002.