TOWARDS A FORMAL SEMANTICS FOR AUTONOMIC COMPONENTS

Marco Aldinucci
Department of Computer Science, University of Pisa, Italy
aldinuc@di.unipi.it

Emilio Tuosto
Department of Computer Science, University of Leicester, UK
et52@mcs.le.ac.uk

Abstract Autonomic management can improve the QoS provided by parallel/distributed applications. Within the CoreGRID Component Model, the autonomic management is tailored to the automatic – monitoring-driven – alteration of the component assembly and, therefore, is defined as the effect of (distributed) management code.

This work yields a semantics based on *hypergraph* rewriting suitable to model the dynamic evolution and non-functional aspects of Service Oriented Architectures and component-based autonomic applications. In this regard, our main goal is to provide a formal description of adaptation operations that are typically only informally specified. We advocate that our approach makes easier to raise the level of abstraction of management code in autonomic and adaptive applications.

Keywords: Components, adaptive applications, autonomic computing, grid, semantics, graph rewriting.

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

1. Introduction

Developers of grid applications cannot rely neither on fixed target platforms nor on stability of their status [14]. This makes dynamic adaptivity of applications an essential feature in order to achieve user-defined levels of Quality of Service (QoS). In this regard, component technology has gained increased impetus in the grid community for its ability to provide a clear separation of concerns between application logic and QoS-driven adaptation, which can also be achieved *autonomically*. As an example, GCM (the Grid Component Model defined within the CoreGRID NoE) is a hierarchical component model explicitly designed to support component-based autonomic applications in highly dynamic and heterogeneous distributed platforms [7].

An assembly of components may be naturally modelled as a graph and, if components are autonomic, the graph can vary along with the program execution and may change according to input data and/or grid hardware status. These changes can be encoded as reaction rules within the component *Autonomic Manager* (hereafter denoted as AM). A proper encoding of these rules effectively realises the management policy, which can be specific of a given assembly or pre-defined for parametric assemblies (such as *behavioural skeletons*) [2, 1]. In any case, the management plan relies on the reconfiguration operation exposed by the component model run-time support.

A major weakness of current component models (including GCM) is that the semantics of these operations are informally specified, thus making hard to reason about QoS-related management of components. In this work

- We introduce few operations useful for component adaptation; the chosen operations are able to capture typical adaptation patterns in parallel/distributed application on top of the grid. These are presented as *non-functional interfaces* of components that trigger component assembly adaptation (Sec. 2).
- We detail a semantics for these operations based on *hypergraph* rewriting suitable for the description of component concurrent semantics and the run-time evolution of assemblies of autonomic components along adaptations (Sec. 3, 4, and 5).
- We discuss the appropriateness of the level of abstraction chosen to describe adaptation operations to support the design of component-based applications and their autonomic management (Sec. 6).

The key idea of our semantical model consists in modelling component-based applications by means of *hypergraphs* which generalise usual graphs be allowing *hyperedges*, namely arcs that can connect more than two nodes. Intuitively, hyperedges represent components able to interact through *ports* represented

by nodes of hypergraphs. The *Synchronised Hyperedge Replacement* (SHR) model specifies how hypergraphs are rewritten according to a set of *productions*. Basically, rewritings represent adaptation of applications possibly triggered by the underlying grid middleware events (or by the applications themselves).

SHR has been shown suitable for modelling non-functional aspects of service oriented computing [10–11] and is one of the modelling and theoretical tools of the SENSORIA project [20]. For simplicity, we consider a simplified version of SHR where node fusion is limited and restriction is not considered. Even if, for the sake of simpleness, the SHR framework used in this work is not the most general available, it is sufficient to give semantics to the management primitives (aka adaptation operations) addressed here. The autonomic manager – by way of these adaptation operations – can structurally reconfigure an application to pursue the (statically or dynamically specified) user intentions in terms of QoS.

2. Autonomic Components and GCM

Autonomic 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. These systems typically rely on a clear separation of concerns between adaptation and application logic [15]. 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.

The idea of autonomic management of parallel/distributed/grid applications is present in several programming frameworks, although in different flavours: ASSIST [22, 3], AutoMate [18], SAFRAN [9], and GCM [7] all include autonomic management features. The latter two are derived from a common ancestor, i.e. the Fractal hierarchical component model [17]. All the named frameworks, except SAFRAN, are targeted to distributed applications on grids.

GCM builds on the Fractal component model [17] and exhibits three prominent features: hierarchical composition, collective interactions and autonomic management. GCM components 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 AM, interacting with other man-

agers in other components via the component non-functional interfaces. The AM implements the autonomic cycle via a simple program based on reactive rules. These rules are typically specified as a collection of `when-`*event*`-if-`*cond*`-then-`*adapt_op* clauses, where *event* is raised by the monitoring of component internal or external 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 component internal attributes (e.g. component life-cycle status); *adapt_op* represents an adaptation operation (e.g. create, destroy a component, wire, unwire components, notify events to another component's manager) [9].

We informally describe some common adaptation operations that may be assigned to configuration interfaces are the following:

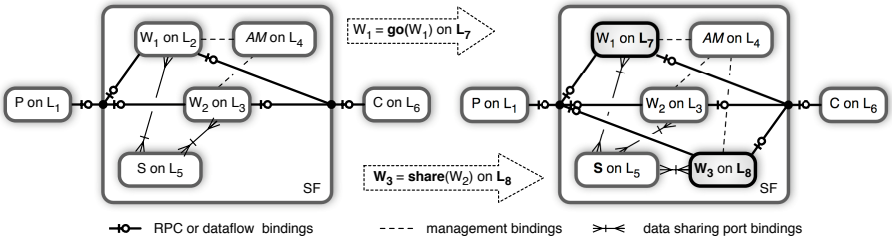
Migration A component is required to change its running location (e.g. platform, site). The request must include the new location and can be performed while keeping its attached external state (**go**) or restating from a fresh default state (**start**).

Replication A component (either composite or primitive) is replicated. Replication operation is particularly targeted to composite components exhibiting the parametric replication of inner components (such as behaviour skeletons), and can be used to change their parallelism degree (and thus their performance and fault-tolerance properties). Replication events are further characterized with respect to their relation with replicated component state, if any. A component replica may be created with a fresh external state, carry a copy of the external state (**copy**), or share the external state with the source component (**share**).

Kill A component is killed. Due to this kind of action disconnected components (and in particular storage managers) can subject to garbage collection.

Described primitives make possible the implementation of several adaptation paradigms. In particular, migration may be used to adapt the application to changes of grid topology as well as to performance drop of resources. Replication and kill may be used to adapt both data and task parallel computation. In particular, replication with share makes it possible the redistribution of sub-task in data parallel computations; replication with copy enables hot-redundancy. Both stateful and stateless farm computation (parameter-sweeping, embarrassingly parallel) may be reshaped both in parallelism degree and location run by using replication and kill.

EXAMPLE 1 *Let* P, C, SF, S, AM, W_1, W_2, W_3 *components (Producer, Consumer, Stateful Farm*[1]*, Storage, Autonomic Manager, and Workers);* $\mathrm{L}_1 \cdots \mathrm{L}_8$ *locations (e.g. sites, platforms). Thee kinds of bindings are used in the assembly (see also Sec. 4).*



The described assembly of components (left) is paradigmatic of many producer-filter-consumer applications, where the producer (P) generates a stream of data and the filter is parallel component (SF) exhibiting a shared state among its inner components (e.g. a database). The original assembly (left) can be dynamically adapted (right) by way of two adaptation operations to react to runtime events, such as a request of increasing the throughput. The **go** *operation moves* W_1 *from* L_2 *to* L_7 *(as an example to move a component onto a more powerful machine); the* **share** *operation that replicates* W_2 *and place it in the new location* L_8 *(to increase the parallelism degree). Both operations preserve the external state of the migrated/replicated component, which is realised by way of a storage component) attached via a data sharing interface [4].*

Example 1 illustrates how the management can be described from a *global viewpoint*. Indeed, the system is described by in a rather detailed way, e.g., components are explicitly enumerated along with their connections. Even if this global viewpoint is useful (and sometime unavoidable) when designing distributed systems, it falls short in describing what single components are supposed to do when a reconfiguration is required. In other terms, it is hard to tell what the *local* behaviour of each component should be in order to obtain the reconfiguration described by the *global* view.

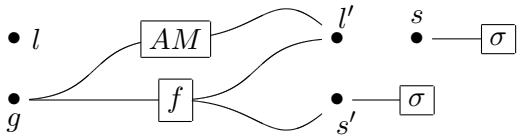
Also, it is worth remarking that, though the diagram clearly describes the changes triggered by AM in this scenario, the lack of a formal semantics leaves some ambiguities. For example, it is not clear if the reconfiguration should take place if, and only if, the system is configured as on the lhs or this is rather a "template" configuration (e.g., should the system reconfigure itself also when W_2 is connected to W_1? What if W_2 was not present?). Of course, such ambiguous situations can be avoided when a formal semantics is adopted.

[1] This component is a composite component, and in particular it is an instance of a behavioural skeleton [2].

3. A Walk through SHR

Synchronised Hyperedge Replacement (SHR) can be thought of as a rule-based framework for modelling (various aspects of) of distributed computing [11] modelled as *hypergraphs*, a generalisation of graphs roughly representing (sets of) relations among nodes. While graphs represent (sets of) binary relations (labelled arcs connect exactly two nodes), labelled *hyperedges* (hereafter, edges) can connect any number of nodes. We give an informal albeit precise description of *hypergraphs* and SHR through a suitable graphical notation. The interested reader is referred to [11, 16] and references therein for the technical details.

EXAMPLE 2 *In our graphical notation, a hypergraph is depicted as*



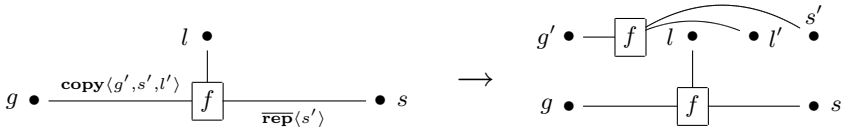
Edges (labelled by f, AM and σ) are connected to nodes (g, l, l', s and s'). Specifically, AM connects g and l', f connects g, l' and s' while two σ-labelled edges are attached to s and s'. Notice that nodes can be isolated (e.g., l).

Hyperedges represent (distributed) components that interact through *ports* represented by nodes. Connections between edges and nodes, called *tentacles*, allow components sharing ports to interact (e.g., in Example 2, f and AM can interact on g and on l').

EXAMPLE 3 *The hypergraph in Example 2 represents (part of) a system where a manager AM and a component f are located at l' and can interact on port g. The component f has access to the store at s' (e.g. by way of a data port [4]). In the system are also present another location l and store s.*

As in string grammars, SHR rewriting is driven by *productions*. In fact, strings can be rewritten according to a set of *productions*, i.e. rules of the form $\alpha \longrightarrow \beta$, where α and β are strings (over fixed alphabets of terminal and non-terminal symbols). Similarly, in SHR hypergraph rewritings are specified by productions of the form $L \longrightarrow R$, where the lhs L is a hyperedge, the rhs R is a hypergraphs and states that occurrences of L can be replaced with R. Intuitively, edges correspond to non-terminals and can be replaced with a hypergraph according to their productions. In SHR, hypergraphs are rewritten by *synchronising* productions, namely edge replacement is *synchronised*: to apply the productions of edges sharing nodes, some conditions must be fulfilled.

More precisely, an SHR production can be represented as follows:



where on the lhs is a decorated edge and on the rhs a hypergraph. The production above should be read as a rewriting rule specifying that edge f on the lhs can be replaced with the hypergraph on the rhs provided that the conditions on the tentacles are fulfilled. More precisely, $\mathbf{copy}$ and $\overline{\mathbf{rep}}$ must be satisfied on node g and s, respectively while f is *idle* on node l, namely it does not pose any condition on l. According to our interpretation, this amounts to say that when component f is said to replicate with copy by its AM (condition $\mathbf{copy}$ on node g), it tells its store to duplicate itself (condition $\mathbf{rep}$ on node s). When such conditions are fulfilled, edge f is replaced with the hypergraph on the rhs which yield two instances of f one of which connected to the communicated nodes as prescribed by the rhs of the production. Indeed, f exposes three nodes on condition $\mathbf{copy}$ and one on $\mathbf{rep}$; these represent nodes that are communicated, i.e. g and l are node communication accounts for mobility as edges can dynamically detach their tentacles from nodes and connect them elsewhere.

SHR has a declarative flavour because programmers specify synchronisation conditions of components independently from each other. Once the system is built (by opportunely connecting its components) it will evolve according to the possible synchronisations of the edges. Global transitions are obtained by parallel application of productions with "compatible" conditions where compatibility depends on the chosen synchronisation policy[2]. Conditions on $L \longrightarrow R$ make it possible to introduce the concept of *"context-freeness"*: the productions with a left-hand-side (lhs) which is either a node or an edge confer a context-free flavour to graph grammars. Indeed, such productions do not consider the "surroundings" of their lhs. This makes it possible to design graph rewritings that can be *locally* applied, whereas other graph rewriting mechanisms (such as double-pushout) requires to be applied in a context, which may the be in the worst case the entire graph [11, 16]. As we shall discuss in Sec. 6, the context-freeness of the approach is one of key features making SHR well-suited to describe autonomic component in a grid framework.

[2]SHR is parametric with respect to the synchronisation mechanism adopted and can even encompass several synchronisation mechanisms.

4. Productions for Non-functional Interfaces

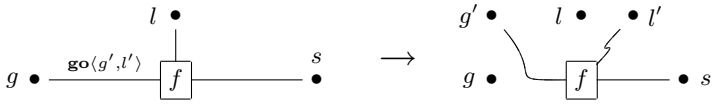
SHR can adequately formalise the non-functional interface mechanisms informally described in Sec. 2. Three conceptually distinct interfaces can be considered: i) interfaces between components and AM (for management non-functional bindings); ii) interfaces toward the external state (for data sharing functional bindings); iii) interfaces for communicating with other components (for RPC/dataflow functional bindings).

Since interfaces iii are application dependent, we focus on the coordination-related interfaces i and ii.

A main advantage of our approach is that all aspects of non-functional interfaces are captured in a uniform framework based on SHR. Indeed,

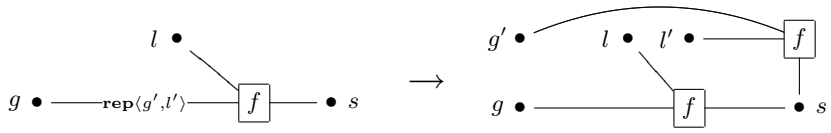
- components are abstracted as edges connected to form a hypergraph;
- the coordination interface of each component is separately declared and is not mingled with its computational activity;
- being SHR a *local* rewriting mechanism, it is possible to specify confined re-configuration of systems triggered by *local* conditions;

Migration. The migration of a component f is triggered when its AM raises a signal **go** with the new location on node g. The synchronisation of f on the **go** signal is given by following production:



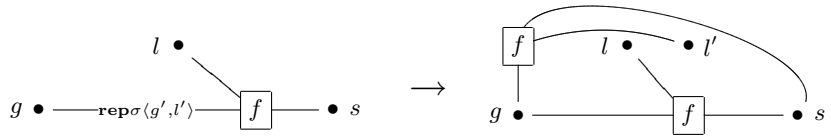
specifying that f running at l accepts to migrate to l' (lhs); the "location" tentacle of f is disconnected from l and attached to l' (rhs). Notice that f maintains the connection to the previous state s and l is still present. The tentacle connected to g on the lhs is connected to g' on the rhs; however, it might well be that $g' = g$ (f is still connected to the original AM) or $g \neq g'$ (f changes manager). Similarly, **start** moves the component to a new location l'. However, a new external state σ is created together with its attaching node:

Replication. Unlike migration, replication of f preserves the location of the original edge (i.e. a component):



the effect of the above production is to add a new instance of f at l' with AM connected to g'; of course, $l = l'$ and $g = g'$ are possible. The newly generated instance shares external state with the original one.

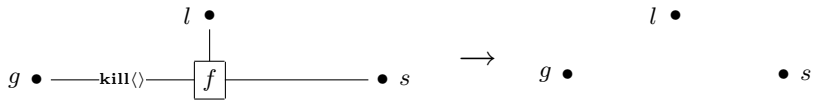
Replication can also activate the new instance with a different state:



The production above creates a fresh replica of f at l' and assigns to it the manager at g'; notice that the two instances of f share the state s.

Replication can also trigger a new instance of f that acts on a copy of the state original state as described in the production of page 7 where f must notify to its state to duplicate itself and connect the new copy on s'. Hence, the state connected to s duplicate itself on the node s' when the action complementary to $\overline{\mathbf{rep}}$ is received, as stated below.

Component killing. Components are killed using the following production:



stating that f disappears when its corresponding AM sends a **kill** signal.

5. Synchronising productions

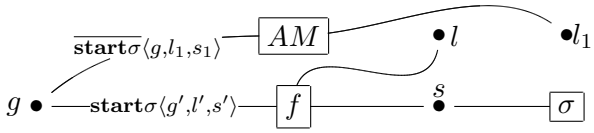
The operational semantics of SHR is illustrated through an example that highlights the following steps:

1 individuate the adjacent tentacles labelled by compatible conditions;

2 determine the synchronising productions and replace the (instances of) edges on their lhs with the hypergraphs on their rhs;

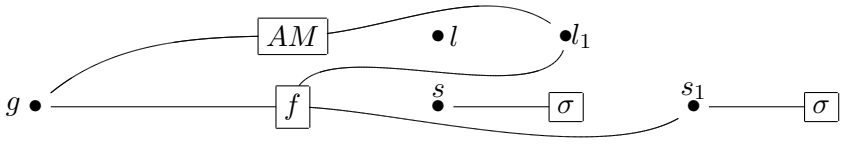
3 fuse the nodes that are equated by the synchronisations.

Let us apply the previous steps to show how migration works in a situation represented by the following hypergraph



where component f is running at l and shares g with a manager AM located at l_1. For brevity, tentacles are decorated with the conditions triggering the rewriting (step 1). Indeed, the tentacles of AM and of f incident on node g yield compatible output and input conditions respectively so that AM orders f to migrate to l_1 and to use the store at s_1 while staying connected to g.

Productions synchronisation consists in replacing the occurrences of the edges on the lhs with the hypergraphs specified in the rhs of the productions and applying the node fusions obtained by the node communicated. For instance, in the previous example the synchronising productions are the $\mathbf{start}\sigma$ production of f given in Sec. 4 and the production of AM whose lhs and rhs consist of AM connected to g and l_1 (step 2). Hence, after the synchronisation, the node fusions $g' = g$, $l' = l_1$ and $s' = s_1$ are applied (step 3), so that the hypergraph is rewritten as



Let us remark that l, σ and s remain in the final hypergraph. In fact they should not be removed because other edges can be allocated on l or access σ.

The intuitive description of SHR given in this section suggests the following design style and execution style:

- assign an edge to each component and specify their productions;
- represent the system as a hypergraph;
- decorate the tentacles with the synchronisation conditions;
- synchronise the productions until possible.

It is worth remarking that, unlike other semantical frameworks (e.g., process calculi), in SHR synchronisation conditions may require more than two (productions of) components to be synchronised. This actually depends on the synchronisation policy at hand. For instance, in the migration rewriting described in this section, it is possible to use broadcast interactions on the node g so that *all* the components connected on g will move at l' when the productions are synchronised.

6. SHR provides a suitable abstraction for GCM Managers

We envision the GCM applications as composed of assemblies autonomic components. These components are *locally* managed by their own AM, whereas the *global* managing of the application is distributely realised via the cooperation of all AMs. This cooperation may happen in different fashions, although an arrangement in a hierarchical fashion appears quite natural for GCM applications due to the hierarchic nature of the model [7].

Irrespectively of any given schema chosen for managing orchestration, each manager can be described in terms of the adaptations that it can locally induce, and the coordination actions it can handle towards other managers. Observe, however, that the ultimate nature of those coordination actions consist in give rise to a broader adaptation involving (also) not directly managed components[3]. As discussed in Sec.3, SHR enables the system designer to uniformly formalise and encode *adapt_op* as local rules in the AM. These rules may

- drive the adaptation of directly managed components, via the synchronisation with nodes included in the managed (composite) component, such as **go**, **start**, **rep**, **copy**, and **kill**;
- drive broader adaptations via the synchronisation with other AMs. The formalisation of these rules is currently under investigation and it not fully discussed in this work. Preliminary results suggest the feasibility of a design based on just two rules for interaction among managers: a

[3] Indeed, both classes of operations have been denoted as *adapt_op* (see Sec. 2).

rule to (dynamically) send a new set of rules to other AMs, and a rule to raise exception/violation toward other AMs.

An implementation of GCM exploiting described principles is currently ongoing. The feasibility of the approach has been prototyped with SCA/Tuscany [19, 21] leveraging on a JBoss-based [13] encoding of `when-`*event*`-if-`*cond*`-then-`*adapt_op* rules. [8]. This kind of encoding makes easy the serialisation of rules to support their portability across different AMs.

A distinguished feature of our approach is the high level of abstraction that can be achieved through SHR formalisation of adaptation operations. This results in:

- The possibility to model very different attributes related to QoS management. As an example, in the previous sections we uniformly used nodes of the graphs to model locations, storage ports, functional ports, and non-functional ports. The concept can be easily and uniformly extended to cover other attributes that may be of interest of a particular instance of the model, inter alia attributes concerning security, robustness, and platform configuration.
- The possibility to describe autonomic behaviour irrespectively of any particular implementation. This is mostly due to the neutrality of the description with respect to lower level detail of the component model behaviour, inter alia component life-cycle, interactions between functional and non-functional ports. As an example, the proposed description of adaptation operation is suitable for GCM/P [2], ASSIST [3], and SCA/Tuscany [8] implementation of autonomic components.

In regard to the latter point, observe that our approach substantially differs from other formalisation efforts aiming to model and check a particular implementation of an adaptive component framework (such as [6]). In particular, the SHR description cannot be directly checked before being mapped onto a concrete model. We believe, however, this is a strength of the approach rather than a limitation. On the one hand, because the concrete model can be automatically generated through compilation once implementation-specific details has been fixed, whereas in other approaches the model is entirely manually designed. On the other hand, because it can support different concrete models matching different implementations.

EXAMPLE 4 *The reference implementation of GCM (GCM/P [12], developed on top of the Proactive middleware [5]) and ASSIST [3] exploit slightly different autonomic component models and substantially different implementations.* i) *ASSIST is implemented in C++ whereas GCM/P in Java.* ii) *ASSIST does not require a component subject of a* **copy** *to be in stopped state, whereas GCM/P*

does. iii) ASSIST implements **kill** *as component destruction, whereas GCM/P as logical marking. iv) ASSIST provides a native Distributed Shared Memory for external storage in* **share***, whereas GCM/P does not. v) ASSIST does not implements* **go** *for all components, whereas GCM/P does. However, they both implement the same set of adaptation operations, which is the one described in the previous sections. As expected, the same operation exhibits different limitations and overheads in the two implementations*[4].

Finally, observe that proposed approach to formalisation of autonomic components slightly extends classic (run-time) autonomic approach. We believe that the GCM equipped with those adaptation operations make it possible the definition of a malleable component model in which adaptations may be either applied autonomically at run-time (under the control of the AMs) or statically exploited to achieve static or launch-time optimisation targeted to generate/configure a particular component assembly for a well-known running environment (e.g. a cluster), thus potentially achieving a significant reduction of overhead in the running code while keeping the full ubiquity potential of the GCM applications.

7. Conclusions

In this work we introduced a SHR formalisation of adaptation operations suitable to support the definition and the evolution autonomic components, and in particular GCM-based autonomic components, which has been defined within the CoreGRID NoE.

A reference implementation of GCM (GCM/P) autonomic components is currently ongoing within the GridCOMP STREP project [12]. In this implementation, the autonomic manager of a component is currently defined as a chunk of plain Java code (wrapped into a proper placeholder) invoking monitor and adaptation operations. This approach, despite already fully functional [2], is excessively low-level and implementation-dependent, thus is unlikely to properly support the design of management for large/complex component assemblies, to sustain the design of reusable management policies, and to survive to the porting of these policies to other implementations of the same (or similar) component models.

The proposed formalisation aims to raise the level of abstraction of adaptation operations and their effects (i.e. their semantics), thus providing

- the application designer with a theoretical tool to design management policies, and reason about their effects (effectiveness, correctness, etc.);

[4] On the whole, GCM/P focuses on generality whereas ASSIST on performance [3, 2].

- the component model developers with a formal specification of adaptation operations as reference for their implementation and manipulation (parsing, serialisation, dynamic installation, etc.)

SHR has been previously exploited in [10] for managing application level *service level agreement* (SLA) in a distributed environment, and in [11] to tackle several programming and modelling facets arising in service oriented computing. Here, we shown that SHR is a suitable tool to describe adaptation operations at the "proper" level of abstraction, thus making possible to achieve

- the uniform description of the attributes involved in component assembly adaptation (such as location, storage ports, etc.);
- describe adaptations at the level of the component model (as opposed to its implementation);
- the design of effective and reusable autonomic management policies.

The presented adaptation operations are currently implemented (as Java code) in GCM/P; their effectiveness and overhead in managing the QoS of grid applications is discussed in [1–2].

Acknowledgments

This research has been supported by the FP6 Network of Excellence CoreGRID, the FP6 GridCOMP project funded by the European Commission (IST-2002-004265 and FP6-034442), and the SENSORIA project funded by the European Commission (FET-GC II IST-2005-16004).

References

[1] M. Aldinucci, S. Campa, M. Danelutto, P. Dazzi, P. Kilpatrick, D. Laforenza, and N. Tonellotto. Behavioural skeletons for component autonomic management on grids. In M. Danelutto, P. Frangopoulou, and V. Getov, editors, *Making Grids Work*, CoreGRID. Springer, May 2008.

[2] 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 D. E. Baz, J. Bourgeois, and F. Spies, editors, *Proc. of Intl. Euromicro PDP 2008: Parallel Distributed and network-based Processing*, pages 54–63, Toulouse, France, Feb. 2008. IEEE.

[3] M. Aldinucci and M. Danelutto. Algorithmic skeletons meeting grids. *Parallel Computing*, 32(7):449–462, 2006.

[4] G. Antoniu, H. Bouziane, L. Breuil, M. Jan, and C. Pérez. Enabling transparent data sharing in component models. In *6th IEEE Intl. Symposium on Cluster Computing and the Grid (CCGRID)*, pages 430–433, Singapore, May 2006.

[5] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, Jan. 2006.

[6] T. Barros, L. Henrio, and E. Madelaine. Behavioural models for hierarchical components. In P. Godefroid, editor, *Model Checking Software, Proc. of the 12th Intl. SPIN Workshop*, volume 3639 of *LNCS*, pages 154–168, San Francisco, CA, USA, Aug. 2005. Springer.

[7] CoreGRID NoE deliverable series, Institute on Programming Model. *Deliverable 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`.

[8] M. Danelutto and G. Zoppi. Behavioural skeletons meeting services. In *Proc. of ICCS: Intl. Conference on Computational Science, Workshop on Practical Aspects of High-level Parallel Programming*, volume 5101 of *LNCS*, pages 146–153, Krakow, Poland, June 2008. Springer.

[9] 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.

[10] R. De Nicola, G. Ferrari, U. Montanari, R. Pugliese, and E. Tuosto. A formal basis for reasoning on programmable Qos. In *Intl. Symposium on Verification – Theory and Practice – Honoring Z. Manna's 64th Birthday*, volume 2772 of *LNCS*. Springer, June 2003.

[11] G. Ferrari, D. Hirsch, I. Lanese, U. Montanari, and E. Tuosto. Synchronised hyperedge replacement as a model for service oriented computing. In F. de Boer, M. Bonsangue, S. Graf, and W. de Roever, editors, *Formal Methods for Components and Objects: 4th Intl. Symposium, FMCO*, volume 4111 of *LNCS*, Amsterdam, The Netherlands, Nov. 2006. Springer. Revised Lectures.

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

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

[14] K. Kennedy, M. Mazina, J. Mellor-Crummey, K. Cooper, L. Torczon, F. Berman, A. Chien, H. Dail, O. Sievert, D. Angulo, I. Foster, D. Gannon, L. Johnsson, C. Kesselman, R. Aydt, D. Reed, J. Dongarra, S. Vadhiyar, and R. Wolski. Toward a framework for preparing and executing adaptive Grid programs. In *Proc. of NSF Next Generation Systems Program Workshop (IPDPS 2002)*, 2002.

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

[16] I. Lanese and E. Tuosto. Synchronized Hyperedge Replacement for Heterogeneous Systems. In J. Jacquet and G. Picco, editors, *International Conference on Coordination Models and Languages*, volume 3454 of *LNCS*, pages 220 – 235. Springer, April 2005.

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

[18] 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.

[19] Service component architecture. `http://www.ibm.com/developerworks/library/specification/ws-sca/`, 2008.

[20] Sensoria Project. Software Engineering for Service-Oriented Overlay Computers, 2008. `http://sensoria.fast.de/`.

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

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