INTEGRATING APPLICATION AND SYSTEM COMPONENTS WITH THE GRID COMPONENT MODEL

Michal Ejdys, Ula Herman-Izycka, Namita Lal, Thilo Kielmann*
Vrije Universiteit
Dept. of Computer Science
De Boelelaan 1083
1081HV Amsterdam, The Netherlands
kielmann@cs.vu.nl

Enric Tejedor, Rosa M. Badia
Univ. Politècnica de Catalunya
C/ Jordi Girona, 1–3
E-08034 Barcelona, Spain

Abstract The Grid Component Model (GCM) is becoming a promising development platform for flexible and adaptable grid applications. Recently, a set of mediator components has been proposed for providing a uniform and integrated platform to access grid middleware, services, and resources from an application. In this paper, we present our experiences with building such mediator components using GCM, focusing on two functionality areas. First, we show how application adaption support can be realized via mediator components, based on a set of component controllers through which the application components can be adapted and steered. Second, we show how a service and resource abstraction layer can be controlled at runtime from the mediator components.

Keywords: Grid Component Model (GCM), Mediator Components, Grid Application Toolkit (GAT)

*Contact author.

1. Introduction

Developing grid applications has proven to be a hard problem. What distinguishes grids from other environments is their heterogeneity, dynamic variability of resource quality, and their non-negligible failure rates, in their totality requiring approaches to application development that take these non-functional properties into account [14].

Many grid programming models have been proposed. Component models like CCA [4]or Fractal [6]provide the flexibility that is needed to address the challenges of grid programming. The Grid Component Model (GCM) [8]is becoming a promising development platform for flexible and adaptable grid applications. Recently, an integrated toolkit for grid applications has been proposed [9], using both a set of mediator components and a service and resource abstraction layer to integrate GCM-based applications with grid middleware environments.

In this paper, we present our experiences with building such mediator components using GCM, focusing on two functionality areas. First, we show how application adaption support can be realized via mediator components, based on a set of component controllers through which the application components can be adapted and steered. Second, we show how a service and resource abstraction layer can be controlled at runtime from the mediator components. We show how both GCM-aware and GCM-unaware applications can be used with our mediator component toolkit.

The remainder of this paper is organized as follows. Section 2 and Section 3 briefly present the GCM component model, and survey the integrated toolkit, respectively. Section 4 describes a more detailed design for the integration of the mediator components with application components. In Section 5, we describe how GCM components can be used to dynamically adapt the service and resource abstraction layer, too, shown on the example of the JavaGAT [16]implementation. Section 6 discusses related work. Section 7 concludes.

2. The Grid Component Model (GCM)

GCM allows applications to be written in a way that they can cope with the specific requirements of grid environments, most prominently resource heterogeneity, performance variability, and fluctuating availability. GCM is addressing these issues by the following properties.

First, GCM is a *hierarchical* component model. This means that users of GCM (programmers) have the possibility of building new GCM components as compositions of existing GCM components. The new, composite components programmed in this way are first class components, in that they can be used in every context where non-composite, elementary components can be used.

Programmers need not necessarily perceive these components as composite, unless they explicitly want to consider this feature.

GCM allows component interactions to take place with several distinct mechanisms. In addition to classical use/provide (or client/server) ports, GCM allows *data*, *stream* and *event ports* to be used in component interaction. Using data ports, components can express data sharing between components while preserving the ability to properly perform ad hoc optimization of the interaction among components sharing data. While stream ports can be easily emulated by classical use/provide ports, their explicit inclusion allows much more effective optimizations to be performed in the component run-time support (framework). Event ports may be used to provide asynchronous interaction capabilities to the component framework. Events can be subscribed and generated. Furthermore, events can be used just to synchronize components as well as to synchronize *and* to exchange data while the synchronization takes place.

Regarding collective interaction patterns, GCM supports several kinds of collective ports, including those supporting implementation of structured interaction between a single use port and multiple provide ports (multicast collective) and between multiple use ports and a single provide port (gathercast collective). The two parametric (and therefore customizable) interaction mechanisms allow the implementation of most (hopefully all) of the interesting collective interaction patterns deriving from the usage of composite (parallel) components.

GCM is intended to be used in grid contexts, that is in highly dynamic, heterogeneous and networked target architectures. GCM therefore provides several levels of *autonomic managers* in components, that take care of the *non-functional* features of the component programs. GCM components have thus two kind of interfaces: a functional one and a non-functional one. The functional interface includes all those ports contributing to the implementation of the functional features of the component, i.e. those features directly contributing to the computation of the result expected of the component. The non-functional interface comprises all those ports needed to support the com ponent manager activity in the implementation of the non-functional features, i.e. all those features contributing to the efficiency of the component in the achievement of the expected (functional) results but not directly involved in actual result computation. Each GCM component therefore contains one or more managers, interacting with other managers in other components via the component's non-functional interfaces and with the managers of the internal components of the same component using the mechanism provided by the GCM component implementation. Each component has a manager whose job it is to ensure efficient execution of the component on the target grid architecture.

3. The Mediator Component Toolkit

The goal of the mediator component toolkit is to integrate system-component capabilities into application code, achieving both steering of the application and performance adaptation by the application to achieve the most efficient execution on the available resources offered by the Grid.

By introducing such a set of components, resources and services in the Grid get integrated into one overall system with homogeneous component interfaces. The advantage of such a component system is that it abstracts from the many software architectures and technologies used underneath.

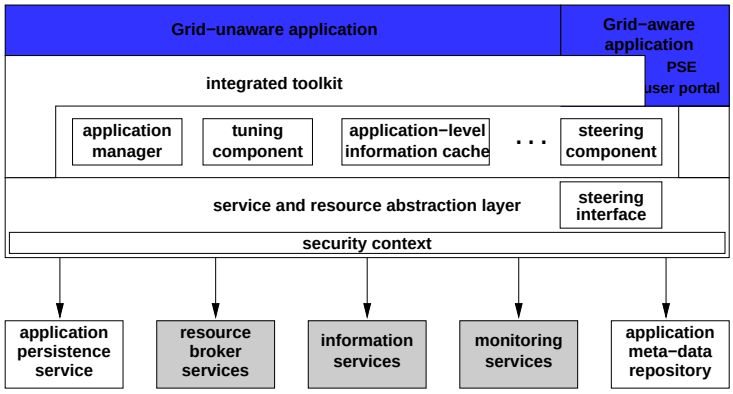

Figure 1. The generic component platform with mediator components, from [9].

The strength of such a component-based approach is that it provides a homogeneous set of well-defined (component-level) interfaces to and between all software systems in a Grid platform, ranging from portals and applications, via mediator components to the underlying system software. The set of envisioned mediator components, with their embedding in the generic component platform, can be seen in Figure 1; a detailed description can be found in [9]. We briefly summarize the mediator components in the following.

Application-level information cache

This component is supposed to provide a unified interface to deliver all kinds of meta-data (e.g., from a GIS, a monitoring system, from application-level meta data) to the application. Its purpose is twofold. First, it is supposed to provide a unifying component interface to all data (independent of its actual storage), including mechanisms for service and information discovery. Second, this application-level cache is supposed to deliver the information really fast, cutting down access times of current implementations like Globus GIS (up to multiple seconds) to the order of a method invocation. For the latter purpose, this component may have to

prefetch (poll) information from the various sources to provide them to the application in time. An implementation of such a component, albeit without a "real," e.g. GCM, component interface, has been described in [2].

Application steering and tuning components
Controlling and steering of applications by the user, e.g. via application managers, user portals, and problem solving environments (PSE's), requires a component-level interface to give external components access to the application. Besides the steering interface, also dedicated steering components are necessary, both for mediating between application and system components, but also for implementing pro-active steering systems, carrying their own threads of activity. The steering components thus provide a framework for performance tuning, which can be used to improve the execution time of applications automatically as well as for improving services and tools which are involved in the environment.

Application manager component
The envisioned application manager component establishes a pro-active user interface, in charge of tracking an application from submission to successful completion. Such an application manager is in charge of guaranteeing such successful completion in spite of temporary error conditions or performance limitations. (In other settings, such an active component might be referred to as an agent.) For performing its task, the application manager will need to interoperate with most of the other mediator components, like the application itself (via the steering interface), the application meta data repository and cache, as well as an application persistence service, like the one published in [15].

The components as described so far denote the core of the mediator set. In the course of ongoing work, this set is being refined and enriched as new experience will be gained.

4. Application Adaptation Support

The generic component platform, along with the mediator components, provides a platform for grid applications to adapt themselves to changing conditions and resources at runtime. In this section, we propose how to interface application components to this platform. We assume a parallel application that can adapt itself via its data distribution or by migration to other compute nodes. An example for such an application could be *Successive Over Relaxation* (SOR) which is based on nearest-neighbour communication. Applications with other communication patterns (like master/worker) would also be applicable. We also assume that the application shall be steered by its user.

For both adaptation by tuning and management components, as well as by the user via the steering component, the application components have to be called by the mediator components. For this purpose, we propose the interface shown in Figure 2, with specialized controllers that are added to the application components' membrane.

Based on experience gathered when investigating the GCM component framework, we propose the following extensions. First, in order to effectively modify the structure of a running application, we propose to implement an *explorer component*. Thus, the user could switch between different implementations of his/her algorithms without the need to stop and re-run their application.

Second, the mediator toolkit can greatly benefit from implementing different control aspects of the application separately, namely by using controllers. We propose to introduce into the architecture the following controllers, as depicted in Figure 2:

- steering – for modifying application parameters, which would allow for computational steering during runtime
- persistence – for handling checkpoints: initiating checkpoints, as well as starting (from checkpoint or from scratch) and stopping the application
- distribution – for optimal utilization of allocated resources, and for adapting to changes in environment (releasing and acquiring resources, changes in quality of network connections)
- component – for investigating the application's structure (in terms of components) and modifying it (e.g. switching to alternative implementation, replacing subcomponents)

Note that the component controller is already implemented in GCM. However, the other controllers have to be added according to the necessary functionality. Another important observation is that communication with the application is via its controllers only.

4.1 Persistence Controller and Life Cycle Controller

The *Persistence Controller* is the manager of an application instance. Not only is it responsible for checkpointing, but also for starting (from scratch or from a checkpoint) and stopping an application. For this to be accomplished, we propose bounding the Persistence Controller with GCM's *LifeCycleController*. The latter is a simple state automaton (with two states: *started* and *stopped*). We propose extending the state-cycle to service checkpointing. See Figure 3.

We propose to extend the *started* state of the component by adding substates representing different stages of the running application (*created*, *initialized*, *running*, and *finished*), and *checkpointing*.

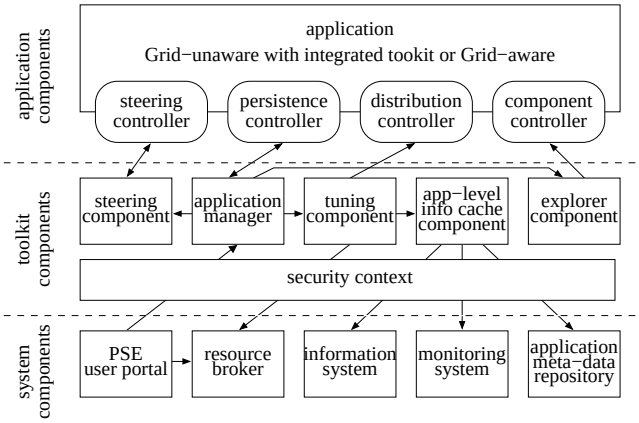

Figure 2. Generic component platform with application controllers.

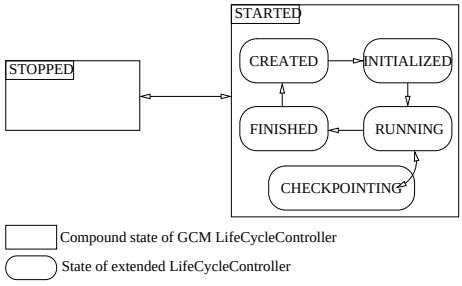

Figure 3. Extended states of LifeCycleController.

The GCM LifeCycleController is responsible for starting and stopping the component. There are certain conditions under which a component can be stopped. For example, all method invocations on this component should have finished (a special interceptor keeps a counter of active method invocations). Similarly, only a component with all mandatory interfaces bound can be started.

Our system also benefits from this approach. Transitions between *stopped* and *started* states are limited to only a few *started* substates. The component must not be allowed to stop while checkpointing is in progress. Additionally, stopping an application in the *running* state could mean interrupting the application (transition to *finished*) first.

4.2 Application controllers

The proposed application controllers (steering – *sc*, persistence – *pc*, distribution – *dc*, and component – *cc*) are implemented as GCM-controllers, part of the membrane, shown in Figure 4 (left). Alternatively, the controllers can also

be implemented as components inside a compound component, together with the application component itself, shown in Figure 4 (right). This design can be used for GCM-unaware applications as discussed below.

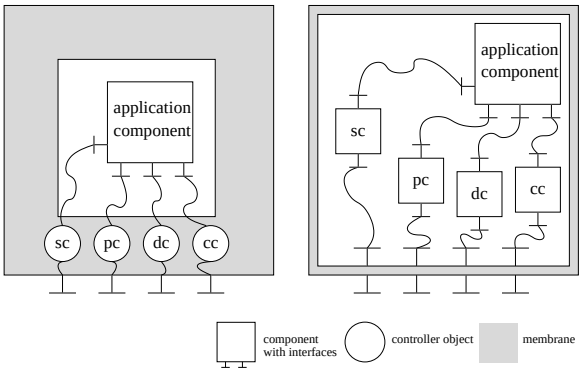

Figure 4. Controllers inside the component membrane (left) or as subcomponents (right).

4.3 Dealing with GCM-aware and unaware applications

The GCM Mediator Toolkit is ready to run not only with applications that have been developed with GCM in mind, but also with application *objects* (rather than components), from "legacy" applications, as shown in Figure 5.

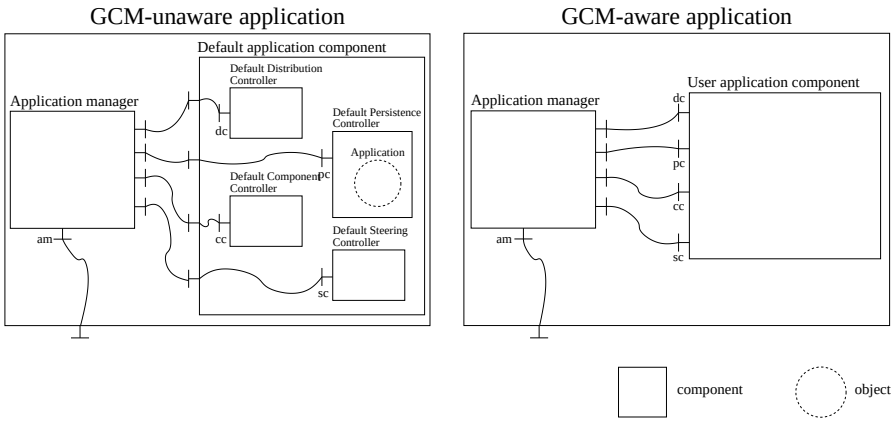

Figure 5. Integrating applications with mediator controllers.

GCM-unaware applications. The framework is able to cooperate with applications that do not use the GCM framework, shown in Figure 5, left. In that case, a set of default controllers is created. A user application is encapsulated by

the default persistence controller as this component is responsible for starting and stopping the application, and it has direct access to the application object. This controller, together with the default distribution, component, and steering controllers, are integrated to a default application component, which is bound to the rest of the framework via the application manager.

The default implementations of controllers are very simple. Only the persistence controller is able to perform some actions – starting and terminating the application. All other methods in this and the remaining controllers throw a *not implemented* exception, as they cannot be provided without specific knowledge about the application.

GCM-aware applications. These are very easy to connect to the framework. The only requirement towards the application developer is to deliver a GCM component (*user application component*, see Figure 5, right) with exported interfaces for each of the controllers (`dc`, `pc`, `cc`, and `sc`). Internally, they are expected to be bound to the user's implementation of the controllers.

5. Service and Resource Abstraction

The mediator component toolkit is using a *service and resource abstraction layer* for the boundary between system and application components on one side, and (remote) services and resources, on the other side. The Java Grid Application Toolkit (JavaGAT) [16]is an implementation of such a layer. It provides an object-oriented, high-level, and middleware-independent interface to the grid.

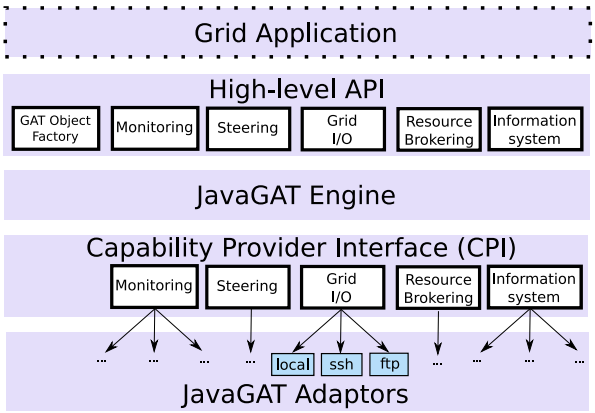

Figure 6. Structure of the JavaGAT implementation, from [16].

JavaGAT uses nested exceptions and intelligent dispatching of method invocations to automatically select the most suitable grid middleware that imple-

ments the requested operations [16]. For instance, file transfers are typically faster with GridFTP than with SSH, as GridFTP can use parallel data streams. Another reason for a particular preference could be security: the most secure transfer protocol could be tried first (much like SSH does). Figure 6 illustrates the structure of the JavaGAT implementation. The JavaGAT engine is using adaptor selection policies to express preferences for different adaptors and the middleware they interface to. We propose to modify the engine such that the configuration of these policies is more dynamic than its current implementation.

In JavaGAT, the selection process of the appropriate middleware (adaptors) is done at runtime using a default ordering policy that defines the order in which the adaptors are tried. This default ordering can be overridden by a user-defined policy allowing the user to define the order in which the adaptors are tried to service a particular request call. This can be done by defining an *AdaptorOrderingPolicy* class and specifying the name of the new ordering class using a command line option that sets a Java system property.

However, since the user-defined ordering policy is specified as a system property when the application is started, it is a one-time configuration that cannot be changed while the application is running. In order to make this configuration more flexible, we investigate how we can utilize GCMâŁ™s component architecture in order to make the policy more dynamic.

In order to make this possible we define a GCM component that exposes an interface that can allow the user of the application to provide a new adaptor ordering policy for a particular GATobject. Internally to the engine, each GATobject provides a method that can be invoked internally by the component to change the ordering of the adaptors in the adaptor list. Access to this list has to be synchronized since the list can be concurrently modified through the GCM component while it is being used by the adaptor selection process. This setup is illustrated in Figure 7.

6. Related Work

The work presented in this paper is presenting our experiences with integrating application and system components into a homogeneous system of components. It is directly based on work within CoreGRID, namely the Grid Component Model (GCM) [8], and the set of mediator components [9]. We are using the ProActive/GCM implementation from the GridCOMP project [10].

What distinguishes grids from other environments is their heterogeneity, dynamic variability of resource quality, and their non-negligible failure rates, in their totality requiring approaches to application development that take these non-functional properties into account [14]. In consequence, approaches to dynamically adapt applications to changing grid environments are legion. Here, we can only mention a few.

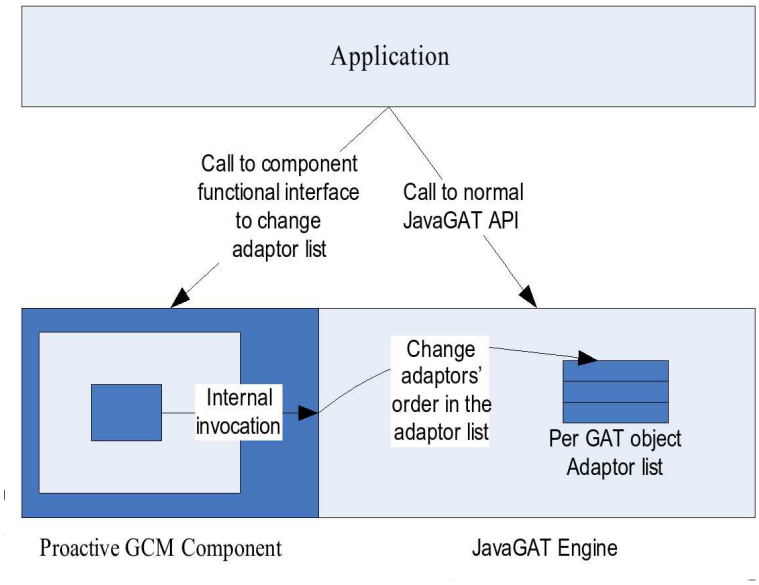

Figure 7. Dynamic adaptor ordering using GCM components.

First of all, the GCM model provides the core mechanisms for our work. Important, in this respect, is the work on skeleton-based, autonomic management of grid components within GCM [1]. Behavioural skeletons are closely related to higher-order components (HOC's) that are likewise proposed for performance adaptation of grid applications [3]. Both skeletons and HOC's are providing structural frameworks in which application components can be inserted and being leveraged from directly dealing with adaptation issues. In contrast, mediator components do not require applications to fit into certain structures but let them provide application-specific code to be interfaced with the provided controllers.

Of course, there also exist many approaches to adapting grid applications that are not based on components. Examples providing some form of application frameworks or infrastructures are [5, 11, 12, 18, 19]. The most puristic approach is to modify the application code itself, or to develop new, grid-aware applications [7, 17]. In contrast to these approaches, we propose to build both applications and their supportive environments from the same grid component model (GCM), and to tightly integrate them for flexible composition of efficient and adaptive grid applications.

Whereas application performance is the predominant goal of running applications in grids, it is not the only purpose for which dynamic adaptation is required. Via service and resource abstraction, applications become inde-

pendent of and portable across different grid middleware and infrastructure. This purpose is addressed by the Grid Application Toolkit (the JavaGAT) [16], or by the implementation of the recently standardized SAGA API [13]. Both SAGA and the JavaGAT need some form of configuration information from the user in order to identify and select proper middleware adaptors. Our mediator component-based framework provides an integrated mechanism to provide all necessary information to a resource and service abstraction layer, like SAGA or JavaGAT.

7. Conclusions

With the Grid Component Model (GCM), applications can be written in ways to cope with specific requirements of grid environments, namely resource heterogeneity, performance variability, and fluctuating availability. The integrated toolkit for grid applications is providing both a set of mediator components and a service and resource abstraction layer, allowing to integrate application components with grid middleware systems.

In this paper, we have presented our design for integrating the mediator components with the application itself. We have also shown, on the example of the JavaGAT, how the service and resource abstraction layer can be made adaptive, too. As of the time of writing, the mediator component toolkit has been implemented partially. With the advent of a complete GCM platform implementation, a fully integrated component platform will become available.

Acknowledgments

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

References

[1] 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 *Intl. Euromicro PDP 2008: Parallel Distributed and network-based Processing*, pp. 54–63, Toulouse, France, Feb. 2008. IEEE.

[2] G. Aloisio, Z. Balaton, P. Boon, M. Cafaro, I. Epicoco, G. Gombas, P. Kacsuk, T. Kielmann, and D. Lezzi. *Integrating Resource and Service Discovery in the CoreGrid Information Cache Mediator Component*. CoreGRID Integration Workshop 2005, Pisa, Italy, 2005.

[3] M. Alt, C. Dumitrescu, S. Gorlatch, A. Kertesz, G. Sipos, and D. Epema. Towards user-transparent performance prediction for workflows of higher-order components. In *Proceedings of the CoreGRID Integration Workshop*, pp. 345–356. CYFRONET Poland, 2006. ISBN 83-915141-6-1.

[4] R. Armstrong, G. Kumfert, L.C. McInnes, S. Parker, B. Allen, M. Sottile, T. Epperly, and T. Dahlgren. The CCA component model for high-performance scientific computing. *Concurrency and Computation: Practice and Experience*, 18(2):215–229, 2006.

[5] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov. Adaptive Computing on the Grid using AppLeS. *IEEE Trans. on Parallel and Distributed Systems*, 14(4):369–382, 2003.

[6] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The Fractal Component Model and Its Support in Java. *Software Practice and Experience*, special issue on Experiences with Auto-adaptive and Reconfigurable Systems, 36(11-12), 2006.

[7] W. Chrabakh and R. Wolski. GridSAT: A Chaff-based Distributed SAT Solver for the Grid. In *ACM/IEEE Conference on Supercomputing*, page 37, 2003.

[8] CoreGRID Institute on Programming Models. *Basic Features of the Grid Component Model (assessed)*, Deliverable D.PM.04, CoreGRID Network of Excellence, 2007. http://www.coregrid.net/mambo/images/stories/Deliverables/d.pm04.pdf.

[9] CoreGRID Institute on Grid Systems, Tools, and Environments. *Design of the Integrated Toolkit with Supporting Mediator Components*. Deliverable D.STE.05, CoreGRID Network of Excellence, 2007. http://www.coregrid.net/mambo/images/stories/Deliverables/d.ste05.pdf.

[10] The GridCOMP project, http://gridcomp.ercim.org/, 2008.

[11] E. Heymann, M.A. Senar, E. Luque, and M. Livny. Adaptive scheduling for master-worker applications on the computational grid. In *1st IEEE/ACM International Workshop on Grid Computing*, pp. 214–227, LNCS 1971, Springer Verlag, 2000.

[12] E. Huedo, R.S. Montero, and I.M. Llorente. A framework for adaptive execution in grids. *Software – Practice and Experience*, 34(7):631–650, May 2005.

[13] S. Jha, H. Kaiser, A. Merzky, and O. Weidner. Grid Interoperability at the Application Level Using SAGA. *International Grid Interoperabilty and Interoperation Workshop 2007(IGIIW 2007)*.

[14] T. Kielmann. Programming Models for Grid Applications and Systems: Requirements and Approaches. *IEEE John Vincent Atanasoff International Symposium on Modern Computing (JVA 2006)*, Sofia, Bulgaria, October 2006, pp. 27-32.

[15] E. Krepska, T. Kielmann, R. Sirvent, R.M. Badia. A Service for Reliable Execution of Grid Applications. In *Achievements in European Research on Grid Systems*, Springer Verlag, 2007.

[16] R.V. van Nieuwpoort, T. Kielmann, and H.E. Bal. User-friendly and reliable grid computing based on imperfect middleware. *ACM/IEEE Conference on Supercomputing (SC'07)*, 2007.

[17] A. Plaat, H.E. Bal, and R.F.H. Hofman. Sensitivity of parallel applications to large differences in bandwidth and latency in two-layer interconnects. In *5th International Symposium on High Performance Computer Architecture*, pp. 244–253, 1999.

[18] S.S. Vadhiyar and J.J. Dongarra. Self adaptivity in Grid computing. *Concurrency and Computation: Practice and Experience*, 17(2–4):235–257, 2005.

[19] G. Wrzesinska, J. Maassen, and H.E. Bal. Self-adaptive applications on the Grid. In *ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP'07)*, San Jose, CA, USA, March 2007.