

REDESIGNING THE SEGL PROBLEM SOLVING ENVIRONMENT: A CASE STUDY OF USING MEDIATOR COMPONENTS

Thilo Kielmann and Gosia Wrzesińska

*Dept. of Computer Science
Vrije Universiteit
Amsterdam, The Netherlands*

kielmann@cs.vu.nl

gosia@cs.vu.nl

Natalia Curre-Linde and Michael Resch

*High Performance Computing Center (HLRS)
University of Stuttgart
Germany*

linde@hls.de

resch@hls.de

Abstract The Science Experimental Grid Laboratory (SEGL) problem solving environment allows users to describe and execute complex parameter study workflows in Grid environments. Its current implementation provides much high-level functionality for executing complex parameter-study workflows. Alternatively, using a toolkit of mediator components that integrate system-component capabilities into application code would allow to build a system like SEGL from existing, more generally applicable components, simplifying its implementation and maintenance. In this paper, we present the given design of the SEGL PSE, analyze the provided functionality, and identify a set of mediator components that can generalize the functionality required by this challenging application category.

Keywords: Grid component model, mediator components, SEGL

1. Introduction

The SEGL problem solving environment [9] allows end-user programming of complex, computation-intensive simulation and modeling *experiments* for science and engineering. Experiments are complex workflows, consisting of domain-specific or general purpose simulation codes, referred to as *tasks*. For each experiment, the tasks are invoked with input parameters, that are varied over given parameter spaces, together describing individual parameter studies.

SEGL allows users to program so-called *applications* using a graphical user interface. An application consists of several tasks, the *control flow* of their invocation, and the *data flow* of input parameters and results. For the parameters, the user can describe iterations for parameter sweeps; also, conditional dependencies on result values can be part of the control flow. Using such a user application program, SEGL can execute the tasks, provide them with their respective input parameters, and collect the individual results in an experiment-specific database.

SEGL's current implementation allows executing complex parameter study workflows, involving a GUI-based frontend, an execution engine that schedules and monitors the progress of the experiment, as well as a data base server using an experiment-specific schema. By following this design, much high-level functionality has been implemented on top of existing Grid middleware, however in a way that is specific to SEGL.

Alternatively, using a toolkit of mediator components that integrate system-component capabilities into application code would allow to build a system like SEGL from existing, more generally applicable components, simplifying its implementation and maintenance. In this paper, we propose a redesign of SEGL based on such mediator components. Important insights are (a) the necessity to integrate components with (legacy) Web-service based middleware, and (b) the requirement of a persistent application-execution service.

In the following, we revisit our view of component-based Grid application environments (Section 2), present SEGL's current architecture and functionality (Section 3), and identify a set of mediator components that can generalize the functionality required by this challenging application category (Section 4). Ongoing work related to the development of such mediator components is presented in Section 5.

2. Component-based Grid application environments

A technological vision is to build Grid software such that applications and middleware will be united to a single system of components [7]. This can be accomplished by designing a toolkit of components that mediate between both applications and system components. The goal 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. Both the strength and the challenge 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 middleware and system software.

As outlined in [16], both components and Web services parallel traditional objects by encapsulating state from their clients behind well-defined interfaces. They differ, however, in their applicability within given environments. Objects allow client/server communication within a single application process. With components, client and server can be distributed across different processes, however, they have to share the same execution environment which is the component model and one or more *interoperable* implementations of this model. Web services, finally, allow the distribution of client and server across different processes and execution environments, allowing the loosely-coupled integration of heterogeneous clients, resources, and services.

Components are to be preferred over Web services as they provide higher execution performance, however, at the price of reduced interoperability. Besides better performance, components also allow reflective behavior and re-composition of application software at run time, opening the path to fault-tolerant and behavior-adaptive Grid applications [8]. The limitation to a single execution environment, however, contradicts the idea of Grid computing where interoperability plays a central role for the integration of independently created and maintained resources and services. In consequence, we have to treat existing, Web-service based middleware as legacy systems that have to be integrated into a component-based Grid software platform.

A possible rendering of the envisioned mediator components along with their embedding into a generic component platform is shown in Figure 1. This diagram is based on our previous work described in [6]. Boxes in grey are examples of external services that are integrated into the overall platform.

The upper part of Figure 1 outlines a component-based Grid application, where we distinguish between three layers. The lowest layer, the runtime environment, provides the interface of the application with external (Web-service based) resources and services. The middle layer in the application stack consists of an extensible set of mediator components that provide higher-level functionality to the application. The topmost layer consists of the application components themselves, possibly enriched by a so-called *Integrated Toolkit*

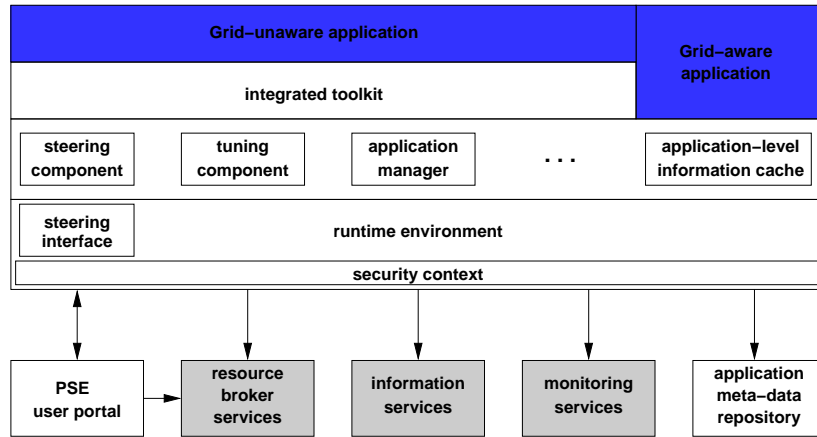


Figure 1. Envisioned generic component platform

that provides Grid-unaware programming abstractions to the application. In the following, we present the envisioned components individually.

Runtime Environment The runtime environment implements a set of component interfaces to various kinds of Grid services and resources, like job schedulers, file systems, etc. It implements a delegation mechanism that forwards invocations to service providers. Doing so, the runtime environment provides an interface layer between application components and both system components and middleware services. Examples of such runtime environments are the GAT [2], or GGF's SAGA [12]. By providing dynamic bindings to the various service providers, the runtime environment bridges the gap between components and services, and allows to use system services with either type of interface, next to each other at the same time.

Security Context As the runtime environment implements the application's interface to services and resources outside its own scope, care has to be taken of authentication and authorization mechanisms each time external entities are getting involved. For this purpose, the security context forms an integral part of the runtime environment.

Steering Interface A dedicated part of the runtime environment is the steering interface. It is supposed to make applications accessible by system entities and user-interfaces (like portals or PSE's) like any other component in the system. This interface at the border of component-based applications and external services and components is supposed to relay to (and

protect) internal component interfaces. Access control to the steering interface is subject to the security context.

Application-level meta-data repository This repository is supposed to store meta data about a specific application, storing, e.g., timing or resource requirements from previous, related runs. The collected information will be used by other components to support resource management (location and selection) and to optimize further runs of the applications automatically.

Application-level information cache

This component is supposed to provide a unified interface to deliver all kinds of meta-data (e.g., from a Grid information service (GIS), a monitoring system, or 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 access times of current implementations like Globus GIS (up to multiple seconds) down to the order of a single method invocation.

Steering Components Controlling and steering of applications by the user, e.g., via application managers, user portals, and PSE's, requires a component level interface to give external entities access to the application. From outside the application, the steering components will be accessible via the steering interface. For example, we envision steering components with the following kinds of interfaces:

steering controller – for modifying application parameters

persistence controller – for externally triggering checkpoints

distribution strategy controller – for changing the data distribution

component explorer – for exploring (and modifying) the current component composition

Tuning Components Tuning components can be used to optimize the application's runtime behavior, based on observed behavior of the application itself and on external status information, as provided by the application-level information cache component. Tuning components can be either passive, or active, in the latter case carrying their own threads of activity.

Application Manager An application manager establishes a pro-active user interface, in charge of tracking an application from submission to successful completion. It will be in charge of guaranteeing such successful

completion in spite of temporary error conditions or performance limitations. A persistent service will become an integral part of this functionality.

3. The SEGL system architecture

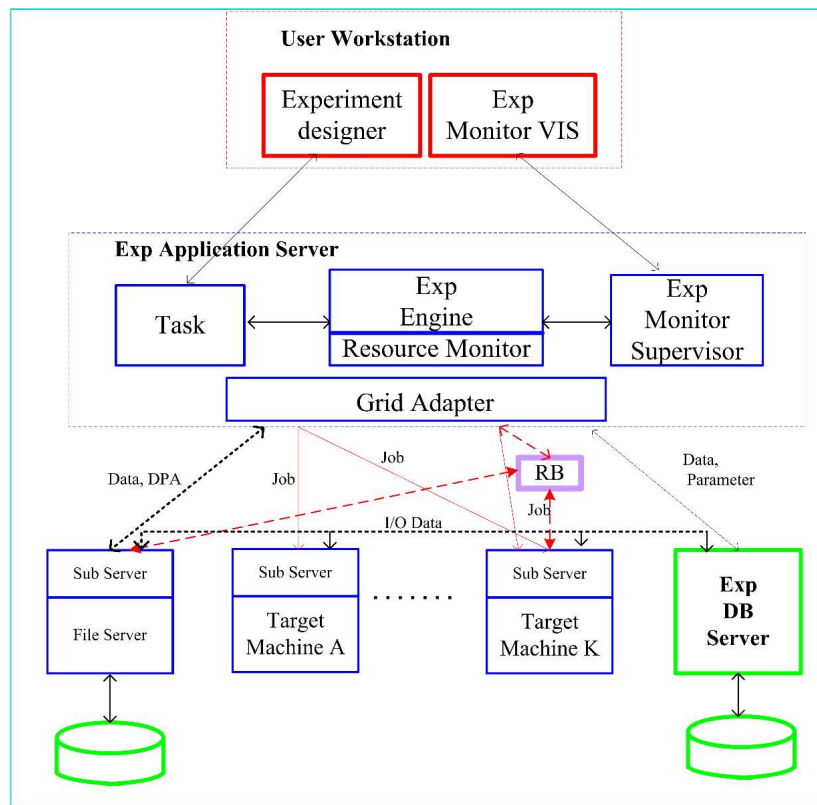


Figure 2. Current SEGL architecture

Figure 2 shows the current system architecture of SEGL. It consists of three main components: the User Workstation (Client), the Experiment Application Server (ExpApplicationServer), and the Experiment database server (ExpDB-Server). Client and ExpApplicationServer communicate with each other using a traditional client/server architecture, based on J2EE middleware. The interaction between ExpApplicationServer and the Grid resources is done through a Grid Adaptor, interfacing to Globus [11] and UNICORE [15] middleware.

The client on the user's workstation is composed of the graphical experiment designer tool (ExpDesigner) and the experiment process monitoring and visu-

alization tool (ExpMonitorVIS). The ExpDesigner is used to design, verify and generate the experiment's program, organize the data repository and prepare the initial data, using a simple graphical language.

Each experiment is described at three levels: control flow, data flow and the data repository. The control flow level describes which code blocks will be executed in which order, possibly augmented by parameter iterations and conditional branches. Each block can be represented as a simple parameter study. An example is shown in Fig. 3. The data flow level describes the flow of parameter data between the individual code blocks. On the data repository level, a common description of the metadata repository is created for the given experiment. The repository is an aggregation of data from the blocks at the data flow level.

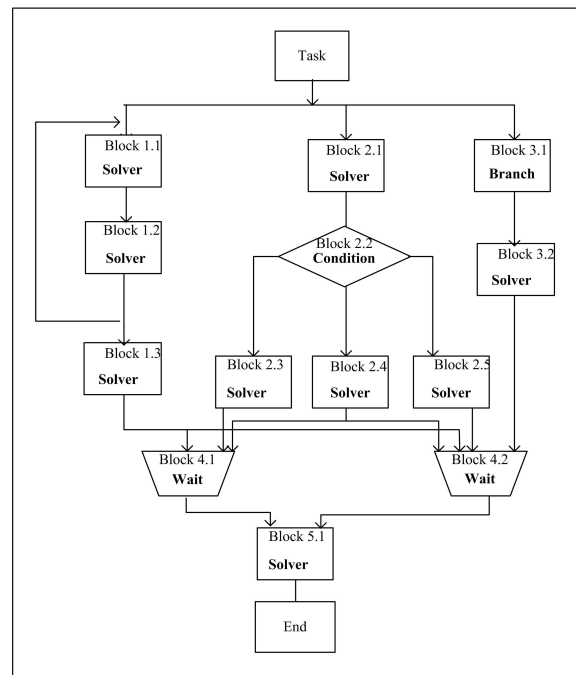


Figure 3. Example experiment control flow

After completing the graphical design of the experiment program, it is “compiled” to the *container application*. This creates the experiment-specific parts for the ExpApplicationServer as well as the experiment's data base schema. The container application of the experiment is transferred to the ExpApplicationServer and the schema descriptions are transferred to the server data base. Here, the meta data repository is created.

The ExpApplicationServer consists of the experiment engine (ExpEngine), the container application (Task), the controller component (ExpMonitorSupervisor) and the ResourceMonitor. The ResourceMonitor holds information about the available resources in the Grid environment. The MonitorSupervisor controls the work of the runtime system and informs the Client about the current status of the jobs and the individual processes. The ExpEngine is executing the application Task, so it is responsible for actual data transfers and program executions on and between server machine in the Grid.

The final component of SEGL is the data base server (ExpDBServer). The automatic creation of the experiment is done according to the structure designed by the user. All data produced during the experiment such as input data for the parameter study, parameterization rules etc are kept in the ExpDBServer.

As SEGL parameter studies may run for significant amounts of time, application progress monitoring becomes necessary. The MonitorSupervisor, being part of the experiment application server, monitors the work of the runtime system and notifies the client about the current status of the jobs and the individual processes. The ExpEngine is the actual controller of the SEGL runtime system. It consists of three sub systems: the TaskManager, the JobManager and the DataManager. The TaskManager is the central dispatcher of the ExpEngine. It coordinates the work of the DataManager and the JobManager as follows:

- 1 It organizes and controls the execution sequence of the program blocks. It starts the execution of the program blocks according to the task flow and the conditions within the experiment program.
- 2 It activates a particular block according to the task flow, selects the necessary computer resources for the execution of the program and deactivates the block when this section of the program has been executed.
- 3 It informs the MonitorSupervisor about the current status of the program.

The DataManager organizes data exchange between the ApplicationServer and the FileServer and between the FileServer and the ExpDBServer. Furthermore, it provides the tasks processes with their the input parameter data. For progress monitoring, the MonitorSupervisor is tracking the status of the ExpEngine and its sub components. It forwards status update events to the ExpMonitorVIS, closing the loop to the user. SEGL's progress monitoring is currently split in to parts:

- 1 The experiment monitoring and visualization on the client side (ExpMonitor VIS). It is designed for visualizing the execution of the experiment and its computation processes. The ExpMontitorVis allows the user to start, stop, the experiment, and to change the input data and to subsequently re-start the experiment or some part of it.

- 2 The MonitorSupervisor within the application server controls and observes the work of the runtime system (Exp Engine). It sends continuous messages to the ExpMonitorVis on the client workstation.

This subdivision allows the user to disconnect from its running experiment. In this case, all status update messages will be stored with the application server for delivery to the client as soon as it will become reconnected.

4. Extracting mediator components from the SEGL functionality

The SEGL system constitutes an interesting use case for component-based Grid systems as it comprises all functionality required for complex task-flow applications. In this section, we try to identify, within the existing SEGL implementation, generic functionality that could be implemented in individual, re-usable or exchangeable components.

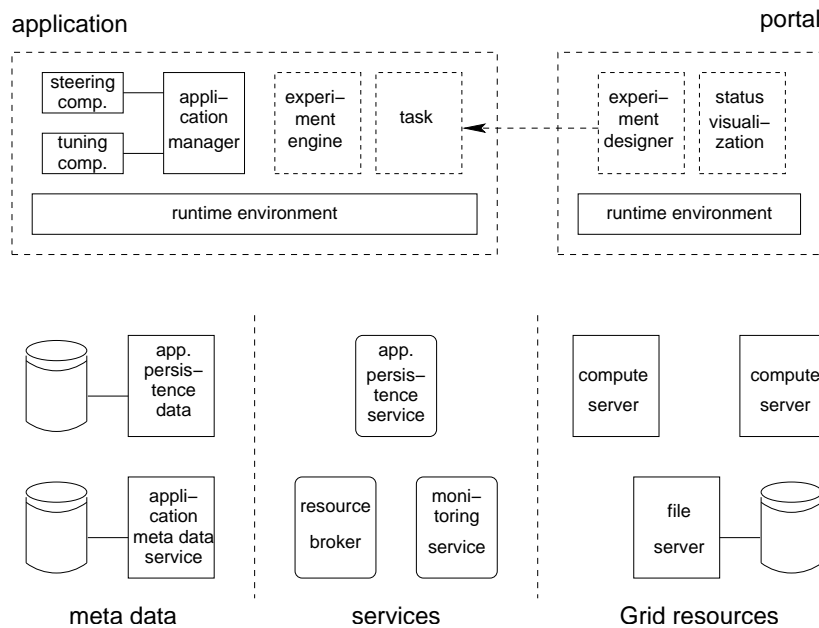


Figure 4. SEGL redesigned using mediator components

In the current SEGL architecture, as shown in Fig. 2, there is a subdivision to three major areas: the user interface, the experiment application server, and the Grid resources and services. the latter consist of file servers for the experiment’s data, compute servers for experiment tasks, and additionally the experiment database, storing all experiment-specific status information. The user interface

consists of the experiment programming environment (the “designer”) and the application execution visualization component.

The most interesting element of SEGL is the experiment application server. It concentrates the application logic (implemented via the *experiment engine* and the experiment-specific *task*), a Grid middleware interface layer (called *adaptor*), as well as progress monitoring functionality. Less visible in Fig. 2 is the fact that the experiment application server is a persistently running service. It has been designed as such to decouple the user interface from possibly long-running experiment codes.

Having such a persistently running service is certainly necessary to guarantee application completion in spite of transient error conditions, without user involvement. However, adding such a domain-specific, permanent service to the pre-installed middleware may be causing administrative and security-related concerns.

Based on this analysis, we propose the following re-design based on mediator components, trying to refactor SEGL’s functionality into domain-specific components, complemented by general-purpose, reusable components. This redesign is shown in Fig. 4.

In this design, the software Grid infrastructure is organized in three tiers: resources, services, and meta data. For SEGL, relevant Grid resources are both compute and file servers, the machines that are able to execute experimentation tasks and providing the application data. These servers are accessible via Grid middleware, whichever happens to be installed on each resource.

Relevant Grid services are a resource monitoring service, like e.g. Delphoi [14] and a resource broker that matches tasks to compute servers. For the Grid services, we also propose an *application persistence service*. This is a persistent service that keeps track of a given application and ensures it runs until successful completion, possibly restarting it in case of failures. Being a general-purpose, domain-independent service, it can be deployed in a virtual organization without overly administrative efforts, relying on a security concept that needs to be deployed only once for all kinds of applications. In a component-based architecture, we assume these services to have interfaces that fit into the component model.

The final infrastructure category is meta data. For persistent storage of such meta data, one or more servers can be deployed. One such component is the application meta data repository, equivalent to SEGL’s current experiment data base. In addition, a meta data storage component is needed for the status information of the application persistence service.

The Grid infrastructure is used by two programs, the SEGL *application* and a user *portal*. Within these programs, Fig. 4 shows general-purpose components as solid boxes and domain-specific components as dashed boxes. Both programs are using the runtime environment for communication with the Grid

infrastructure. The portal is implementing both the experiment designer as well as the experiment status visualization.

SEGL's monitoring and steering facilities are divided across *application* and *portal*. Within the portal, the status visualization provides the user interface. Within the application, the *steering component* handles change requests for the parameter data. To allow the user to disconnect and later re-connect to his or her application, also the progress monitoring needs storage for its events that is persistent, at least until completion of the overall experiment. For this purpose, the *application meta data service* provides the appropriate storage facilities. The actual progress monitoring then takes place within the *application manager* component, but possibly a dedicated application monitoring and event handling component could be added.

The SEGL application is composed of components only. The experiment engine implements the SEGL-specific application logic, while the task component is created by the experiment designer within the SEGL portal. The experiment engine is accompanied by the generic application manager component which is responsible for both runtime optimization, using dedicated tuning and steering components, and for registering the SEGL application with the application persistence service. In the proposed combination, the experiment engine is responsible for the SEGL-specific control flow, while the application manager is in charge of all Grid-related control aspects, leading to a clear separation of concerns.

5. Related Work and Ongoing Developments

The work presented here is embedded in a larger scope of developments, both in a wider context and directly regarding the development of mediator components.

5.1 Related Work

Whereas notions and models for *components* are still diverse [1, 5, 8, 17, 18], there is a trend towards building Grid application environments from entities that can be selected and dynamically loaded at runtime [13].

Ibis [20] is a runtime environment for executing parallel Java applications in Grid environments. It uses Java's dynamic byte code loading for matching application needs to the given network environments and protocols, such as TCP/IP or local Myrinet clusters. The work in [10] extends this concept to configuring whole protocol stacks from runtime components.

The Grid Application Toolkit (GAT) [2] provides a simple and uniform API to various Grid middleware, like Globus [11] or Unicore [15]. The GAT API is implemented via a so-called engine that uses dynamically linked *adaptors* to bind Grid applications to the actual Grid environment. The Commodity Grid

Kits (CoG kits) [21] similarly provide simplified API's to Globus, and more recently also to ssh-based environments.

ProActive [4] is another Java-based execution environment for parallel Grid applications. Unlike Ibis, it uses the Fractal [5] component model for providing the units of dynamic composition. Assist [1] is another component-based execution environment for parallel Grid applications. Both ProActive and Assist are using components for deployment and runtime adaptation. Neither of them, however, is proposing a comprehensive component toolkit for mediating between application needs and middleware services. The proposed lightweight, generic grid platform [19] aims in this direction by building a component-based Grid middleware infrastructure, on top of which mediator components could be implemented with ease.

5.2 Ongoing Developments of Mediator Components

Several efforts are currently undertaken to investigate the feasibility of building the envisioned set of mediator components. These efforts are explorative, aiming at gaining early experiences. Completeness and production quality code, however, are beyond our current scope.

Grid Component Model A suitable model for Grid components (GCM) [8] is vital for developing mediator components, too. Currently, our group at Vrije Universiteit is experimenting with the Fractal component model [5] which is considered a starting point for developing the GCM. First results of this work have led to the refinements of the generic Grid component platform, as shown in Fig. 1.

Runtime Environment We are currently using the Grid Application Toolkit (GAT) [2] as runtime environment. We are designing component-level (wrapper) interfaces to its provided functionality. The design alternative of redesigning the whole runtime environment based on components has been ruled out, due to the requirement of integrating legacy services and resources, as outlined in Section 2.

Application-level Information Cache The functionality of such a cache component is currently being developed that can integrate various kinds information providers [3]. The design of a proper (Fractal) component interface is subject to ongoing work.

Application Manager The design of an application manager, in combination with an application persistence service and data repository, as shown in Fig. 4, is also currently being investigated within our group at Vrije Universiteit.

6. Conclusions

The SEGL problem solving environment allows end-user programming of complex, computation-intensive simulation and modeling *experiments* for science and engineering. As such, it constitutes an interesting use case for component based Grid systems as it comprises all functionality required for complex task-flow applications.

In this paper, we have identified, within the existing SEGL implementation, generic functionality that can be implemented in individual, reusable components. We have proposed a three-tier Grid middleware architecture, consisting of the resources themselves, persistent services, and meta data. Important insights are (a) the necessity to integrate components with (legacy) Web-service based middleware, and (b) the requirement of a persistent application-execution service.

Based on this architecture, we were able to compose a SEGL experiment execution application from mostly general-purpose components, augmented only by a SEGL-specific experiment engine and the dynamically created experiment task description. With this architecture we tried to refactor a system like SEGL such that general-purpose functionality is implemented in reusable components while a minimal set of domain-specific components can be added to compose the overall application.

With currently available technology, such components do not exist yet, as suitable component models, and especially generally accepted and standardized interfaces, are subject to ongoing work, as outlined in Section 5. Once such components become available and mature [6], refactoring SEGL's implementation will be an interesting exercise.

Acknowledgements

This research work is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265). We would like to thank Urszula Herman-Izycka and Michal Ejdy for their valuable contributions to refining the generic component platform.

References

- [1] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, and C. Z. occolo. Assist as a research framework for high-performance grid programming environments. In J. C. Cunha and O. F. Rana, editors, *Grid Computing: Software environments and Tools*. Springer-Verlag, 2004.
- [2] G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. van Nieuwpoort, A. Reinefeld, F. Schintke, T. Schütt, E. Seidel, and B. Ullmer. The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid. *Proceedings of the IEEE*, 93(3):534–550, 2005.

- [3] 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. In *CoreGRID Integration Workshop*, Pisa, Italy, 2005.
- [4] F. Baude, D. Caromel, and M. Morel. From distributed objects to hierarchical grid components. In *International Symposium on Distributed Objects and Applications (DOA)*, Catania, Sicily, Italy, 3-7 November, Springer Verlag, 2003. Lecture Notes in Computer Science, LNCS.
- [5] E. Bruneton, T. Coupaye, and J. B. Stefani. Recursive and Dynamic Software Composition with Sharing. In *Seventh International Workshop on Component-Oriented Programming (WCOOP02)*, Malaga, Spain, 2002. Held at ECOOP 2002.
- [6] CoreGRID Institute on Problem Solving Environments, Tools, and GRID Systems. Proposal for mediator component toolkit. CoreGRID deliverable D.ETS.02, 2005.
- [7] CoreGRID Institute on Problem Solving Environments, Tools, and GRID Systems. Roadmap version 1 on Problem Solving Environments, Tools, and GRID Systems. CoreGRID deliverable D.ETS.01, 2005.
- [8] CoreGRID Institute on Programming Models. Proposal for a Common Component Model for GRID. CoreGRID deliverable D.PM.02, 2005.
- [9] N. Curre-Linde, U. Küster, M. Resch, and B. Risio. Science Experimental Grid Laboratory (SEGL) Dynamical Parameter Study in Distributed Systems. In *ParCo 2005*, Malaga, Spain, 2005.
- [10] A. Denis. Meta-communications in Component-based Communication Frameworks for Grids. In *HPC-GECO Workshop, held in conjunction with HPDC-15*, Paris, France, 2006.
- [11] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Int. Journal of Supercomputer Applications*, 11(2):115–128, 1997.
- [12] Global Grid Forum (GGF). Simple API for Grid Applications (SAGA). <https://forge.gridforum.org/projects/saga-rg/>, 2005.
- [13] T. Kielmann, A. Merzky, H. Bal, F. Baude, D. Caromel, and F. Huet. Grid Application Programming Environments. In *Future Generation Grids*, pages 283–306. Springer Verlag, 2006.
- [14] J. Maassen, R. V. van Nieuwpoort, T. Kielmann, K. Verstoep, and M. den Burger. Middleware Adaptation with the Delphoi Service. *Concurrency and Computation: Practice and Experience*, 2006. Special issue on Adaptive Grid Middleware.
- [15] D. Erwin (Ed.). *Joint Project Report for the BMBF Project UNICORE Plus*. UNICORE Forum e.V., 2003.
- [16] R. Sessions. Fuzzy Boundaries: Objects, Components, and Web Services. *ACM Queue*, 2(9):40–47, 2005.
- [17] The CCA Forum. The Common Component Architecture (CCA) Forum home page, 2005. <http://www.cca-forum.org/>.
- [18] The Object Management Group (OMG). CORBA Component Model, V3.0. <http://www.omg.org/technology/documents/formal/components.htm>, 2005.
- [19] J. Thiyagalingam, N. Parlavantzas, S. Isaiadis, L. Henrio, D. Caromel, and V. Getov. Proposal for a Lightweight, Generic Grid Platform Architecture. In *HPC-GECO Workshop, held in conjunction with HPDC-15*, Paris, France, 2006.
- [20] R. V. van Nieuwpoort, J. Maassen, R. Hofman, T. Kielmann, and H. E. Bal. Ibis: an Efficient Java-based Grid Programming Environment. In *Joint ACM Java Grande - ISCOPE 2002 Conference*, pages 18–27, Seattle, Washington, USA, November 2002.

- [21] G. von Laszewski, I. Foster, J. Gawor, and P. Lane. A Java Commodity Grid Kit. *Concurrency and Computation: Practice and Experience*, 13(8–9):643–662, 2001.