Componentising a scientific application for the grid.

Nikos Parlavantzas[1,2]
Matthieu Morel[2]
Françoise Baude[2]
Fabrice Huet[2]
Denis Caromel[2]
Vladimir Getov[1]

[1] Harrow School of Computer Sciences, University of Westminster
[2] INRIA Sophia Antipolis, 2004, route des Lucioles, BP 93, F-06902 Sophia Antipolis Cedex, France

This is a reproduction of CoreGRID Technical Report Number TR-0031, April 25, 2006 and is reprinted here with permission.

The report is available on the CoreGRID website, at:

http://www.coregrid.net/mambo/images/stories/TechnicalReports/tr-0031.pdf

---

Componentising a Scientific Application for the Grid

Nikos Parlavantzas[1,2], Matthieu Morel[2], Françoise Baude[2], Fabrice Huet[2], Denis Caromel[2], Vladimir Getov[1]

[1]Harrow School of Computer Science, University of Westminster, HA1 3TP, U.K.
{N.Parlavantzas, V.S.Getov}
@westminster.ac.uk

[2]INRIA Sophia Antipolis, 2004, route des Lucioles, BP 93, F-06902 Sophia Antipolis Cedex, France
FirstName.LastName@inria.fr

CoreGRID Technical Report
Number TR-0031
April 25, 2006

Institute on Grid Systems, Tools, and Environments

CoreGRID - Network of Excellence
URL: http://www.coregrid.net

CoreGRID is a Network of Excellence funded by the European Commission under the Sixth Framework Programme

Project no. FP6-00426

Componentising a Scientific Application for the Grid

Nikos Parlavantzas[1,2], Matthieu Morel[2], Françoise Baude[2], Fabrice Huet[2], Denis Caromel[2], Vladimir Getov[1]

[1]Harrow School of Computer Science, University of Westminster, HA1 3TP, U.K.
{N.Parlavantzas, V.S.Getov}
@westminster.ac.uk

[2]INRIA Sophia Antipolis, 2004, route des Lucioles, BP 93, F-06902 Sophia Antipolis Cedex, France
FirstName.LastName@inria.fr

CoreGRID TR-0031
April 25, 2006

Abstract

Building and evolving Grid applications is complex. Component-based development has emerged as an effective approach to building flexible systems, but there is little experience in applying this approach to Grid programming. This paper presents our experience with reengineering a high performance numerical solver to become a component-based Grid application. The adopted component model is an extension of the generic Fractal model that specifically targets grid environments. The paper provides qualitative and quantitative evidence that componentisation has improved the modifiability and reusability of the application while not significantly affecting performance.

1. Introduction

As Grid technologies are becoming widely available, managing the complexity of building and evolving Grid applications is becoming increasingly important. Component-based development has emerged as an effective approach to building complex software systems; its benefits include reduced development costs through reusing off-the-self components and increased adaptability through adding, removing, or replacing components. Naturally, applying component-based development to Grid programming is currently attracting much interest. Examples of component models applicable to this field include CCA (Common Component Architecture) [11], CCM

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

(Corba Component model) [20], and the emerging GCM (Grid Gomponent Model) [12], currently under development within the CoreGRID European project. Despite this growing interest, there is still little experience in applying components to Grid computing, and developers are not provided with adequate guidance and support.

The main aim of this work is to present our experience with applying component-based development to the domain of high performance scientific applications running on the Grid. Specifically, the work describes how a numerical solver, originally implemented as distributed object application, was reengineered into a component-based application. The adopted component model extends the generic Fractal model [9], similarly to the GCM. The model is implemented on top of the ProActive middleware [21]. We show that componentisation has increased the modifiability of the application without any significant negative effects on performance. Another aim of this work is to present the process that underpinned the reengineering effort. This is a general architecture-based process that can be applied for transforming any object-based system to a component-based system.

The rest of this paper is structured as follows. Section 2 provides background on the numerical application, called Jem3D, and the distributed object platform on which it is built. Section 3 presents our approach to reengineering this application, which comprises a general componentisation process and a Grid-enabled component model. Section 4 then describes our componentisation experience and the resulting system. Section 5 provides some performance results, and section 6 discusses related work. Finally, section 7 concludes the paper.

2. Background on Jem3D

This section provides background on Jem3D, the application at the focus of this paper, and the ProActive library, the distributed object platform used by Jem3D.

2.1. Jem3D overview

Jem3D is a numerical solver for the 3D Maxwell's equations modelling the time domain propagation of electromagnetic waves [6]. It relies on a finite volume approximation method operating on unstructured tetrahedral meshes. At each time step, the method evaluates flux balances as the combination of elementary fluxes computed through the four facets of a tetrahedron. The complexity of the calculation can be changed by modifying the number of tetrahedra in the domain. This is done through setting the *mesh size*; i.e., the triplet (m1×m2×m3) that specifies the number of points on the x, y, and z axes used for building the mesh. Parallelisation relies on dividing the computational domain into a number of subdomains; the domain division is controlled by another triplet (d1×d2×d3) that determines the number of subdomains on each axis. Since some facets are located on the boundary between subdomains, neighbouring subdomains must communicate to compute the values of those border facets. The original Jem3D builds on the ProActive library, outlined next.

2.2. The ProActive library

The ProActive library is a Java middleware for parallel, distributed, and concurrent programming [21]. The ProActive core supports a uniform programming model based on remotely accessible

active objects. Each active object has its own thread of control and decides in which order to serve incoming method calls, which are stored in a queue of pending requests. Remote method calls on active objects are asynchronous with automatic synchronization. This is achieved via automatic creation of *future objects* combined with a synchronization mechanism known as *wait-by-necessity*. The library includes a set of high-level services, such as weak migration, security, and fault tolerance.

Two key features of ProActive are its support for typed group communication and descriptor-based deployment. *Group communication* enables triggering method calls on a group of active objects with compatible type, dynamically generating a group of results. Invoking a group of active objects takes exactly the same form as invoking one active object, which simplifies the programming of applications with similar activities running in parallel. Moreover, group invocations incorporate optimisations that make them more efficient than sequentially invoking a set of objects. *Descriptor-based deployment* enables deploying distributed applications anywhere without having to modify the source code. References to hosts, protocols and other infrastructure details are removed from the application code, and specified in XML descriptor files.

2.3. Jem3D architecture

Figure 1 shows the runtime structure of the original Jem3D (a 2×2×1 domain division is assumed); the main elements of the architecture are outlined next.

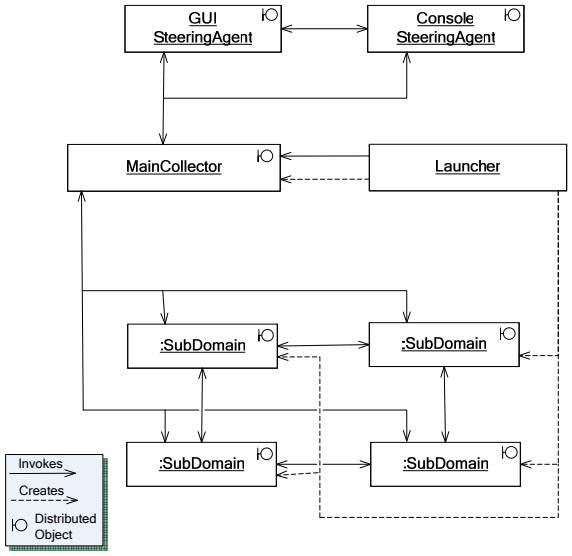

Figure 1. Jem3D Architecture

Subdomains correspond to partitions of the 3D computational domain; they perform electromagnetic computations and communicate with their closest neighbours in the 3D grid. Moreover, they send partial solutions with a predefined frequency to the main collector. The *main collector* is responsible for monitoring and steering the computation by interacting with the subdomains. The monitoring and steering functionality is used by one or more *steering agents*, which are dynamically registered with the main collector. The application includes a command-

line agent and a graphical agent with visualisation capabilities. Steering agents communicate with each other to ensure that only a single agent at a time has the right to control the computation. Finally, the *launcher* is responsible for obtaining the input data, creating the main collector and the subdomains, setting up the necessary connections between them, initialising them with the necessary information, and starting the computation. Communication between the entities relies on the asynchronous remote invocation and group communication mechanisms provided by ProActive.

The original Jem3D application suffers from limited modifiability and limited reusability of its parts. This can be largely attributed to two factors. First, the application lacks reliable architectural documentation, which is essential for understanding and evolving complex software systems. Jem3D has been subjected to successive changes by multiple people without corresponding updates to the architectural information. Second, the application parts are tightly coupled together. Indeed, as in most object-oriented applications, the code includes hard-wired dependencies to classes, which limits the reusability of classes, increases the impact of changes, and inhibits run-time variability. For example, changing the subdomain implementation requires updating the source code of both the main collector and the launcher and rebuilding the whole application. As another example, although the Jem3D parallelisation follows a typical *geometric decomposition pattern* [18], no part of the application can be reused in other contexts where this pattern is applicable. To address such modifiability and reusability limitations, Jem3D was re-engineered into a component-based system.

3. Approach

This section presents our approach for addressing the modifiability and reusability limitations of Jem3D. The approach consists of a general componentisation process and the use of the Fractal/ProActive component technology, discussed in the following two sections.

3.1. Componentisation process

The purpose of the componentisation process is to transform an object-based system to a component-based system. The process assumes that the target component platform allows connecting components via provided and required interfaces, and that it minimally supports the same communication styles as the object platform (e.g., remote method invocation, streams, events). Figure 2 shows the main activities and artefacts defined by the componentisation process. Note that the activities do not necessarily proceed sequentially. For example, the activity “Restructure Original System” may start when an initial component architecture is designed, and it may be revisited when an updated architecture is available. The activities are explained next.

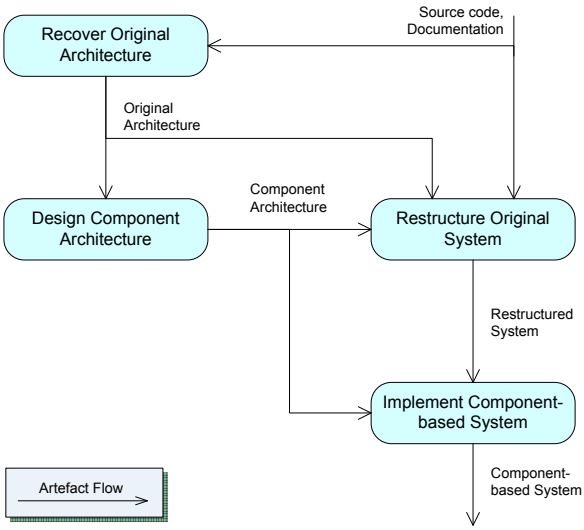

Figure 2. Componentisation process

Recover Original Architecture

The goal of this activity is to produce an architectural description of the original system, serving as the basis for understanding and transforming the system. The activity uses as input the source code, documentation, build files, and any other software artefacts. It consists in analysing the source system, extracting architecturally significant information, and documenting different views of the architecture. At a minimum, the documentation must include a run-time view describing executing entities (e.g., distributed objects, objects, processes), communication paths, and interactions over those paths (e.g., sequences of remote method invocations).

Design Component Architecture

The goal of this activity is to design the target component architecture using as input the original architecture. The component architecture specifies a set of components, their relationships, and the interactions among them and builds on the target component model. The activity can be divided into four steps:

- *Define initial architecture*. The executing entities of the original architecture are used as candidate components to form an initial component architecture.
- *Refine component selection*. Candidate components are decomposed into smaller components or integrated into larger components, and their relationships and interactions are updated accordingly. These changes are driven by modifiability and performance concerns. Decomposition is typically used to increase the reusability of components and the flexibility of the architecture, whereas integration is used to reduce performance overheads.
- *Specify component interfaces*. By analysing and organising the interactions between each component and its environment, this step identifies provided and required interfaces. Multiple interfaces for each component are defined in order to reduce dependencies.

- *Refine architecture using available component model features*. The component architecture is adapted to exploit all the available features provided by the target component model, such as hierarchical composition in Fractal, or implicitly-accessed, container services in CCM.

Restructure Original System

The goal of this activity is to restructure the original code to make it match closely the target component architecture, while avoiding any dependencies on the target component platform. Specifically, the activity involves implementing and testing an interface-based version of the system in which entities communicate as much as possible via explicitly identified provided/required interfaces. The motivation for the activity is to validate a large part of the target architecture at an earlier time. Moreover, the activity makes the migration to the component platform easier than it would otherwise be. The activity can be divided into the following steps:

- *Align code with component architecture*. This step ensures that the code includes classes which correspond to all intended components, and that these classes implement all interfaces provided by their corresponding components.
- *Add dependency injection mechanism*. Supporting configurable connections requires a uniform mechanism for injecting references to required interfaces into objects. Such mechanisms are provided by most component models, and are manifested as standard methods for accepting and managing interface references. This step ensures that all classes corresponding to intended components support an injection mechanism, thus making their dependencies explicit and externally modifiable.
- *Use injection mechanism*. This step modifies the classes so that they invoke collaborating classes only through injected references. Moreover, the step modifies any "injector" code that supplies a class with references to required objects to use the uniform injection mechanism.

Implement Component-based system

The goal of this activity is to implement and test the new component-based system. It uses as inputs the component architecture and the restructured, interface-based version. It typically involves minor changes for repackaging classes as component implementations. It may also involve changes for exploiting features of the component model that were unavailable in the original object platform.

3.2. Fractal/ProActive

Fractal/ProActive is a parallel and distributed component model that specifically targets Grid applications [7]. Fractal/ProActive conforms to the generic Fractal model [9] and extends it with a number of features that support Grid programming. Fractal/ProActive is implemented on top of the ProActive library [21]. Fractal and the Fractal/ProActive-defined extensions are examined in turn next.

Fractal components are runtime entities that communicate exclusively through interfaces of two types: *client interfaces* that emit operation invocations and *server interfaces* that accept them. Interfaces are connected through communication paths, called *bindings*. Fractal distinguishes *primitive* components from *composite* components formed by hierarchically assembling other components (called sub-components). This hierarchical composition is a key Fractal feature that helps managing the complexity of understanding and developing component systems. Another important Fractal feature is its support for extensible reflective facilities. Specifically, each Fractal component contains a *controller* that embodies control behaviour associated with the component. The controller exposes a set of interfaces for inspecting and reconfiguring internal features of the component. Fractal defines a basic set of controller interfaces that can be extended as necessary. The basic set includes interfaces for managing the bindings of client interfaces, modifying the set of sub-components, and suspending/resuming component activities. Finally, Fractal includes an architecture description language (ADL) for specifying configurations comprising components, their composition relationships, and their bindings.

The Fractal/ProActive model extends Fractal in the following ways. Primitive components are specialised to obtain the properties of remotely accessible active objects. Composite components can contain multiple active objects and can be distributed over different machines. Component communication relies on asynchronous method invocations. A multicast communication style is also supported, analogous to the group communication mechanism in ProActive. Specifically, the model defines a specialisation of Fractal interfaces, called *multicast interfaces*, that enable treating a set of invocations as a single invocation. As with standard interfaces, multicast interfaces can have a client or server type. Finally, the component model supports configurable component deployment based on the deployment descriptors provided by ProActive.

4. Componentising Jem3D

Jem3D was componentised using the approach presented earlier. Most of the effort was spent on the architecture recovery activity because of the undocumented and degraded structure of the system. The run-time view of the original architecture was described using UML object diagrams—such as the one in Figure 1—and UML interaction diagrams. During the component architecture design, the launcher entity (an executing Java program) was decomposed into a *subdomain factory* component and an *activator* component; the former is assigned the responsibilities for creating, initialising, and connecting the subdomains, and the latter the responsibilities for obtaining the input data, passing them to the factory, and starting the computation. The reason for the decomposition was to make the factory reusable beyond Jem3D. A later iteration of the activity grouped the factory and the subdomains into a composite *domain* component, exploiting the hierarchical composition feature of Fractal/ProActive. Implementing the interface-based version served to increase confidence in the new component architecture and drastically simplified the final component-based implementation. The component-based implementation involved wrapping classes to form Fractal components and replacing a large part of the injector logic with Fractal ADL descriptions, as seen next.

Figure 3 shows the static structure of the resulting component-based Jem3D using a UML component diagram (multicast interfaces are represented as stereotyped UML interfaces with

special notation). The runtime configuration consists of multiple subdomains, logically arranged in a 3D mesh, with each subdomain connected to its neighbours via multicast interfaces. The runtime configuration also includes a dynamically varying number of steering agents. The main collector is connected to the current set of agents via a multicast interface. A multicast interface is also used to connect each agent to all other agents.

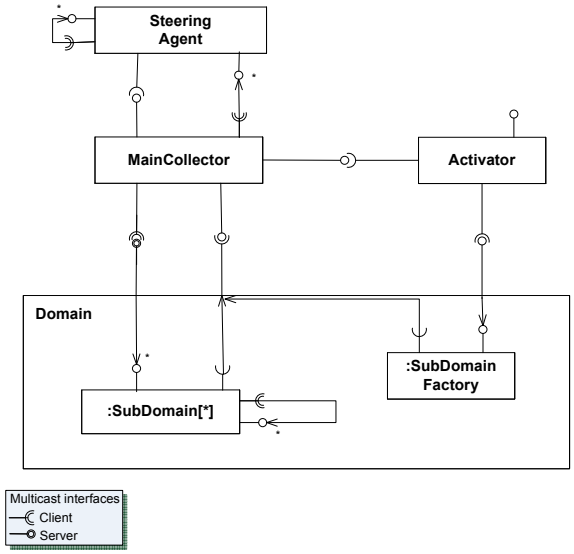

Figure 3. Component-based Jem3D structure

The initial configuration of Jem3D is described using the Fractal ADL, as seen in Figure 4 (pseudocode is used for brevity). Note that the ADL is not used to express the configuration of subdomains, which depends on the dynamically-determined domain division. Since allowable configurations follow a fixed, canonical structure in the form of a 3D mesh, a parameterised description would be useful for automatically generating subdomain configurations. However, the Fractal ADL includes currently no variability mechanisms for expressing such descriptions. The ADL does include a simple parameterisation mechanism, which is used to configure the factory with the required subdomain implementation.

```
Component ConsoleSteeringAgent
        definition = SteeringAgentImpl
Component MainCollector
        definition = MainCollectorImpl
Component Activator
        definition = ActivatorImpl
Component Domain
        Interface ...   // interfaces omitted
        Component SubDomainFactory
          Definition=FactoryImpl (SubDomainImpl)

        // bindings within composite
        // interfaces names omitted
        Binding This to SubDomainFactory
        Binding SubDomainFactory to This

// bindings among top-level components
// interface names omitted
Binding ConsoleSteeringAgent to MainCollector
Binding MainCollector to ConsoleSteeringAgent
Binding Activator to MainCollector
Binding Activator to Domain
Binding MainCollector to Domain
Binding Domain to MainCollector
```

Figure 4. Initial configuration in the ADL

Evaluation

We now examine whether the new, component-based Jem3D addresses the modifiability and reusability limitations of the original system. Owning to the componentisation process, the new system has gained reliable architectural documentation, which facilitates understanding and evolving the system. Moreover, an important part of the architecture—i.e., the initial component configuration—is captured in the ADL. As a result, the component platform can automatically enforce architectural structure on implementation, which helps reduce future architectural erosion. The use of provided and required interfaces as specified by the component model minimizes inflexible, hard-wired dependencies and allows flexible configuration after development time. Considering the scenario of changing the subdomain implementation, this can now be achieved simply by replacing a name in the ADL description (i.e., the SubDomainImpl name in Figure 4). Moreover, the domain component now serves as a reusable unit of functionality that supports the geometric decomposition pattern. Specifically, the component accepts as input the subdomain implementation and the domain division and embodies the logic to create and manage the runtime subdomain configuration.

5. Performance results

To assess the impact of componentisation on performance, we conducted experiments with the aim to compare execution times of the object-based and the component-based Jem3D versions. The experiments were performed on Grid'5000, a French experimental Grid platform currently featuring 2000 processors distributed over 9 geographical sites [10]. The sites host locally administered clusters connected through 1Gb/s links. Each experiment involved running the two

Jem3D versions for a given mesh size on the same number of processors allocated on up to 3 clusters of Grid'5000. Table 1 shows the mesh size and total number of processors used for each experiment.

Table 1. Jem3D experiments

Experiment	Mesh size	Number of Processors
1	41×41×41	20
2	81×81×81	70
3	201×201×201	130
4	201×201×201	138
5	201×201×201	258
6	241×241×241	258
7	241×241×241	308

Figure 5 shows the execution times for each experiment. We distinguish two kinds of execution time: (1) initialisation time, the time spent after deployment of the ProActive runtime and before the start of the calculation, and (2) computation time, the time spent performing the calculation. One can observe that execution times for the two versions are similar. As regards initialisation times, this result was unexpected as the component-based version creates a larger number of entities (e.g., the domain and factory components). Moreover, creating components is more costly than creating distributed objects due to the need to maintain extra meta-information. Initialisation times are similar probably because Fractal/ProActive incorporates optimisations absent from the ProActive library. Computation times are similar because the costs of subdomain communications are similar. This can be attributed to that the cost of remote object invocation outweighs any small overhead incurred by the component model. The domain component does impose an overhead on communications between the main collector and subdomains, but such infrequent communications have little impact on the calculation time. In summary, the results provide evidence that componentisation has no adverse impact on the performance of the Jem3D application.

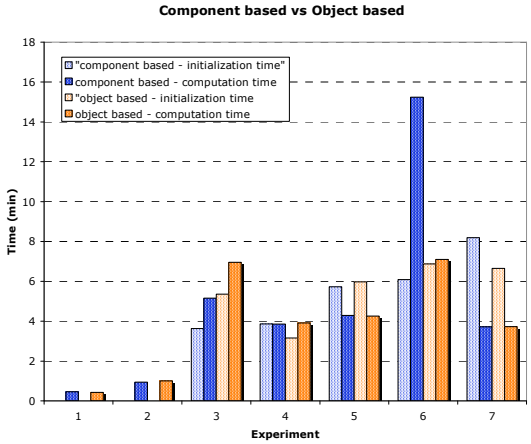

Figure 5. Comparison of execution times

6. Related work

As mentioned earlier, there is little experience in applying component based development to Grid computing. Most related work to ours is that associated to the CCA (Common Component Architecture) [11]. CCA is a component model for high-performance scientific computing that has been applied to a large range of application domains [8]. CCA components are dynamically connected through *provides* and *uses* ports. The main difference with Fractal is that CCA lacks hierarchical composition as a first-class part of the model. Ccaffeine [4] is an implementation of CCA that supports parallel computing. Ccaffeine-based components interact within a given process using CCA ports; parallel instances of Ccaffeine-based components interact across different processes using a separate programming model, typically MPI. XCAT3 [14] is another CCA implementation that supports components distributed over different address spaces and accessible as collections of Grid services compliant to OGSI (Open Grid Services Infrastructure). In [19], CCA/Ccaffeine is used to componentise simulation software for partial differential equations. Components are produced by creating thin wrappers over existing numerical libraries. A simple process for converting such libraries to components is presented in [5]; the process involves first grouping provided and used library functions to provides and uses CCA ports, and then deciding how ports are associated to components.

Beyond grid computing, several researchers have reported experiences with componentising large software systems. [15] describes the componentisation of operating system software for MPSoC (multi-processor system on chip) platforms. Componentisation relies on a lightweight Fractal implementation that targets embedded systems software. Other case studies have concentrated on componentising programmable controller software [17] and real-time telecommunication software [3]. Such work provides evidence of the positive effect of componentisation on modifiability but does not focus on the componentisation process.

Turning now to work related to the componentisation process, [1] describes re-engineering an existing Java program to obtain a implementation based on ArchJava. ArchJava is an extension of Java with language constructs that express components, ports, and connections. The adopted re-engineering process includes activities for identifying the source and target architectures, refactoring the original program, and then migrating it to the ArchJava environment. The process is thus similar to the one presented in this paper, but it is specific to ArchJava.

[16] and [13] present reengineering methods for migrating from object-oriented systems to component-based systems. The methods rely on clustering techniques, metrics, design rules, and heuristics. Both these methods address only a part of the proposed process—that is, the component architecture design activity—and they can be accommodated within our process. [2] presents a tool that transforms industrial C++ code to a proprietary variant of CCM. Engineers need only to specify how class methods are factored into provided interfaces, and the tool generates component-based code that preserves the original functionality. Manual tuning of the generated code is also typically necessary. This tool cannot assist in architectural enhancements of the original system. Similar transformation technology can be applied within the proposed componentisation process for supporting the migration from the restructured object-oriented version to the final component-based system.

7. Conclusion

This paper has presented a case study in reengineering a scientific application into a component-based, grid-enabled application built on Proactive/Fractal. The transformation from an object-based to a component-based system has followed a general componentisation process, reusable in other contexts. The paper has provided qualitative evidence that componentisation using Fractal/ProActive is beneficial to the modifiability and reusability of the application. The paper has also provided quantitative evidence that componentisation has no adverse effect on performance.

There are two main directions for future work. First, we plan to apply the componentisation process and the Fractal/ProActive component technology to other applications in diverse domains. Such work will enable a more complete assessment of their usefulness and usability, and generate further suggestions for improvement. Second, we plan to add support for dynamic reconfiguration in the component-based Jem3D application in order to accommodate variations in the availability of underlying resources. Supporting reconfiguration will involve the introduction of manager components that build on the reconfiguration primitives already provided by the component model (e.g., connect or disconnect components), without requiring any change to existing code.

8. References

[1] M. Abi-Antoun, and W. Coelho, “A Case Study in Incremental Architecture-Based Re-engineering of a Legacy Application”, 5th Working IEEE/IFIP Conference on Software Architecture (WICSA-5), 2005.

[2] R. Akers, I. Baxter, M. Mehlich, B. Ellis, K. Luecke, “Re-engineering C++ Component Models Via Automatic Program Transformation”, Twelfth Working Conference on Reverse Engineering, IEEE, 2005.

[3] H. Algestam, M. Offesson, L. Lundberg, "Using Components to Increase Maintainability in a Large Telecommunication System," Ninth Asia-Pacific Software Engineering Conference (APSEC'02), 2002, p. 65.

[4] B.A. Allan, R.C. Armstrong, A.P. Wolfe, J. Ray, D.E. Bernholdt, and J.A. Kohl, “The CCA core specifications in a distributed memory SPMD framework,” *Concurrency Comput. Pract. Exp.*, no. 5, 2002, vol. 14, pp. 323-345.

[5] B.A. Allan, S. Lefantzi, J.Ray, "ODEPACK++: Refactoring the LSODE Fortran Library for Use in the CCA High Performance Component Software Architecture," Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04), 2004, pp. 109-119.

[6] L. Baduel, F. Baude, D. Caromel, C. Delbe, S. Kasmi, N. Gama, and S. Lanteri. “A Parallel Object-Oriented Application for 3D Electromagnetism”, 18th International Parallel and Distributed Processing Symposium, IEEE Computer Society, Santa Fe, New Mexico, USA, April 2004.

[7] F. Baude, D. Caromel, and M. Morel. “From distributed objects to hierarchical grid components”, In International Symposium on Distributed Objects and Applications (DOA), Springer, Catania, Italy, volume 2888 of LNCS, 2003, pages 1226 – 1242.

[8] D.E. Bernholdt, B.A. Allan, R. Armstrong, F. Bertrand, K. Chiu, et al., “A Component Architecture for High Performance Scientific Computing”, ACTS Collection special issue, *Intl. J. High-Perf. Computing Applications,* 20 (2006)

[9] E. Bruneton, T. Coupaye, and J. B. Stefani, “Recursive and dynamic software composition with sharing”, In Proceedings of the Seventh International Workshop on Component-Oriented Programming (WCOP2002), 2002.

[10] F. Cappello, F. Desprez, M. Dayde, E. Jeannot, et al, “Grid'5000: A Large Scale, Reconfigurable, Controlable and Monitorable Grid Platform”, 6th IEEE/ACM International Workshop on Grid Computing, Grid'2005, Seattle, Washington, USA, November 13-14, 2005.

[11] CCA Forum Home Page, The Common Component Architecture Forum, 2004. http://www.cca-forum.org.

[12] Grid Component Model (GCM) Proposal, CoreGRID Deliverable, D.PM.002, Nov. 2005.

[13] S.D. Kim, S.H. Chang, "A Systematic Method to Identify Software Components", 11th Asia-Pacific Software Engineering Conference (APSEC'04), 2004, pp. 538-545.

[14] S. Krishnan and D. Gannon. “XCAT3: A Framework for CCA Components as OGSA Services”, 9th Intl Workshop on High-Level Parallel Programming Models and Supportive Environments, IEEE Computer Society Press, 2004.

[15] O. Layaida, A.E. Özcan, and J.B. Stefani. “A Component-based Approach for MPSoC SW Design: Experience with OS Customization for H.264 Decoding”, 3rd Workshop on Embedded Systems for Real-Time Multimedia under CODES+ISSS, New York, USA, 2005.

[16] E. Lee, B. Lee, W. Shin, C. Wu, "A Reengineering Process for Migrating from an Object-oriented Legacy System to a Component-based System", 27th Annual International Computer Software and Applications Conference, 2003, p. 336.

[17] F. Lüders, I. Crnkovic, P. Runeson, “Adopting a Component-Based Software Architecture for an Industrial Control System – A Case Study, Component-Based Software Development for Embedded Systems”, Springer, LNCS 3778, ISBN: 3-540-30644-7, 2005, p 232-248

[18] B.L. Massingill, T.G. Mattson, and B.A. Sanders. “Patterns for parallel application programs”, In Proceedings of the Sixth Pattern Languages of Programs Workshop (PLoP99), 1999

[19] B. Norris, S. Balay, S. Benson, L. Freitag, P. Hovland, L. McInnes and B. Smith, “Parallel components for PDEs and optimization: some issues and experiences”, *Parallel Computing*, Volume 28, Issue 12, December 2002, pp 1811-1831.

[20] Object Management Group, CORBA Component Model v3.0, OMG Document formal/2002-06-65.

[21] ProActive web site, ttp://www.inria.fr/oasis/ProActive/