View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by WestminsterResearch

WestminsterResearch
http://www.wmin.ac.uk/westminsterresearch

Advanced Grid programming with components: a biometric identification case study

Thomas Weigold[1]
Peter Buhler[1]
Jeyarajan Thiyagalingam[2]
Artie Basukoski[2]
Vladimir Getov[2]

[1] IBM Zurich Research Laboratory
[2] Harrow School of Computer Science

Copyright © [2008] IEEE. Reprinted from the proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference, 28 July - 1 August 2008, Turku, Finland: COMPSAC 2008. IEEE, Los Alamitos, USA, pp. 401-408. ISBN 9780769532622.

This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of the University of Westminster's products or services. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE. By choosing to view this document, you agree to all provisions of the copyright laws protecting it.

The WestminsterResearch online digital archive at the University of Westminster aims to make the research output of the University available to a wider audience. Copyright and Moral Rights remain with the authors and/or copyright owners.
Users are permitted to download and/or print one copy for non-commercial private study or research. Further distribution and any use of material from within this archive for profit-making enterprises or for commercial gain is strictly forbidden.

Whilst further distribution of specific materials from within this archive is forbidden, you may freely distribute the URL of the University of Westminster Eprints (http://www.wmin.ac.uk/westminsterresearch).

In case of abuse or copyright appearing without permission e-mail wattsn@wmin.ac.uk.

Advanced Grid Programming with Components: A Biometric Identification Case Study[1]

Thomas Weigold, Peter Buhler
{twe, bup}@zurich.ibm.com
IBM Zurich Research Laboratory

Jeyarajan Thiyagalingam, Artie Basukoski, Vladimir Getov
{jeyarat, A.Basukoski02, V.S.Getov}@westminster.ac.uk
University of Westminster, London, UK

Abstract

Component-oriented software development has been attracting increasing attention for building complex distributed applications. A new infrastructure supporting this advanced concept is our prototype component framework based on the Grid component model. This paper provides an overview of the component framework and presents a case study where we utilise the component-oriented approach to develop a business process application for a biometric identification system. We then introduce the tools being developed as part of an integrated development environment to enable graphical component-based development of Grid applications. Finally, we report our initial findings and experiences of efficiently using the component framework and set of software tools.

1. Introduction

In recent years component technologies have become a paradigm of choice in state-of-the-art software construction. Various component models such as the CORBA Component Model (CCM) [1], the Enterprise Java Beans (EJB) [2], or the Distributed Component Object Model (DCOM) [14] have been available for different operating environments. Even though many existing component models can be applied in distributed systems, they do not address the Grid infrastructure issues in their full depth. For example, in most models a single component cannot itself be distributed and thus it cannot be used to abstract the complexity of a distributed application computation spanning multiple computers.

Furthermore, the capabilities of the underlying component model vary significantly between different frameworks. For example, the Common Component Architecture (CCA) [3] does not support hierarchical composition nor does it provide extensive support for component management and dynamic reconfiguration. Other frameworks such as ASSIST [19] support autonomic management but are not component based. Further, there is no integrated development environment to support these component models, especially addressing the contexts of Grid computing.

As a result, the design of a new Grid Component Model (GCM) has been brought forward in the context of the European project CoreGRID [4]. GCM provides a much higher level of abstraction than contemporary component frameworks and it explicitly takes Grid specific issues such as the programmability of large-scale, heterogeneous, and dynamic Grid infrastructures into account. To drive the GCM ideas developed under the CoreGRID project one step further the GridCOMP (Grid Programming with Components) project was established in 2006 [5]. Its main goal is the design and implementation of a component-based framework suitable to support the development of efficient Grid applications. An integrated development environment for the Grid (GIDE) is being developed to support this framework. Here, the basic GCM architecture defined in CoreGRID is used as the starting point. We then demonstrate the development of a biometric identification system case study, using the GIDE, to highlight the advantages of the approach.

This paper provides an overview of the ongoing work and some initial results of the GridCOMP project. After introducing the current GridCOMP GCM framework implementation in Section 2, the biometric identification case study is discussed in Section 3. Afterwards, Section 4 introduces an integrated development framework designed to best exploit the Grid component platform. Section 5 describes our initial experiences. Finally, we close with conclusions and future work in Section 6.

2. The GCM Framework

The core task within the GridCOMP project is to further refine the GCM developed within CoreGRID and to produce a component framework implementation (CFI) acting as a reference prototype platform. Here, ProActive [6] is used as the basic Grid middleware on top of which the CFI is realised. The following sub-sections briefly

[1] This research work is carried out under the FP6 GridCOMP project partially funded by the European Commission (Contract FP6-034442).

0730-3157/08 $25.00 © 2008 IEEE
DOI 10.1109/COMPSAC.2008.97

IEEE computer society

introduce ProActive and outline the main GCM features currently being implemented.

2.1. The ProActive Middleware

ProActive is an open source Java library providing a toolkit that simplifies the programming of parallel, distributed, and multi-threaded applications for Grids. It is based on the pattern of active objects (AO) and asynchronous method calls with implicit futures [7]. An AO is a remote object with its own thread that sequentially processes calls received on its public methods. Pending method calls are stored in a request queue. Such method calls towards AOs are asynchronous and, if non-void, return so-called future objects as a result. If a future object is accessed the caller is automatically blocked until the result is available. This implicit synchronisation mechanism is known as wait-by-necessity. Internally, ProActive implements a meta-object protocol and uses Java RMI as a portable transport layer to provide this functionality while hiding its underlying complexity. As a result, an AO appears as a normal, transparently remote, Java object to the developer.

Furthermore, to strictly separate the development of AOs from their execution on a particular physical infrastructure, ProActive provides a deployment framework based on the virtual nodes (VN) concept. VNs are used in the source code of ProActive applications as an abstraction defining where to locate AOs. The mapping from VNs to Java virtual machines (JVM), their creation mechanisms, and real machines is then defined in an infrastructure dependent deployment descriptor file in XML (Extensible Markup Language) format [6].

However, ProActive does not provide higher-level of abstraction like distributed Grid components since AOs are not hierarchical and can not be distributed themselves. Therefore, it is currently being extended by the GridCOMP CFI as described below.

2.2. Grid Component Framework

The main technical features of the GridCOMP component framework can be summarised as follows:

- Support for primitive and composite distributed components and hierarchical composition.
- Components specification in XML format.
- Collective interfaces to comply with Grid specific multi-way communication requirements.
- A comprehensive run-time API.
- Extensive support for non-functional aspects such as component control and autonomicity.
- Advanced component deployment via the notion of VNs.
- An XML schema for component packaging.

To turn these features into reality the Fractal component model [8] has been chosen as the basis for the definition of GCM. Consequently, the CFI can be considered an implementation of the Fractal specification with a number of Grid specific extensions.

Fractal basically consists of a general conceptual model, an XML-based architecture description language (ADL) used to define component systems along that model, and a runtime API. The model defines components to consist of content, controller, interfaces (client, server, or control), and bindings. Depending on their content, components are either primitive or composite. Figure 1 illustrates how a composite component built from two primitive components is modelled in Fractal.

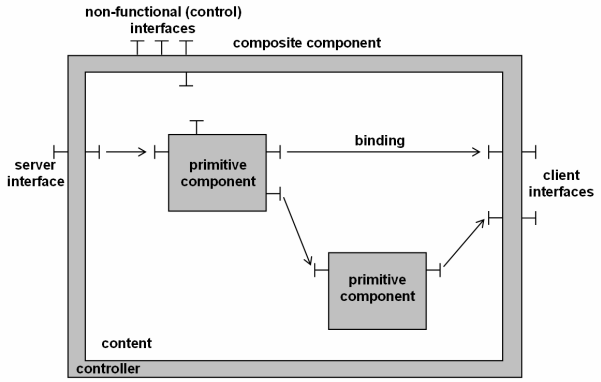

Figure 1: Fractal model of a composite component

GCM extends this model by allowing that all components can be arbitrarily distributed. For example, all three components shown in Figure 1, the composite and the two primitives, could run on different remote nodes. To achieve this, the distributed deployment facilities offered by ProActive are reused, and the notion of virtual nodes is integrated in the component ADL. This means, component ADL definitions have to be associated with a deployment descriptor. The CFI provides a number of ADL extensions to support this. For instance, VNs can be referenced in component definitions and cardinality attributes are available to further control parallelism and distribution when mapping components to real nodes. Also, VNs can be renamed to adjust different ADLs.

To implement this extended Fractal model within ProActive the component model must be mapped to the AO pattern. This is achieved by representing each component by one AO and extending the meta-object protocol with controller objects implementing non-functional component control. Additionally, an extended version of the Fractal run-time API is being provided. It allows the manipulation of components at execution and includes facilities for legacy code wrapping, which turns legacy code into GCM components.

2.3. Collective Interfaces

A further grid specific Fractal extension introduced by GCM is the notion of collective interfaces [9]. This eases parallel programming and allows exposing the collective behaviour of a component on its interface level.

Collective interfaces correspond to new cardinalities for interfaces, multicast or gathercast, representing one-to-many or many-to-one communication, respectively. In the CFI the collective behaviour of an interface can be defined via Java annotations on class level as well as on method level. Additionally, the desired data distribution mode (broadcast, one-to-one, or round-robin) can be defined for multicast interfaces. Figure 2 shows how collective interfaces are modelled in GCM and how invocation parameters are distributed and aggregated using the one-to-one (or scatter) mode.

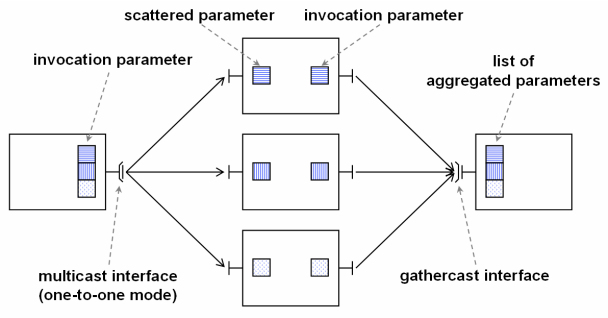

Figure 2: Multicast/gathercast interfaces in GCM

Collective interfaces allow managing a group of interfaces as a single entity while the CFI takes care of parallel invocations, data distribution, and synchronisation. This further simplifies the design, programming, and (re)configuration of a component system.

2.4. Non-Functional Aspects

Non-functional aspects are targeting those features of the GCM which are contributing to the efficiency and reliability of the components in obtaining functional results but are not directly involved in result computation. The CFI provides mechanisms for autonomic component management that can be used to deal with such non-functional issues. Here, dynamic reconfiguration of component properties or relationships is the prime example.

Again, the Fractal model is extended to support component autonomic control. As indicated in Figure 1, the "membrane" (controller) of a component already exposes some non-functional interfaces, for instance, for binding and life-cycle control. Here, the CFI adds an autonomic behaviour control (ABC) server interface. Via this interface, a component can expose a set of reconfiguration actions which can be triggered by its environment. Additionally, a component can have an autonomic manager (AM) implemented as a dedicated sub-component that has some rules for autonomic reconfiguration. The AM can interact with other AMs of other components and it uses the ABC to trigger reconfiguration actions. A component just exhibiting ABC is called passive whereas a component also exhibiting an AM is called active with respect to autonomic control [10]. Finally, the CFI includes a number of so-called behavioural skeletons, which can be used for component composition including ABC/AM implementations for application-specific reconfiguration strategies [13].

3. Case Study: A Biometric Identification System

In recent years biometric methods for verification and identification of people have become very popular. Applications span from governmental projects like border control or criminal identification for civil purposes such as e-commerce, network access, or transport. Frequently, biometric verification is used to authenticate people meaning that a 1:1 match operation of a claimed identity to the one stored in a reference system is carried out. In an identification system, however, the complexity is much higher. Here, a person's identity is to be determined solely on biometric information, which requires matching the live scan of biometrics against all enrolled (known) identities. Such a 1:N match operation can be quite time-consuming making it unsuitable for real-time applications.

In order to tackle this challenge, one of the use cases developed to evaluate the GridCOMP CFI is a biometric identification system (BIS). Its goal is to build a real-time biometric identification system, based on fingerprint biometrics, which can work on a large user population of up to millions of individuals. To achieve real-time identification within a few seconds period our BIS application takes advantage of the Grid via GCM components.

3.1. BIS Architecture

The BIS use case can be considered a business-process or workflow-driven application. Figure 3 outlines its high-level architectural design.

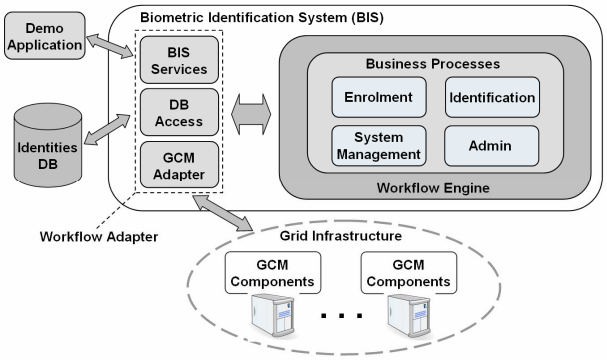

Figure 3: BIS high-level architecture

The BIS is built around a workflow execution engine acting as the central control unit of the system. A number of business processes are implemented as workflow scripts running within the engine. The processes comprise functionality accessible from the demo application (e.g. enrolment, identification) as well as internal system management logic required to control the distributed biometric matching. Furthermore, the BIS provides a number of adapters to the workflow engine such that the business processes can interact with external entities, namely, the database (DB) storing information about enrolled identities, and the interface to the Grid infrastructure.

The workflow engine used in our implementation is the embeddable process virtual machine (ePVM) [11], which is available as a Java library such that it can be easily incorporated into the BIS. Its process model is rooted in the theoretical framework of communicating state machines and its process definition language is JavaScript. Consequently, in ePVM each workflow script represents a state machine implemented in JavaScript. The core logic of the BIS application is defined as a number of such ePVM scripts.

The three workflow adapters, as indicated in Figure 3, consist of Java classes implementing a particular interface such that they can be registered with the ePVM engine. Once registered, they can receive messages from workflow scripts and they can send reply messages. This way the processes defined in JavaScript can interact with Java functionality external to the workflow engine.
The *BIS Services* adapter acts as the interface to external applications making use of the identification system whereas the *DB Access* adapter encapsulates identity DB related functionality. Finally, the *GCM Adapter* provides access to the Grid infrastructure. It is triggered by the workflow scripts and offers functionality to deploy nodes and GCM components, analyze the biometric matching performance of the BIS Grid, distribute the database across the GCM components, and to submit biometric information for distributed identification.

3.2. Component Architecture

In this sub-section we describe the Grid component architecture that allows the GCM adapter to provide the functionality described so far. Figure 4 shows the overall component design, the bindings between components, and their deployment to the physical grid infrastructure.

The basic approach is to have one component encapsulating the biometric matching functionality, which is then deployed on all nodes in a SPMD-style setting. This component is named *CompIDMatcher* and it is a composite component built from two primitive ones, *CompAlgControl* and *CompAlg*. The latter represents the biometric algorithm and, as indicated in Figure 4, it makes use of a native library containing the actual fingerprint matching code via the Java Native Interface (JNI).

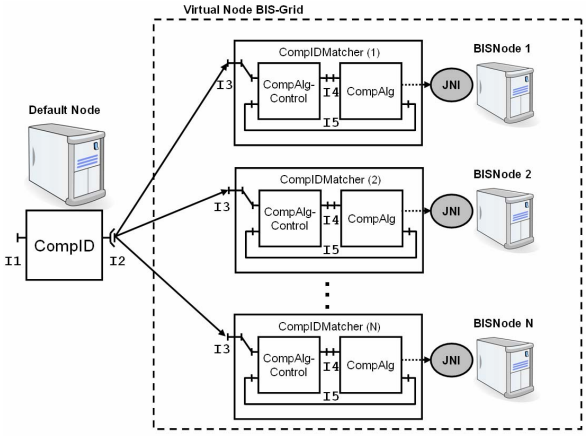

Figure 4: Component design, bindings, and deployment

The purpose of the *CompAlgControl* component is to maintain the state of the identification process and to provide a second control flow such that the *CompIDMatcher* component remains responsive while the *CompAlg* component is busy with the actual matching work.
This setting has to do with fact that in ProActive each GCM component is represented by one AO. Furthermore, each AO is by default single threaded. In other words, there is only one thread that processes queued method invocations. Consequently, the *CompAlg* component cannot respond to status requests while it processes an identification request. With the *CompAlgControl* component a second thread is introduced, which can process requests received via interface I3 while the *CompAlg* component might do biometric matching within the JNI library. The *CompAlg* component updates the current state maintained in *CompAlgConntrol* regularly via interface I5. Requests received via the two interfaces I3 and I5 are synchronized automatically via the request queue of *CompAlgControl's* AO.

To exemplify how a composite component is defined via the GCM ADL, Listing 1 shows the definition of the *CompIDMatcher* component. Firstly, the two interfaces (client and server) of the component are defined referencing their Java interface definitions (I3). Secondly, the ADL files of the two inner components are referenced. Thirdly, the three bindings within *CompIDMatcher* are defined. Finally, the last two tags define the component to be a composite one, which is to be created on the *BIS-Grid* VN as one instance per real node.

```
<definition name="com.ibm.bis.CompIDMatcher">
  <interface name="server" role="server" signature="com.ibm.bis.I3"/>
  <interface name="client" role="client" signature="com.ibm.bis.I3"/>
  <component name="CompAlgControl"
        definition="com.ibm.bis.CompAlgControl"/>
  <component name="CompAlg" definition="com.ibm.bis.CompAlg"/>
  <binding client="this.client" server="CompAlgControl.idServer"/>
  <binding client="CompAlgControl.client" server="CompAlg.server"/>
  <binding client="CompAlg.client"
      server="CompAlgControl.controlServer"/>
  <controller desc="composite"/>
  <virtual-node name="BIS-Grid" cardinality="single"/>
</definition>
```

Listing 1: Composite component ADL example

When the ADL is applied it must be associated with a deployment descriptor XML file defining a VN named *BIS-Grid*. Within the deployment descriptor the VN is mapped to a number of real nodes represented by BISNode 1-N in Figure 4. Also, the creation protocols, for instance, rlogin or SSH, used to create the nodes are defined. Other features such as automatic file transfer, here used to distribute the JNI library, are available in the CFI. As a result, when the GCM adapter activates the deployment descriptor the CFI automatically starts JVMs on the nodes and transfers the JNI library.

The *CompID* component, with its server interface I1, represents the main interface to external entities using the Grid infrastructure. The GCM adapter uses this interface to interact with the component system, for instance, to submit live scan data of a person for identification. Additionally, the *CompID* component provides the multicast client interface I2 as shown in Figure 4 where all the distributed *CompIDMatcher* instances are bound to. It is used to broadcast, among others, identification requests via the method *identify()* as shown in Listing 2.

```
public interface I2 {
        ...
@MethodDispatchMetadata(mode=@ParamDispatchMetadata(
mode=ParamDispatchMode.BROADCAST))
public List<IntWrapper> identify(LiveScan liveScan);
}
```

Listing 2: Multicast interface definition

The interface definition is annotated with the data distribution mode BROADCAST. Additionally, in the ADL file defining the *CompID* component, interface I2 is defined with the attribute *cardinality="multicast"*. As a result, the CFI translates each invocation of the identify method into a number of parallel method invocations depending on the number of components bound to I2. Also, for each invocation a copy of the *liveScan* parameter is used due to the broadcast mode. The CFI automatically gathers the results into a list of future objects of type *IntWrapper*. Accordingly, the definition of interface I3 is similar to I2 except that the identify method returns just a single object of type *IntWrapper* instead of a list. This way, a significant part of the concurrent programming task is hidden behind the multicast interface and parallel invocations as well as synchronisation are handled by the CFI.

3.3. Autonomic Management

The initial BIS architecture as presented above does not address the problem of dynamic data re-distribution in case of changes in the database or changes in the Grid infrastructure. Therefore, it has been revised to take advantage of the autonomic management features offered by the CFI. Here, the concept of behavioural skeletons (BS) [10] is used to reduce development effort. BS are parametric composite components implementing typical reconfiguration strategies such as self-optimization, self-healing, or self-configuring. For the revised BIS implementation we use the task-parallel farm BS included in the CFI. As outlined in Figure 5, the farm skeleton includes one or more worker components all bound to a collective interface (unicast) port. Tasks received via this port are distributed to the workers in a round-robin fashion. Furthermore, the farm is equipped with an ABC that allows dynamic increase or decrease of the number of workers using a pool of nodes defined in a given deployment descriptor. Finally, the farm includes an AM which makes use of the ABC in accordance with a certain autonomic management strategy. The default strategy is to increase/decrease the number of workers if the average service time for the tasks reaches certain thresholds. The threasholds are defined by a QoS contract, here the desired farm performance defined in tasks/second, which can be submitted to the farm.

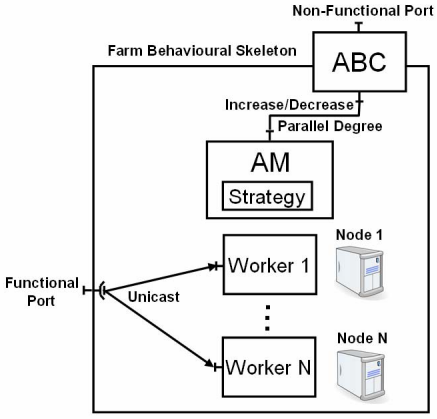

Figure 5: Farm Behavioural Skeleton

In order to make use of the farm skeleton for the BIS application it has to be parameterised appropriately. Firstly, we define the workers as instances of the *CompIDMatcher* component. Secondly, we inject the desired QoS contract into the AM such that it autonomically adjusts the parallel degree depending on the performance of the farm. Finally, we split the identification process into tasks where each task represents a part of the database to be searched. The tasks are then submitted to the farm. Here we assume that all workers have access to a shared database. This change in

the design is due to the fact that the farm skeleton does not consider the case where workers carry state as required in the initial BIS architecture (c.f. Section 3.2). At the time of writing, an additional (data-parallel) skeleton which allows injecting state into the workers during reconfiguration is being added to the CFI to address this issue. Anyhow, the task-parallel farm BS provides a lot of off-the-shelf functionality for the autonomic version of the BIS, which otherwise would have to be implemented from scratch. As a result, the autonomic version required about the same amount of source code to be written as the initial version while adding a substantial amount of functionality.

4. The Grid Integrated Development Environment

The task of effectively using different programming and/or component models becomes less challenging if the developer is assisted by an appropriate tools framework. Therefore, the Grid integrated development environment (GIDE) is being developed along with the CFI. It is targeted towards providing the necessary functionalities for developing GCM-based applications. The vision of the GIDE is to support the user in all aspects relevant to the development phase as well as the post-development phase. For this purpose, the GIDE provides the following features:

- Graphical composition of GCM component-based applications
- Deployment of applications
- Component and resource monitoring
- Steering of components and applications.

Furthermore, the philosophy of the GIDE is to provide enhanced support with user-friendly graphical interface while enabling direct code editing. This means that a developer can freely switch between graphical development and direct coding of the required artefacts.

The GIDE is based on the Eclipse framework and is developed as a plug-in to it [12]. This approach ensures that developers can still benefit from the Eclipse functionalities such as the plain Java development tools while leveraging the GCM-specific features. In addition, the GIDE is facilitated by the Eclipse-specific application programming frameworks, interfaces and libraries, including the Eclipse Modelling Framework and Eclipse Graphical Editing Framework. The functionalities of the GIDE are provided as different perspectives – an Eclipse-specific method for grouping a set of functionalities as a graphical view. The following sub-sections describe different perspectives within the GIDE that support application composition, deployment, monitoring, and steering.

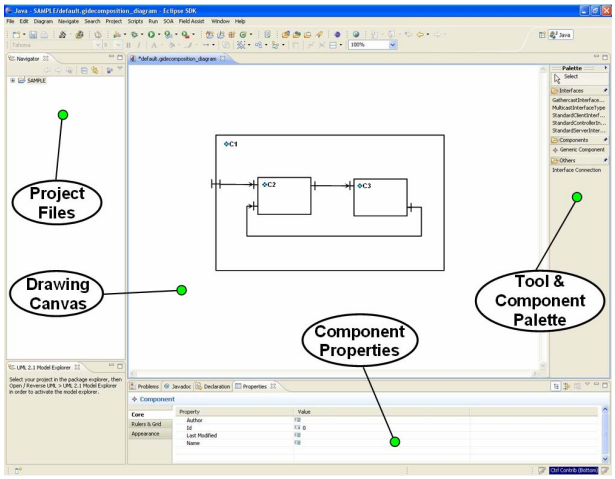

Figure 6: The Composition perspective of the GIDE

4.1 Composition Perspective

The composition view is one of the essential features where the application development life-cycle begins. The composition perspective provides a toolbox with a list of components and with a set of tools so that applications can be visually composed. The toolbox is a mutable collection which is used as a repository of frequently used components. The back-end of the composition perspective generates necessary GCM-specific development artefacts such as ADL files and Java interfaces (c.f. Listing 1/2). Figure 6 shows a general screen layout of the composition perspective while defining the *CompIDMatcher* component during the case study (c.f. Section 3.2). The composition perspective also supports importing from and exporting to these artefacts.

4.2 Monitoring Perspective

The monitoring perspective provides the views that data centre operators need in order to properly monitor the environment in which components operate. Three types of monitoring are provided in order to enable proper management of applications and thus compositions. Firstly, resources monitoring of hosts is supported. This includes monitoring CPU utilization of hosts, storage space, and other platform specific status information. Secondly, monitoring of the GCM components is supported to provide status and location information, for instance, to verify component co-allocation as desired in the BIS use case. The monitoring is enabled with a zoom-in feature for monitoring sub-components. In our example one could zoom into the component design as shown in Figure 4, and visualize the inner composite components and status information of them as depicted in Figure 6. Finally, this perspective allows monitoring of active objects, which is necessary for developers to debug and monitor applications during the development phase.

4.3 Deployment Perspective

This perspective consists of views needed for application deployment. The main view is of a deployment descriptor editor to map physical hosts and JVMs to VNs. For instance, in the case study this has been used to define the BIS-Grid VN and for mapping the BISNodes 1-N to real machines (c.f. Figure 4). A developer may have a set of these deployment descriptors to be used for different hardware configurations. To complement this view, a view of the hosts and their resource statuses is also provided, giving a developer the ability to associate sets of hosts with each deployment descriptor. Within the deployment perspective the operator is able to launch components simply via drag-and-drop operations before moving on to steering.

4.4 Steering Perspective

The steering perspective is especially useful for data centre operators. The perspective builds on the resource and component monitoring view and graphically shows the components location and their status. An additional view shows the geography and resource availability of the hosts, VNs, as well as the components that are running on them. Based on these views, the operator has the facility to start, stop, and relocate components from one VN to another while monitoring their status to ensure correct execution.

5. Initial Experiences

The BIS use case demonstrates that the CFI offers a very high level of abstraction hiding the complexities of Grid programming. This is due to a number of advanced features such as the notion of VNs, the hierarchical nature of the model, and most notably the concept of collective interfaces, implicit futures, and behavioural skeletons. As shown in the use case, it clearly eases Grid programming and represents a significant step towards the "invisible" Grid vision. In particular, the autonomic version of the BIS use case has shown that a comprehensive set of behavioural skeletons for common patterns can significantly reduce the development effort.

Furthermore, a distinct strength of the framework is the strict separation of concerns. In particular, the notion of VNs, the concept of deployment descriptors, and the ADL component definitions allow separating the design infrastructure from the deployment (physical) infrastructure. For example, the BIS GCM components only rely on a VN named *BIS-Grid* to be present no matter which and how many real nodes are behind it. This is true for both the ADL files and the Java code. Additionally, interfaces, bindings, and content are also clearly separated and explicitly named such that relationships are ideally not hidden in code, potentially leading to better software design, documentation, and maintainability.

Also, hierarchical composition allows composing new components or complete applications from existing ones almost without writing code. The *CompIDMatcher* component, for instance, is generated by the CFI automatically at the time its ADL definition is applied; only its interfaces have been defined in Java files. Overall, the advanced support for Grid specific needs supplied by the component framework and the GIDE did clearly speed up the development cycle of the BIS.

The use case also showed that the rich feature set offered by the CFI and the strict separation of concerns does not come without a price. One of the drawbacks that became visible was the large number of development artefacts to be generated and maintained. Quite a number of interface definitions, ADL files, deployment files, and other Java files including the actual functional code were required considering that the component system of the BIS application is relatively simple. This can turn manual code re-factoring such as simply renaming an interface into a fairly complex task since interface names appear in Java files as well as in ADL files. Only a powerful toolkit such as the GIDE outlined in the previous Section can help resolving these issues.

Furthermore, the use case revealed some minor issues to be fixed in the next CFI release. For example, there is no adequate ADL support for the definition of co-allocation constraints. This would provide an easy way of defining that inner components, for instance, *CompAlgControl* and *CompAlg*, must always be co-allocated with their surrounding composite. In case of the BIS component design this is obviously desirable for performance reasons. Also, it turned out that the current version of the ADL does not support creating and binding an arbitrary number of components, for instance, as many as nodes available in a given VN, to a multicast interface. The number of nodes must be known *a priori* when writing the ADL file since the bindings are statically defined there. The case study also revealed that the early version of the farm behavioural skeleton does not consider workers to carry state. In our view, the skeleton should be extended to that end since many real-world applications might bear this requirement.

Overall, the integration of the Grid components into the workflow driven BIS use case went smoothly. However, it must be noted that business process engines such as ePVM follow an event-driven paradigm which needs to be adapted to work with the wait-by-necessity approach of the CFI. In the use case the GCM adapter implemented this adaptation by turning available future objects into JavaScript messages sent to the workflow engine.

Finally, the development work showed that the GIDE is a key part of the process for developing Grid applications which provided the basis for accelerated development. In addition to the support for ADL file generation from

graphical compositions, the monitoring perspective has proven to be very useful throughout the case study for verifying the actual component distribution/co-allocation. Although the GIDE could have been developed as a separate application, the approach of developing it as a pluggable module for Eclipse had several advantages including reduced development time, increased features, and consistent look-and-feel for developers.

A number of initiatives exist in producing development environments for Grid applications, such as the GriDE [17] and Sun Grid Model for NetBeans [18]. However, their underlying component models are not aligned with the needs of Grid computing. The GIDE varies from existing frameworks by providing explicit support for Grid specific needs as well as support for different user groups ranging from software developers to data centre operators.

6. Conclusions and Future Work

Adopting a modularised or component-based approach is the key to developing large-scale software systems [15, 16]. The component model we adopted for the framework implementation, GCM [4], supports hierarchical inclusion and is specifically designed for addressing potential issues of Grid computing.

In this paper we have considered a potentially useful method for developing Grid applications and demonstrated the applicability of the method using an industry-strength case study – a biometric identification system. Our experience so far provides a basis for foreseeing the future of developing and deploying advanced component-based Grid applications. In discussing the approach along with the case study, we have made the following contributions:

- We provided a brief overview of the Grid-specific component model – the GCM, developed as part of the GridCOMP project and the current status.
- We presented a use case application, the distributed biometric identification system, implemented using GCM components.
- We have outlined the approach for developing a GCM-specific IDE while supporting existing development strategies.
- We presented the overall functionalities and features of the GIDE to support the development of GCM-based Grid applications.
- We presented our initial experiences and findings in using the GCM framework and the GIDE.

Although initial prototypes of the CFI, the use case applications, and the GIDE have been implemented, a number of issues remain to be investigated:

- Further explore other behavioural skeletons included in the CFI with respect to the use case.
- Enhance the GIDE to support debugging of distributed applications and language-specific code generation.
- Explore the possibilities of validating compositions for a well specified domain of application.

References

[1] Object Management Group (OMG). The CORBA Component Model. Revision V4.0, 2006.

[2] B. Burke, R. Monson-Haefel. Enterprise JavaBeans 3.0. O'Reilly Media, 2006, ISBN 9780596009786.

[3] R. Armstrong et. al. Toward a Common Component Architecture for High-Performance Scientific Computing. Proc. of HPDC Conference, 1999.

[4] CoreGrid NoE, Institute on Programming Model. Deliverable D.PM.04 – Basic Features of the Grid Component Model, 2007.

[5] GridCOMP – Effective Components for the Grid, 2006, http://gridcomp.ercim.org/.

[6] The ObjectWeb consortium. ProActive – Programming, Composing, Deploying on the Grid. http://www-sop.inria.fr/.

[7] D. Caromel, L. Henrio. A Theory of Distributed Objects. Springer 2005, ISBN 978-3540208662.

[8] E. Bruneton, T. Coupaye, J. B. Stefani. The Fractal Component Model. Technical report, ObjectWeb Consortium, February 2004, http://fractal.objectweb.org/.specification/index.html.

[9] F. Baude, D. Caromel, L. Henrio, M. Morel. Collective Interfaces for Distributed Components. Proc. of CCGrid Conference, 2007.

[10] M. Aldinucci et. al. Behavioural skeletons in GCM: autonomic management of Grid components. Proc. of PDP Conference, 2008.

[11] T. Weigold, T. Kramp, P. Buhler. ePVM – An Embeddable Process Virtual Machine. Proc. of COMPSAC Conference 2007.

[12] Eclipse – An Open Development Platform, http://www.eclipse.org/.

[13] M. Aldinucci et. al. Behavioural Skeletons for Component Autonomic Management on Grids. CoreGRID workshop on grid programming model, Heraklion, Greece, 2007.

[14] R. Sessions. COM and DCOM: Microsoft's Vision for Distributed Objects. John Wiley & Sons, 1997, ISBN 978-0471193814.

[15] O. F. Rana et. al. Implementing Problem Solving Environments for Computational Science. Proc. of EuroPar Conference, pp. 1345-1349, 2000.

[16] J. Cohen et. al. RealityGrid: An Integrated Approach to Middleware through ICENI. Physical and Engineering Sciences, Volume 363, Issue 1833, pp. 1817-1827, 2005, Royal Society.

[17] S. See et.al. GriDE: A Grid-Enabled Development Environment. LNCS, Vol. 3032, pp. 495-502, Springer, 2003.

[18] Sun Grid Plugin for Net Beans, https://sungridplugin.dev.java.net/, 2007.

[19] M. Aldinucci et. al. ASSIST as a Research Framework for High-performance Grid Programming Environments, Springer, 2005, ISBN 1852339985.