Project no. FP6-034442

GridCOMP

Grid programming with COMPonents : an advanced component platform for an effective invisible grid

STREP Project

Advanced Grid Technologies, Systems and Services

D.GIDE.03 – Grid IDE Prototype and Early Documentation

Due date of deliverable: 31 May 2008

Actual submission date: 16 June 2008

Start date of project: 1 June 2006 **Duration**: 30 months

Organisation name of lead contractor for this deliverable: UoW

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	PU

Keyword List: Component-based development, Grid applications, integrated environments
Responsible Partner: Vladimir Getov, UoW

MODIFICATION CONTROL			
Version	Date	Status	Modifications made by
0	DD-MM-YYYY	Template	Patricia HO-HUNE
1	04-06-2008	Draft	Vladimir Getov
2	05-06-2008	Draft	Stavros Isaiadis
3	16-06-2008	Draft	Vladimir Getov
4	27-06-2008	Draft	Artie Basukoski
5	28-06-2008	Draft	Stavros Isaiadis
6	30-06-2008	Final	Vladimir Getov

Deliverable manager

- Vladimir Getov, UoW

List of Contributors

- Stavros Isaiadis, UoW
- Artie Basukoski, UoW
- Jeyan Thiyagalingam, UoW

List of Evaluators

- Eric Madelaine, INRIA
- Thomas Weigold, IBM

Summary

Component-oriented development is a software design method which enables users to build large scale Grid systems by integrating independent and possibly distributed software modules (components), via well defined interfaces, into higher level components. The main benefit from such an approach is improved productivity. Firstly, due to abstracting away network level functionalities, thus reducing the technical demands on the developer. Secondly, by combining components into higher level components, component libraries can be built up incrementally and made available for reuse. In this report, we share our initial experiences in designing and developing an integrated development environment for Grids to support component-oriented development, deployment, monitoring, and steering of large-scale Grid applications. The development platform, which is tightly integrated with Eclipse software framework, was designed to empower the developer with all the tools necessary to compose, deploy, monitor, and steer Grid applications. We also discuss the overall functionality, design aspects, and initial implementation issues. After that, we illustrate the methodology using as an example the development of a complex distributed business process application for a biometric identification system. Finally, we report our initial findings and experiences of applying the methodology and the integrated environment, to best exploit the GCM framework.

Table of Content

1 Introduction

Arguably the most serious obstacle to the acceptance of modern distributed and Grid systems is the so-called *software crisis*. Software, in general, is considered the most complex artefact in distributed computing; since the lifespan of Grid infrastructures has been so brief, their software environments rarely reach maturity making the software crisis especially acute. Hence, shorter development cycle, portability and support for dynamic properties, in particular, are critical issues in enabling large Grid computing systems. This, however, makes the search for both the most appropriate programming abstractions [6] and efficient problem solving environments [12] even more important than before.

Therefore, it is generally accepted that component-based software development is becoming the most cost-effective approach to application construction for complex distributed and Grid systems. The wide adoption of component-based software development and in particular the use of suitable programming models for compatibility and interoperability are key issues towards building effective future Grids. Examples of component models applicable to this field include the Common Component Architecture (CCA) [2], the CORBA Component Model (CCM) [10], and the emerging Grid Component Model (GCM) [7].

The Fractal specification [5] proposes a generic, typed component model in which components are runtime entities that communicate exclusively through interfaces. One of the crucial features of this model is its support for hierarchical composition. Another key feature is its support for extensible reflective facilities: each component is associated with an extensible set of controllers that enable inspecting and modifying internal features of the component. Controllers provide a means for capturing extra-functional behaviours such as varying the sub-components of a composite component dynamically or intercepting incoming and outgoing operation invocations. The GCM proposal is an extension of the Fractal component model that specifically targets Grid environments.

In CCA, components interact using ports, which are interfaces pointing to method invocations. Components in this model define *provides-ports* to provide interfaces and *uses-ports* to make use of non-local interfaces. The enclosing framework provides support services such as connection pooling, reference allocation and other relevant services. Dynamic construction and destruction of component instances is also supported along with local and non-local binding. Though CCA enables seamless runtime interoperability between components, one of the main weaknesses of the CCA is the lack of support for hierarchical component composition and for control mechanisms thereof.

The CCM [10] is a language-independent, server-side component model which defines features and services to enable application developers to build, deploy and manage components to integrate with other CORBA services. The CCM specification introduces the concept of components and the definition of a comprehensive set of interfaces and techniques for specifying implementation, packaging, and deployment of components. The CCM provides the capabilities for composing components (through receptacles) and permits configuration through attributes. However, in contrast to the Fractal component model, the CCM does not permit hierarchical composition; that is, recursively composing components to form more complex, composite components.

2 An Overview of Requirements

The Grid integrated development environment (GIDE) is aimed at supporting a number of different user groups. We can classify the user groups as follows:

2.1 Application Developers

Application developers require support for developing Grid applications through graphical composition as well as having to support source-code based development. This approach aligns with industrial efforts in building applications through graphical composition [9]. However, providing support for developing Grid applications poses additional requirements, including support for Grid component models and composite components, and the complexities of deploying these components over distributed systems. Additional tools are necessary to enable deployment, monitoring of both component status and resource, and steering of components to maximise resource utilisation.

2.2 Application Users

The GIDE should facilitate the deployment of applications and subsequently, the monitoring of deployed applications. The monitoring process provides a set of opportunities for concerned users to monitor their running application in real-time. This functionality is also shared with the application developers who need such facilities to test the application during development.

2.3 Data Centre Operators

Data centres have high turnover rates. Hence there is a need for a design that would facilitate fast handovers and enable other operators to assist newcomers in coming to terms with the applications quickly. In order to achieve this we intend to deliver a standalone application as a Rich Client Platform (RCP) application, which provides the key functionalities that would be required by a data centre. These features are arranged within the deployment, monitoring, and steering perspectives. Also, personalisation of views should be limited as far as possible, so that a uniform design is visible to all operators in order to enhance handover and communication.

2.4 Key Requirements

The following is an overview of the key requirements considered for each user group during the design of the GIDE.
1. Provide a Grid IDE for programmers and composers. The main goal is to produce an integrated programming and composing GUI. It should provide the developer with graphical tools to develop both normal code and legacy code into primitive components, as well as tools for assembling existing Grid components into larger composite components. Additional support tools should also be provided, such as tools to search for suitable components, and tools to finalise the configuration of the application before execution.
2. Provide tools for the deployment of a given Grid component configuration or application. The main goal is to develop a component launcher tool that enables the developer to simply point to a component and execute. Of course the launcher will need to associate a deployment descriptor with each launched component. In addition this tool must provide monitoring at execution. This can be achieved via a components execution monitor tool, capable of monitoring the runtime dynamics of set of components, such as location, memory, status, etc.
3. Provide a Grid IDE for data-centre operators. This simplified toolset provides an easy-to-use support for installing, monitoring and mapping necessary component code to available resources. The tool must support steering, for installing, removing, and re-installing new versions of component code. It must also provide tools for the monitoring of resources. These

include usage level of resources required for execution of component-based code, as well as external services the components might need to execute.

3 New Development Methodology

The main idea behind component-based development is the provision of general reusable software units (components) that perform a particular task. Such components can then be distributed and reused in the context of a different application. Component-based development is based around three concepts:

- The encapsulation and hiding of implementation details and configuration of a component. A component provider can make functionality available in the form of a black box component that consumers can use in various applications. This high level of abstraction makes component-based development easier and less error-prone.
- Support for component composition enables the design and deployment of very complex systems that consist of smaller hierarchically organized sub-systems. This provides the foundations for reconfiguration of a particular sub-system without compromising the whole application.
- Description of the architecture of a system, using the Architecture Description Language (ADL), as well as the deployment details of where each component will be deployed and how it will interact with other (potentially remote) components.

There are generally two types of components: primitive components, which perform a simple task (typically as simple as possible so that they can be reused easily), and composite components, which consist of a number of internal sub-components. Typically composite components do not require a concrete implementation but serve mainly as the context for the coordinated usage of their internal components.

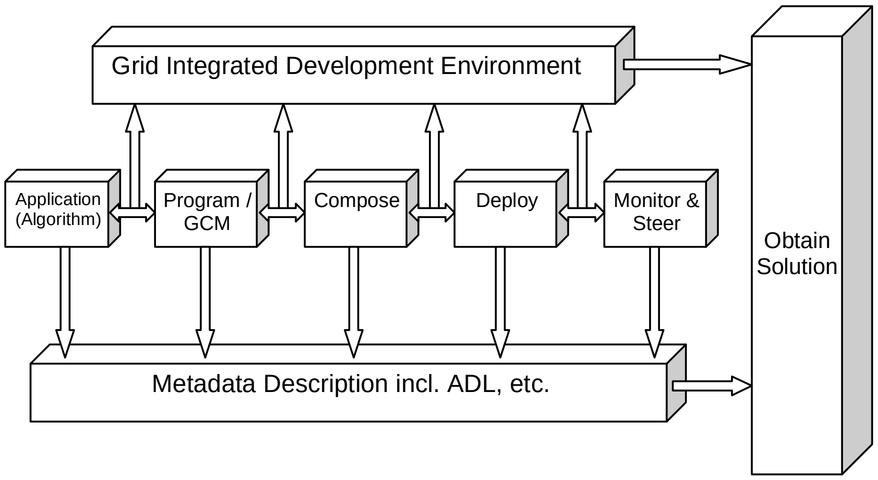


Figure 1: Component-Based Program Development Pipeline

Our vision for the GIDE design is to provide the developer with a single environment for the development pipeline. As can be seen from Figure 1 this includes graphical composition, deployment, monitoring and steering of Grid-based applications. Our methodology can be summarised by the interaction and flow of the elements depicted in Figure 1. The development process builds on the component framework described in the GCM and is dependent on the tools provided by the GIDE, which provides a set of generic components that can be dragged and dropped on a composition window. These components can be edited and composed in a hierarchical fashion to express the functionality of the required application. Facilities are also provided for editing primitive components via the direct

implementation of the required algorithms using the Java language, or legacy component wrapping.
At any point during the composition, the user has the option to save the current composition or any of its constituent primitive components as a reusable component and store it in the repository. All saved components are easily accessible in the composition perspective through a component repository view. Composition then proceeds in a hierarchical manner, allowing the composer to develop top-down or bottom up. Significant support for automatic source code generation is also available through a series of context sensitive options.
When the developer has finished the application composition, or simply wishes to test the current composition, the GIDE provides facilities to export the composition into an ADL and then deploy it according to the deployment descriptors the developer has provided. After deployment the developer can monitor the deployed components through the monitoring perspective of the GIDE. Monitoring properties include the performance of the component, the resource utilization, and the state of the components. Through the monitoring and steering perspective, the user has the option to fine tune some properties and dynamically reconfigure the application (taking into consideration the monitoring data that the GIDE provides), such as add or remove worker components, start, stop, and relocate components from one virtual node (VN) to another. In addition, autonomic component management is being implemented through the use of behavioural skeletons [1]. Behavioural skeletons add reconfiguration actions to components which can be triggered by the environment. Additionally, dynamic reconfiguration of component properties or relationships is enacted through rules for typical reconfiguration strategies. The component framework implementation (CFI) we use is based on ProActive [11] and ensures that the program behaviour remains consistent during any such relocation.
The methodology we propose shortens the development life cycle through the sharing and reuse of components from the repository. ADL's may be exported and imported, and can be deployed on any platform implementing the GCM, thus making ADL's the main functional unit for any application, and hence making applications under this methodology highly portable within the framework.

4 Grid Integrated Development Environment

Developing distributed applications on a grid infrastructure poses significant challenges. In addition to mastering a suitable programming language, the developer must also have a working knowledge of the middleware and occasionally lower level platform dependent features. Such a broad range of expertise adds significantly to the cost of a project in terms of development time, hence time to delivery. The aim of the GIDE (Figure 2) is to abstract the middleware and platform dependent features as much as possible. In doing so we reduce the learning curve as well as the development and debugging effort, which can often be prohibitive when developing in distributed and grid environments. The GIDE was designed with two different user groups in mind: application developers and data centre operators. For the application developers we provide support for developing through graphical composition, but ensure that the developer always has access to the middleware functionalities and source-code based development if required. We recognise that access to lower level functionalities is sometimes necessary for debugging and improving program efficiency. It also helps to avoid the cases where applications become bulky and inefficient if development is forced to adhere to using only pre-built components. Additional tools are also necessary to enable easy deployment and monitoring of both component status and resource usage to facilitate the development process.

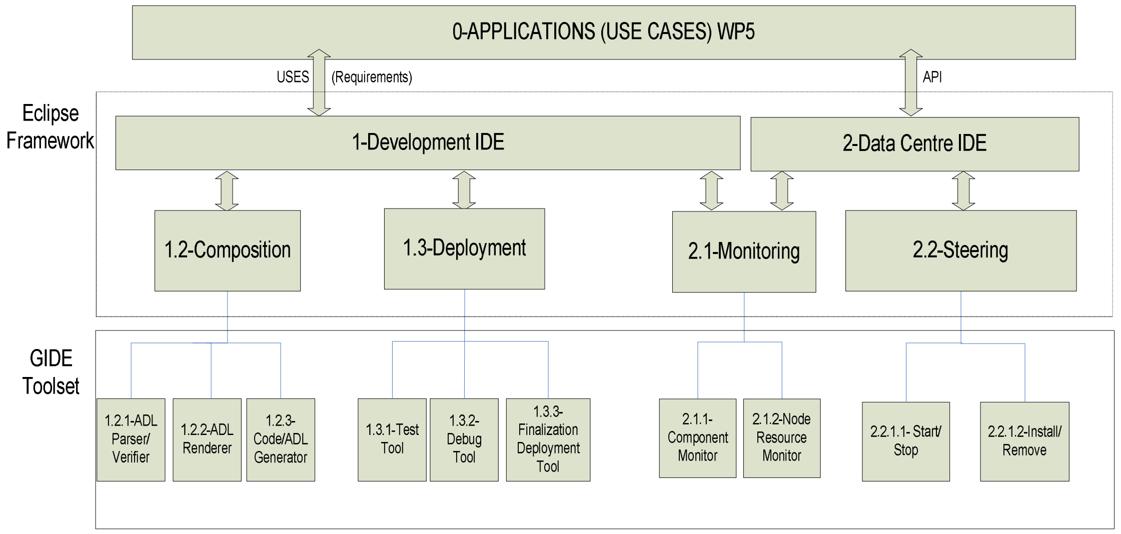
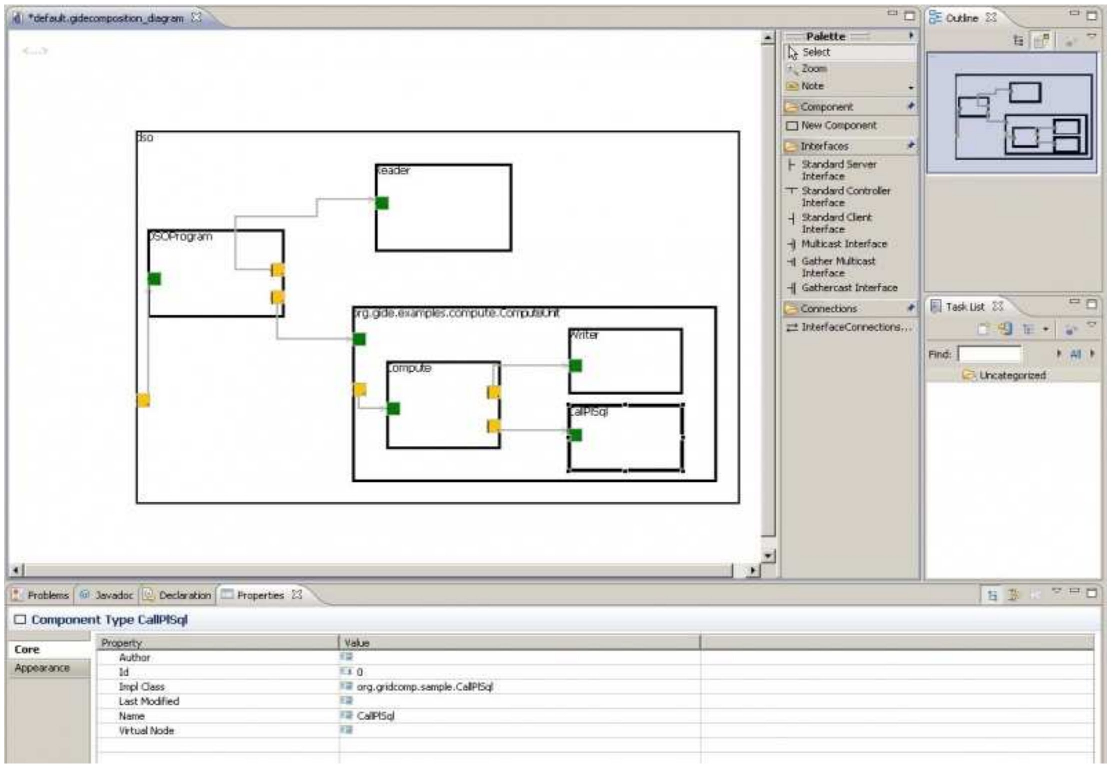


Figure 2: GIDE Block Diagram

Such monitoring functionalities are also applicable to a data centre environment. By repackaging the GIDE functionality, a simplified tool for installing, monitoring and mapping necessary component code to available resources can be provided for a Data Centre environment. However, the framework must provide additional support for steering to enable the data centre operators to install, remove, and upgrade to new versions of component code if required. Data centres have high turnover rates. Hence there is a need for a simple design that would facilitate fast handovers, and enable operators to assist newcomers in coming to terms with the application quickly.
The GIDE is built on the Eclipse [9] framework leveraging the facilities provided by the Eclipse Modelling Framework (EMF) and the Graphical Modelling Framework (GMF). Using Eclipse guards from obsolescence and enables seamless customisation and extension through its plug-in architecture. It is a leading Java development environment, and allows easy integration with many libraries, including the ProActive [11] middleware libraries being used for the CFI. In addition, it provides access to countless other plug-ins available online from other developers. The development environment has been distributed as a set of plug-ins. The GIDE provides its main functionalities for composition, deployment, and monitoring as different so-called perspectives – an Eclipse specific method for grouping a set of functionalities as a graphical view. For the data centre operators, we intend to deliver a standalone application as a Rich Client Platform (RCP) application. Further details of the GIDE and its design may be found in [3].

4.1 Composition Perspective

The composition process is enabled via a fully interactive environment. The underpinning feature which enables such interactivity is the event driven approach. Eclipse acts as the host platform to our environment. The Graphical Editing Framework, GMF-Runtime, and Eclipse facilitate handling of different events within the environment. These events are captured by the host platform through a message loop processed by the Eclipse, which are then routed to the dedicated event handlers or providers of the environment. These event handlers or providers are linked to the underlying model so that the changes are reflected upon editing.



Component Type CallPlSql

Property	Value
Author	
Id	0
Impl Class	org.gridcomp.sample.CallPlSql
Last Modified	
Name	CallPlSql
Virtual Node	

Figure 3: GIDE Composition Perspective

As an example, let us consider the composition process using Atos' use case from the GridCOMP project. The central area of the composition perspective (see Figure 3) focuses the user on the graphical composition view which provides the developer with a palette of available components that can be dragged and dropped on the composition canvas. Components can also be imported from existing Architecture Description Language (ADL) files and stored in the component palette. ADL files conform to the GCM-specification for describing compositions such as in [7]. Components can then be resized and moved, modified and stored. Connections between the interfaces can be drawn directly between the components using the connection tool. Composition is achieved by drawing a membrane around a group of components and defining interfaces. The developer is able to switch between the graphical view and a view of the fractal description of the component as an ADL file. The ADL file can then be exported and used for deployment.

4.2 Deployment Perspective

This perspective consists of views needed for application deployment. Currently deployment is provided via the IC2D plug-in. Planned enhancements are to provide a view of a deployment descriptor editor to map virtual nodes to physical hosts, and then to associate components with virtual nodes. Virtual nodes are included in ADL files to specify the number of virtual nodes that the component will need. A developer may have a set of these deployment descriptors to be used for deployment to 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.

4.3 Monitoring Perspective

The monitoring perspective provides the views that data centre operators need in order to properly monitor the runtime environment (see Figure 5). The monitoring perspective consists of component and resource monitor views. Three types of monitoring are necessary in order to enable proper management of applications. Firstly, monitoring of resources provides the hardware status of hosts. This includes CPU utilization, hard disk space, and other platform specific status information. Secondly, monitoring of the GCM components themselves provides status and location information along with a zoom-in feature for monitoring sub-components. Finally, we allow monitoring of active objects, which is necessary for developers/composers to debug and monitor applications during the development phase.

4.4 Steering Perspective

More useful for data centre operators, the aim of the steering perspective is to provide views to enable the operator to start, stop, and relocate components. Building on the monitoring and host views, it has as its main focus a component monitoring view. This view graphically shows the components location and their status. An additional view shows the geography and resource availability of the hosts, virtual nodes, as well as the components that are running on them. Based on these views, the operator has the facility to start, stop or move components from one virtual node to another while monitoring their status to ensure correct execution.

5 Case Study of Development Process

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 to 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 framework [8] and the GIDE [3] 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.

5.1 Component Programming and Composition

The BIS application can be considered a business-process or workflow-driven code. This means it is built around a workflow execution engine acting as the central control unit of the system. The workflow engine used is the embedded process virtual machine (ePVM) [14], which executes workflows defined as JavaScript scripts. These scripts define processes for identity enrolment, identification, and system management. The workflows can communicate with entities outside of the ePVM engine via so-called workflow adapters. Among them is the GCM adapter, which allows the workflow engine to access the Grid infrastructure via GCM components.

In this sub-section we describe the development of the Grid component architecture that enables the GCM adapter to provide distributed biometric matching functionality via the Grid [13]. The basic approach is to have one component encapsulating the biometric matching functionality, which is then deployed on all Grid nodes in a SPMD-style setting. Then the database of enrolled (known) identities is distributed across the nodes and this way the 1:N matching operation is executed in parallel.

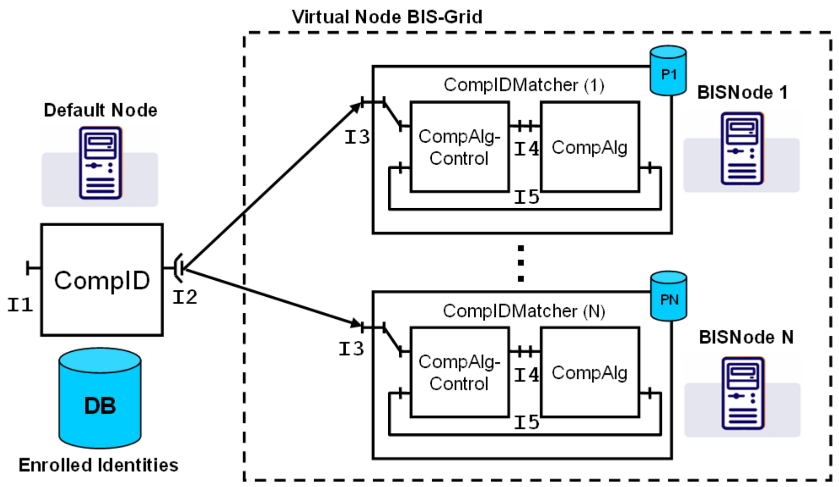


Figure 4: Overall BIS Grid component design

Figure 4 shows the overall component design, the bindings between components, and their deployment to the physical grid infrastructure. The *CompID* component runs on the default node, has access to the database of enrolled identities, and is connected to all distributed components via its multicast interface *I2* [4]. The multicast interface is used to broadcast identification requests within the Grid. The *CompIDMatcher* composite component is deployed to all nodes and represents the biometric matching unit that works on a part of the database depicted as *PN*. It encapsulates the *CompAlg* component representing the actual fingerprint matching algorithm and the *CompAlgControl* component managing the state of the local matching process.

```
<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: CompIDMatcher ADL example

To turn this overall high-level design into reality we use the graphical composition perspective of the GIDE to define the components, their names, and their interface names. We start with the primitive components *CompID*, *CompAlg*, and *CompAlgControl*. Then we define the composite component *CompIDMatcher* from the two primitives (see Figure 6) and we wire their interfaces (*I3*-*I5*) to designate component bindings as outlined in Figure 4. All this is done solely graphically via simple drag and drop operations. Lastly, the GIDE

generates the corresponding GCM ADL description files automatically. Listing 1 exemplifies the resulting ADL for the *CompIDMatcher* component. The GIDE also supports manual editing of ADL files as well as the import of ADL files possibly created by other means. As the current version of the ADL does not support binding an arbitrary number of *CompIDMatcher* components to the interface *I2*, the component allocation and binding has been done programmatically.
In a next development step the Java interfaces *I1-I5* need to be defined and the functional code of the primitive components must be implemented. The GIDE automatically generates Java skeleton code that the user can easily fill in. And since the GIDE is an Eclipse plug-in we can simply switch to the standard Java perspective to edit and compile the required Java classes. The composite component does not need any implementation besides its interface definition. It is automatically generated by the GCM framework at the time the ADL files are deployed. With the help of the GIDE all these development artefacts can be generated and developed, graphically or manually, in one consistent environment by switching between the different Eclipse perspectives.

5.2 Deployment and Monitoring

In this sub-section we describe the tasks required to deploy the BIS component system to a particular physical Grid infrastructure and to verify the deployment. As outlined in Figure 4, it is assumed that a virtual node [11] named *BIS-Grid* comprised of an arbitrary number of nodes named *BISNode 1-N* exists. The virtual-node tag as shown in Listing 1 defines that the GCM framework should allocated one instance of the *CompIDMatcher* component on each node within the *BIS-Grid* virtual node. To define the mapping from the design infrastructure to the physical infrastructure an appropriate GCM deployment descriptor must be created. The GCM adapter passes this descriptor to the GCM framework together with the ADL files defining the components to be allocated. The deployment perspective of the GIDE includes a deployment descriptor editor to support this development step. It also allows working with a set of deployment descriptors for different hardware configurations. Listing 2 shows a snippet of a BIS deployment descriptor designated for local testing. This means a number of *BISNodes*, here only two, are defined within the virtual node and both are simply mapped to different JVMs allocated on the local machine.
The current GridCOMP ADL is rather static such that parameterized architectures, for instance, binding an interface to unknown in advance number of nodes, can not be expressed in the ADL. However, such scenarios can still be implemented via the API by creating components and bindings programmatically. This has been done in the presented case study. Alternatively, results from WP3, namely, behavioural skeletons can be used. With skeletons, the problem is solved via specific dynamic component controllers (so-called autonomic behaviour controllers) which can be optionally accompanied with autonomic managers triggering autonomic operations. This approach is currently implemented in a second version of the BIS case study.

```
…
<componentDefinition>
  <virtualNodesDefinition>
    <virtualNode name="BIS-Grid" property="multiple" />
  </virtualNodesDefinition>
</componentDefinition>

<deployment>
  <mapping>
    <map virtualNode="BIS-Grid" />
      <jvmSet>
        <vmName value="BISNode1" />
        <vmName value="BISNode2" />
      </jvmSet>
    </map>
  </mapping>

  <jvms>
    <jvm name="BISNode1">
      <creation><processReference refId="localJVM" /></creation>
    </jvm>
    <jvm name="BISNode2">
      <creation><processReference refId="localJVM" /></creation>
    </jvm>
  </jvms>
</deployment>

<infrastructure>
  <processes>
    <processDefinition id="localJVM">
      <jvmProcess
        class="org.objectweb.proactive.core.process.JVMNodeProcess">
      </jvmProcess>
    </processDefinition>
  </processes>
</infrastructure>
…
```

Listing 2: BIS-Grid deployment descriptor snippet

Once a deployment descriptor is available all development artefacts required to instantiate the component system are complete. Here we do not further consider the non GCM component related part of the BIS, which connects to the component system via interface I1 (c.f. Figure 2). However, since it is purely Java based this part can be conveniently developed within Eclipse, too.
Finally, we can run the application and verify the component architecture by switching to the monitoring perspective of the GIDE using the ProActive monitoring tool, named IC2D, which has already been integrated into the GIDE. IC2D does not visualize GCM components but the underlying Active Objects [6] of the ProActive middleware. Due to the fact that in our component system each component is mapped to exactly one Active Object, IC2D clearly reflects our component architecture as can be seen in Figure 5. In this run five *BISNodes* have been allocated, each hosting two primitive and one composite component. The *CompID* component is allocated on the default node named "Node Node".

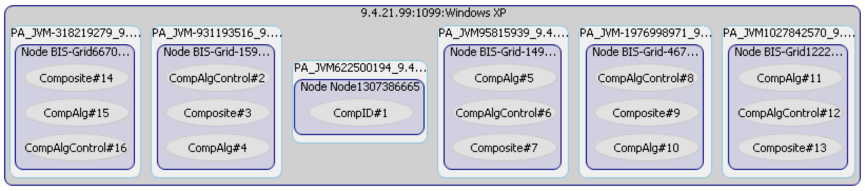
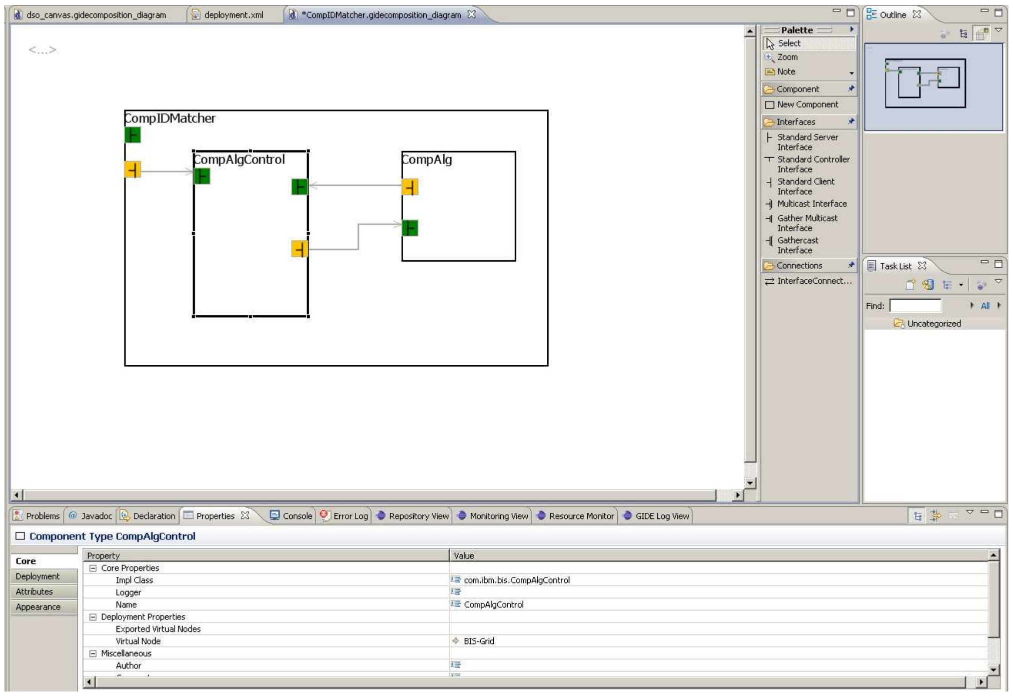


Figure 5: Active Objects rendered by the IC2D monitoring tool

5.3 Initial Experiences

The GCM framework offers a very high level of abstraction hiding the complexities of Grid programming. In combination with the comprehensive tool box of the GIDE supporting these abstraction mechanisms it enables non-Grid experts to leverage the potential of distributed resources. Also, the development methodology presented fits well into the trend towards model driven development strategies involving visual software composition and automatic code generation.
Current case studies where the GIDE is being used to enhance the development process include IBM's BIS system (see Figure 6) as reported here, as well as Atos' use case (see

Figure 3). Reports on the initial findings are encouraging with partners reporting a significant speedup in the generation of ADL's.
A further distinct strength of the GCM framework is the strict separation of concerns. The BIS case study revealed that 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 is not extensive. The GIDE significantly simplifies creating and maintaining these artefacts and thus speeds up the development cycle. Yet, the initial GIDE prototype does not support advanced code refactoring. This would further enhance convenience, for instance, when renaming an interface, because interface names appear in Java files as well as in ADL files.



Component Type CompAlgControl

Property	Value
Core Properties	
Impl Class	com.ibm.bis.CompAlgControl
Logger	
Name	CompAlgControl
Deployment Properties	
Exported Virtual Nodes	
Virtual Node	BIS-Grid
Miscellaneous	
Author	

Figure 6: BIS Example composed within GIDE

The fact that the GIDE has been developed as a pluggable module for Eclipse results in an increased feature set and consistent look-and-feel for developers. Also, the GIDE varies from existing frameworks by providing explicit support for all Grid specific needs and functionalities of the underlying component framework. Consequently, the complete development cycle can be carried out within one consistent environment. For instance, the integrated monitoring perspective has been very helpful during the case study to easily verify correct component distribution or co-allocation, respectively. Unfortunately, the final component monitoring perspective was not available and the integrated IC2D perspective was used as a workaround. Nevertheless, even the initial prototypes of the GCM framework implementation and the GIDE already show the potential of the new development methodology for next generation component-based Grid applications.

6 Conclusions and Future Work

We have presented a methodology for component based development within Grid environments. Our approach builds on the support tools being developed within the framework of the GCM, and was demonstrated through the implementation of a Biometric Identification System case study. The benefits of using such an approach have been highlighted. These are:

- shorter development cycle,
- higher portability, and
- support for dynamic properties.

These benefits have been realised through the use of the development tools provided within the GIDE for graphical and hierarchical component based development, as well as support for deployment, monitoring and steering. Having the GIDE tightly integrated to the CFI, ensures that the correctness properties are inherited by our environment.

While the GIDE is under further development, it is available as a prototype. As of this release, there are still a few features that need to be implemented. Component monitoring is still in progress, with basic monitoring at the Active Object level provided via the IC2D plugin. Deployment is also provided via the IC2D launcher but is planned to be extended when component monitoring is available. Once these functionalities are enabled, we can focus on implementing the steering features. Future work will focus on extending the dynamic monitoring of components in a graphical window with the ability to zoom into/out of server farms and geographic regions. The other main area of focus will be on the use of the tools and methodology presented for the implementation of further case studies for the use cases undertaken as part of the GridCOMP project.

References

1. M. Aldinucci et al. Behavioural skeletons in GCM: autonomic management of Grid components. Proc. of PDP Conference, 2008.
2. R. Armstrong et al. Toward a Common Component Architecture for High-Performance Scientific Computing. Proc. of HPDC Conference, 1999.
3. A. Basukoski, V. Getov, J. Thiyagalingam, and S. Isaiadis. Component-oriented Development Environment for Grid: Design and Implementation, in Making Grids Work, Springer, 2008 (to appear).
4. F. Baude, D. Caromel, L. Henrio, M. Morel. Collective Interfaces for Distributed Components. Proc. of CCGrid Conference, 2007.
5. E. Bruneton, T. Coupaye, J. B. Stefani. The Fractal Component Model. Technical report, ObjectWeb Consortium, February 2004, http://fractal.objectweb.org/.specification/index.html.
6. D. Caromel, L. Henrio. A Theory of Distributed Objects. Springer, 2005.
7. CoreGrid NoE – Institute on Programming Model, Basic Features of the Grid Component Model, *Deliverable Report D.PM.04*, 2007.
8. GridCOMP – Effective Components for the Grid, 2006, http://gridcomp.ercim.org/.
9. Eclipse – An Open Development Platform, http://www.eclipse.org/.
10. Object Management Group (OMG). The CORBA Component Model. Revision V4.0, 2006.
11. The ObjectWeb consortium. ProActive – Programming, Composing, Deploying on the Grid. http://www-sop.inria.fr/.
12. O.F. Rana, M. Li, M.S. Shields, D.W. Walker, and D. Golby. Implementing Problem Solving Environments for Computational Science. Proc. EuroPar Conference, pp. 1345-1349, 2000.
13. T. Weigold, P. Buhler, J. Thiyagalingam, A. Basukoski, V. Getov. Advanced Grid Programming with Components: A Biometric Identification Case Study, Proc. IEEE COMPSAC, IEEE CS Press, 2008 (to appear).

14. T. Weigold, T. Kramp, P. Buhler. ePVM – An Embeddable Process Virtual Machine. Proc. of IEEE COMPSAC, IEEE CS Press, 2007.

Appendix A: GIDE: Documentation

Latest version can be found in:
http://perun.hscs.wmin.ac.uk/dis/gide/wiki/index.php/GIDE:Documentation

This version release date: 30/05/2008

1 Setting Up

1.1 Dependency Setup

Tested with JDK 6. Some features of the GIDE wizards in fact require Java 6. So: download and install Java 6!
Download Eclipse from http://www.eclipse.org/downloads/. We have tested with Eclipse for Java Developers
(http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/europa/winter/eclipse-java-europa-winter-win32.zip) which is also part of the Europa Edition (http://www.eclipse.org/europa/). The instructions that follow apply to this version but should be very similar with all recent versions.

Unzip Eclipse to a folder of your choice (e.g. C:\eclipse)

1. Update Eclipse with the Update Manager using the following instructions:

- Select **Help -> Software Updates -> Find and Install...**
- Select "**Search for new features to install**" and press next

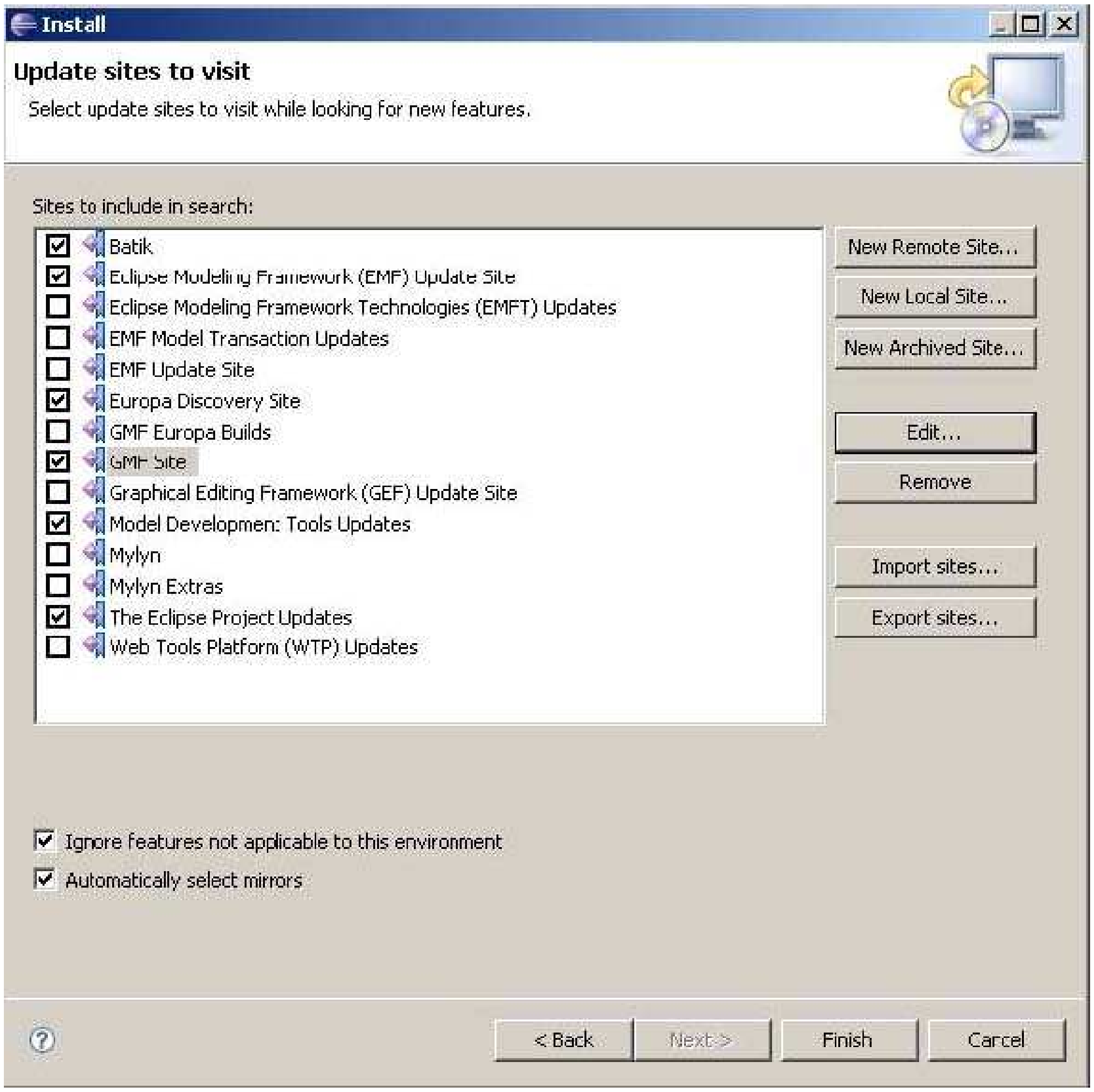
  - Select "New Remote site" and enter name: GMF and URL: http://download.eclipse.org/modeling/gmf/update-site/releases/site.xml (alternative URL in case you run into problems: http://download.eclipse.org/modeling/gmf/update-site/europa/site.xml)
  - Select "New Remote Site" and enter name: Batik and URL: http://download.eclipse.org/modeling/gmf/update-site/batik/site.xml
  - if not already in the list add "New Remote Site" for name: Eclipse Modeling Framework (EMF) Updates and URL: http://download.eclipse.org/modeling/emf/updates/
  - if not already in the list add "New Remote Site" for name: Model Development Tools (MDT) Updates and URL: http://download.eclipse.org/modeling/mdt/updates/
  - if not already in the list add "New Remote Site" for name: The Eclipse Project Updates and URL: http://update.eclipse.org/updates/3.3

- Check the boxes for the GMF and Batik sites as well as for Eclipse Modeling Framework (EMF) Updates, Model Development Tools (MDT) Updates and The Eclipse Project Updates.
- Press Finish. In the resulting dialog uncheck "**Show the latest version of a feature only**"
- Click on the GMF node to expand and check the latest version of GMF (should be the last one at the bottom)

- Press "**Select Required**" a few times until all dependencies have been resolved. For this to work you need to expand each node and then press "Select Required", e.g. expand the Eclipse Modeling Framework (EMF) Updates node and press "**Select Required**". Repeat for all other 3-4 root nodes. If this still does not work you can manually select the following nodes for updating: see Figure

Note: if you are having problems getting a reference to Batik (i.e. if the Batik node is empty) then start at step 1 again and enter and check one extra "New Remote Site" for Name: "Europa Discovery Site" and URL=http://download.eclipse.org/releases/europa. This seems to be some sort of a bug because we will not use the Europa Discovery Site for the Update process at all! However it seems to "activate" the download site for Batik somehow. This might be completely coincidental but it is how it worked in both Windows and Linux in our installations!

- Select Next
- Accept the license agreement and press Next
- In the next page select Finish to install the plugins to the selected folder (e.g. C:\eclipse)



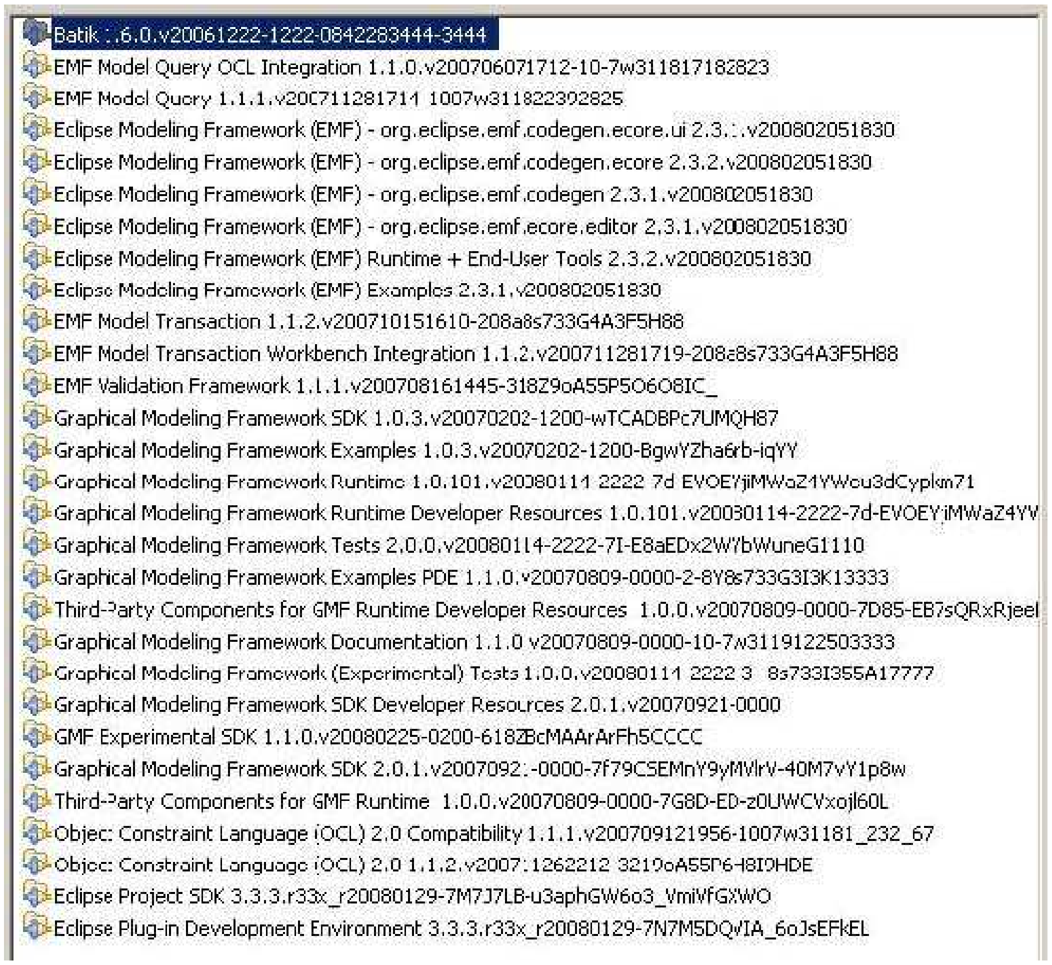
Update sites to visit

Select update sites to visit while looking for new features.

Sites to include in search:

- [✓] Batik
- [✓] Eclipse Modeling Framework (EMF) Update Site
- [ ] Eclipse Modeling Framework Technologies (EMFT) Updates
- [ ] EMF Model Transaction Updates
- [ ] EMF Update Site
- [✓] Europa Discovery Site
- [ ] GMF Europa Builds
- [✓] GMF Site
- [ ] Graphical Editing Framework (GEF) Update Site
- [✓] Model Development Tools Updates
- [ ] Mylyn
- [ ] Mylyn Extras
- [✓] The Eclipse Project Updates
- [ ] Web Tools Platform (WTP) Updates

- [✓] Ignore features not applicable to this environment
- [✓] Automatically select mirrors



Batik 1.6.0.v20061222-1222-0842283444-3444
EMF Model Query OCL Integration 1.1.0.v200706071712-10-7w311817182823
EMF Model Query 1.1.1.v200711281714 1007w311822392825
Eclipse Modeling Framework (EMF) - org.eclipse.emf.codegen.ecore.ui 2.3.1.v200802051830
Eclipse Modeling Framework (EMF) - org.eclipse.emf.codegen.ecore 2.3.2.v200802051830
Eclipse Modeling Framework (EMF) - org.eclipse.emf.codegen 2.3.1.v200802051830
Eclipse Modeling Framework (EMF) - org.eclipse.emf.ecore.editor 2.3.1.v200802051830
Eclipse Modeling Framework (EMF) Runtime + End-User Tools 2.3.2.v200802051830
Eclipse Modeling Framework (EMF) Examples 2.3.1.v200802051830
EMF Model Transaction 1.1.2.v200710151610-208a8s733G4A3F5H88
EMF Model Transaction Workbench Integration 1.1.2.v200711281719-208a8s733G4A3F5H88
EMF Validation Framework 1.1.1.v200708161445-318Z9oA55P5O6O8IC_
Graphical Modeling Framework SDK 1.0.3.v20070202-1200-wTCADBPc7UMQH87
Graphical Modeling Framework Examples 1.0.3.v20070202-1200-BgwYZha6rb-iqYY
Graphical Modeling Framework Runtime 1.0.101.v20080114 2222 7d EVOEYjiMWaZ4YWou3dCyplm71
Graphical Modeling Framework Runtime Developer Resources 1.0.101.v20080114-2222-7d-EVOEYjiMWaZ4YV
Graphical Modeling Framework Tests 2.0.0.v20080114-2222-7I-E8aEDx2WYbWuneG1110
Graphical Modeling Framework Examples PDE 1.1.0.v20070809-0000-2-8Y8s733G3I3K13333
Third-Party Components for GMF Runtime Developer Resources 1.0.0.v20070809-0000-7D85-EB7sQRxRjeel
Graphical Modeling Framework Documentation 1.1.0 v20070809-0000-10-7w3119122503333
Graphical Modeling Framework (Experimental) Tests 1.0.0.v20080114 2222 3 8s733I355A17777
Graphical Modeling Framework SDK Developer Resources 2.0.1.v20070921-0000
GMF Experimental SDK 1.1.0.v20080225-0200-618ZBcMAArArFh5CCCC
Graphical Modeling Framework SDK 2.0.1.v20070921-0000-7f79C5EMnY9yMVlrV-40M7vY1p8w
Third-Party Components for GMF Runtime 1.0.0.v20070809-0000-7G8D-ED-z0UWCVxojl60L
Object Constraint Language (OCL) 2.0 Compatibility 1.1.1.v200709121956-1007w31181_232_67
Object Constraint Language (OCL) 2.0 1.1.2.v20071262212 3219oA55P648I9HDE
Eclipse Project SDK 3.3.3.r33x_r20080129-7M7J7LB-u3aphGW6o3_VmiVfGXWO
Eclipse Plug-in Development Environment 3.3.3.r33x_r20080129-7N7M5DQVIA_6oJsEFkEL

1.2 Installation

Download the GIDE package
Unzip it in a folder of your choice (e.g. C:\gide)
Edit eclipse.ini from the eclipse root folder and add after -vmargs in a separate line: DGIDE_HOME=C:\GIDE or whatever your GIDE root folder is

```
-showsplash
org.eclipse.platform
--launcher.XXMaxPermSize 256M
-vmargs
-DGIDE_HOME=C:\gide
-Dosgi.requiredJavaVersion=1.5
-Xms40m
-Xmx256m
```

Remove all previous GIDE plugins from the eclipse/plugins directory.
Copy the GIDE plugins in the eclipse/plugins directory eclipse.
Start Eclipse

1.3 Directory and File Structure

Inside both versions there are the following folders:

- **doc**: documentation files
- **dtd**: the default ProActive DTD file
- **examples**: example ADL and sample projects
- **libs**: libraries that are needed for GIDE to function properly
- **log**: folder that contains the log files for each session
- **plugins**: the folder that contains the GIDE Eclipse plugins
- **repository**: the root folder for the GIDE component repository
- **temp**: a temp folder used for various purposes for the GIDE

2 Usage

2.1 Starting Up

With the latest plugin version there are no explicit startup procedures as all GIDE tools and perspectives are readily available through Eclipse. Just follow the instructions under "Using GIDE Tools and Perspectives".

2.2 Using GIDE Tools and Perspectives

There are a number of available tools specific to GIDE, ranging from Composition diagrams, import and export facilities, resource monitoring and more. This is only a quick start guide for the most common functionality. For a detailed step by step guide have a look at the Tutorials pages.

- To create a new composition, you must already have a project in your workspace and then go to **File -> New -> Example -> Gidecomposition Diagram.**
- To change the appearance of the composition diagram, right click anywhere on the empty canvas (not inside any components) and select **Show Properties View -> Rulers and Grid** to set the Grid and Rulers appearance or select **Show Properties View -> Appearance** to set the font and colors.
- To show the properties of the composition items (components, interfaces, bindings) select the component you want and its attributes will be loaded in the Properties View Panel. If the latter is not currently visible or in focus, right click on the composition item of interest and select **Show Properties View**
- To import an existing ADL into a new composition go to **File -> Import -> Other -> Import ADL** and follow the Wizard instructions
- To export a composition diagram into one or more ADL files go to **File -> Export -> Other -> Export to ADL** and follow the Wizard instructions
- To view the GIDE log for the current session go to **Window -> Show View -> Other -> GIDE -> GIDE Log View**
- To view the GIDE Component Repository go to **Window -> Show View -> Other -> GIDE -> Repository View**
- To import a component from the repository you must already have a composition diagram open and in focus. Then, select the component you want from the repository view, right click and select 'Import'. Manual refresh might be required on the composition canvas in order to see the bindings correctly.

- To generate source files for primitive components, server interfaces, or attribute interfaces, go to the relevant composition item, right click and go to the GIDE context menu to see all available options

2.3 Composition

2.3.1 Importing ADL files

2.3.1.1 General

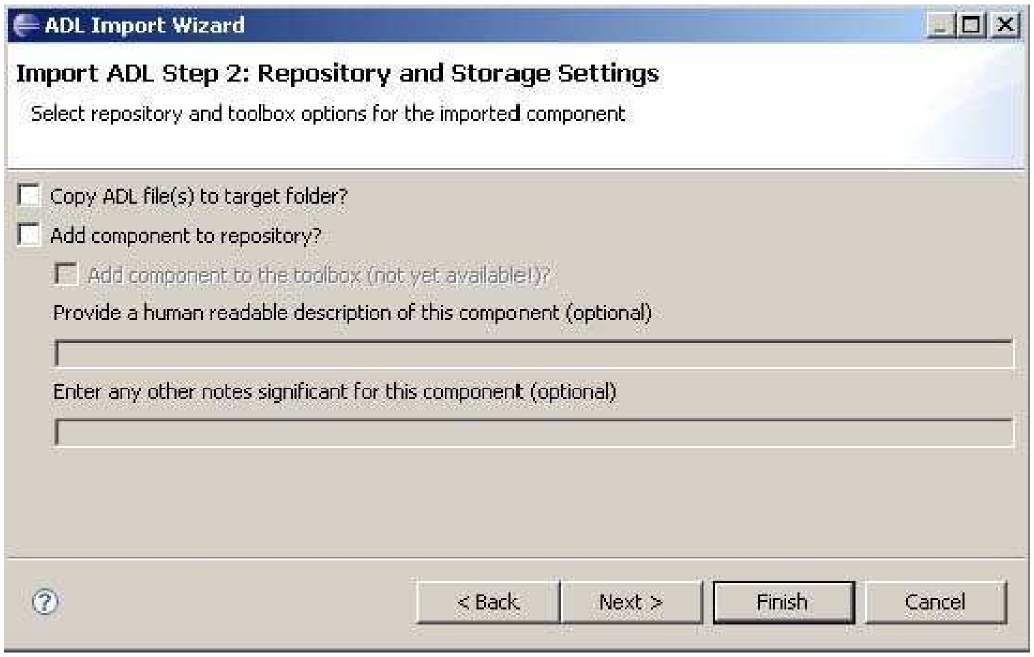
Current requirements for importing: if the top level definition name is package-qualified then the file must be located in a directory structure that reflects the specified package. Any referenced files must be either in the same directory, or in their respective directory hierarchy (if on their own they are package qualified) in either the same root directory or in another directory that is accessible through the classpath.
Any referenced DTDs must be available through http or file URLs, as absolute filenames, or as relative filenames that are available through the classpath. It is the responsibility of the developer and not the GIDE, to provide a DTD in his ADL files that is accessible. However, the GIDE will resort to a default location in the installation directory where it looks for the default DTD (for the GIDE purposes this is the ProActive DTD).
When adding to the repository only the imported component is added -not its internal components independently. This is on purpose in order to avoid cluttering and keep things simple. If the user wants to add some of the internal components separately he can do so by importing those separately.
When importing a component and an identical name already exists in the repository the user will have the option to provide an alternative name. In such case, and after any potential name conflicts have been identified and alerted the user, the repository and the imported component have the new name. However, the locally copied ADL files (if the user opted to copy locally) will be the *original* ADL files!
The option to add to the toolbox an imported component has not been implemented yet.

2.3.1.2 Wizard Options

Select File -> Import -> Other -> Import an ADL.
In the first page, use the "Browse" button or type in the location of the top level ADL file you wish to import. All dependencies will be resolved automatically.
In page 2 (see Figure) there are a number of options relating to the import:

- Copy ADL file(s) to target folder: check this to have the ADL files copied to the target import folder in a hierarchical directory structure following the ADL names, e.g. a component with name "org.gridcomp.examples.SomeComponent" will be copied to a folder \org\gridcomp\examples inside the import target folder selected.
- **Add component to repository**: check this to have the top level (and only the top level!) component of the supplied ADL file added in the GIDE repository for future usage. There will be only one component added to the repository, the top level one with all its potential subcomponents, but there will be no recursive addition of the internal subcomponents as separate repository entries. You can leave this option unchecked if you are testing some ADLs or you are currently exploring the GIDE.
- **Add component to the toolbox**: obsolete now, will be removed in future versions.
- **Descriptions**: you can optionally provide a human readable description as well as other

notes for the imported component. This option is only available if the component is to be added to the repository.


ADL Import Wizard

Import ADL Step 2: Repository and Storage Settings

Select repository and toolbox options for the imported component

☐ Copy ADL file(s) to target folder?

☐ Add component to repository?

☐ Add component to the toolbox (not yet available!)?

Provide a human readable description of this component (optional)

Enter any other notes significant for this component (optional)

2.3.2 Exporting to ADL files

2.3.2.1 General

General Naming Requirements: top level canvas MUST have a name. All components and interfaces must also have names.
Component/definition names may be package qualified in the composition. The package dictates the directory hierarchy where the resulting ADL will be placed. If a
sub component is not package qualified, it is assumed to inherit the package from its parent. If it is package qualified, it will be placed in this potentially different directory hierarchy, nevertheless in the same export root that the user selects
When exporting to an ADL, the potentially different packages of a component will only be significant when splitting the files. Otherwise it might not even be possible to determine the difference.

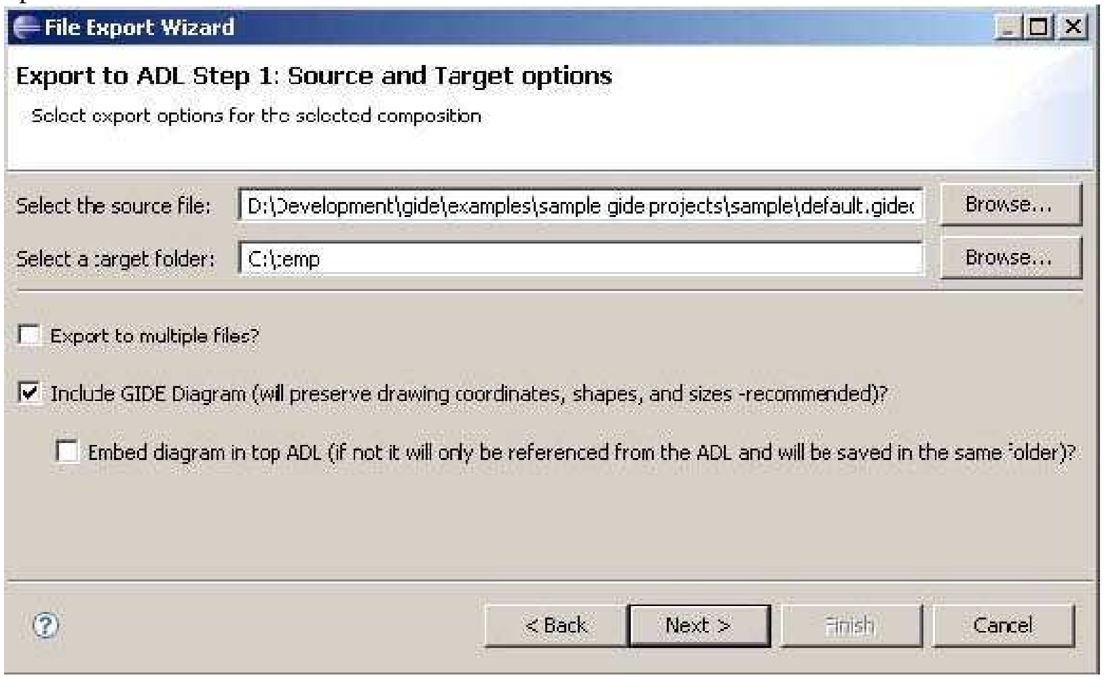
2.3.2.2 Wizard Options

Select File -> Export -> Other -> Export to ADL file(s).
Use the "Browse" button or type in the location of the source GIDE composition diagram file you wish to export.
Use the "Browse" button or type in the target location for the exported ADL file(s).

- **Export to multiple files**: check this option to export the composition to multiple ADL files, one for each (sub) component. The files will be organized in a hierarchy based on their names: package qualified names will go into the equivalent directory hierarchy inside the target folder, e.g. a component named org.gridcomp.examples.SomeComponent will go inside a folder org/gridcomp/examples/ inside the target folder selected. Components with simple (i.e. non package qualified) names, inherit their parents packages. Note: if this option is unchecked, only the top level package (if one exists) will be used to determine the directory structure.

- **Include GIDE diagram**: select this option to export the GIDE diagram file as well. This will preserve the drawing, shapes, sizes, arrangements etc. so that if the exported ADL(s) are imported again (in the same or another GIDE) the diagram will be preserved. This is highly recommended.

Embed diagram in top level ADL: if the GIDE diagram is to be included, it can be done in two ways: through a reference from the top level ADL to an external diagram file (which will have to be moved along the ADL(s)) or through embedding the whole GIDE diagram in the top level ADL in the form of a special comment recognizable from the GIDE parsers. The latter option has the benefit of not requiring a separate file to be maintained but it "litters" the top level ADL. This option is only available if the "Include GIDE diagram" option has been selected.


File Export Wizard

Export to ADL Step 1: Source and Target options

Select export options for the selected composition

Select the source file: D:\Development\gide\examples\sample gide projects\sample\default.gide

Select a target folder: C:\temp

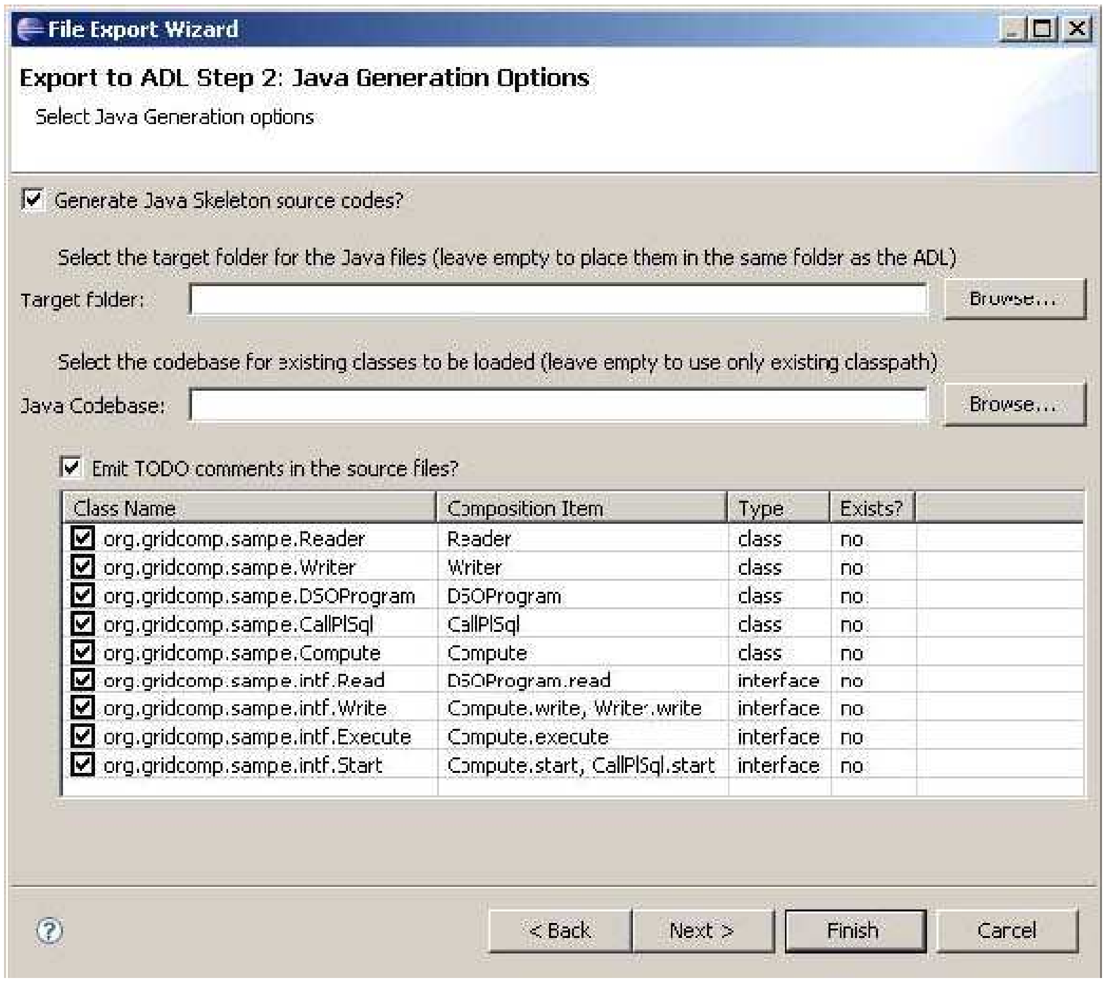
[ ] Export to multiple files?

[✓] Include GIDE Diagram (will preserve drawing coordinates, shapes, and sizes -recommended)?

[ ] Embed diagram in top ADL (if not it will only be referenced from the ADL and will be saved in the same folder)?

- **Generate Java Skeleton Code**: the user has the option to generate Java skeleton files for the specified implementation and interface classes (if any were specified in the composition). All subsequent options are only available when this option has been selected.
- **Target folder**: optionally create the Java files in a different folder than the ADL target folder specified in the previous step.
- **Java Codebase**: the GIDE will make an attempt to discover existing classes in the current classpath in order to generate skeleton methods for classes implementing already existing interfaces. However, the user can also specify an extra implementation codebase folder where existing classes not available in the current classpath can be found and loaded. If none of the specified classes can be found the Java generator will merely create the directory and file hierarchy and provide the class header lines, e.g.

```
package org.gide.composition.somePackage;
/**
* Header JavaDoc
*/
public class SomeClass implements org.gide.composition.SomeInterface {
}
```



Export to ADL Step 2: Java Generation Options

Select Java Generation options

[✓] Generate Java Skeleton source codes?

Select the target folder for the Java files (leave empty to place them in the same folder as the ADL)

Target folder:

Select the codebase for existing classes to be loaded (leave empty to use only existing classpath)

Java Codebase:

[✓] Emit TODO comments in the source files?

Class Name	Composition Item	Type	Exists?
✓ org.gridcomp.sampe.Reader	Reader	class	no
✓ org.gridcomp.sampe.Writer	Writer	class	no
✓ org.gridcomp.sampe.DSOProgram	DSOProgram	class	no
✓ org.gridcomp.sampe.CallPlSql	CallPlSql	class	no
✓ org.gridcomp.sampe.Compute	Compute	class	no
✓ org.gridcomp.sampe.intf.Read	DSOProgram.read	interface	no
✓ org.gridcomp.sampe.intf.Write	Compute.write, Writer.write	interface	no
✓ org.gridcomp.sampe.intf.Execute	Compute.execute	interface	no
✓ org.gridcomp.sampe.intf.Start	Compute.start, CallPlSql.start	interface	no

However, if one of the specified interfaces can be found (either in the classpath or the specified codebase) any implementation source files implementing this interface will be filled with sample methods, as in the following example which assumes that one of the composition components was implementing the org.xml.sax.EntityResolver

```
package org.gide.composition.somePackage;
import org.xml.sax.InputSource;
/**
* Header JavaDoc
*/
public class SomeClass implements org.xml.sax.EntityResolver {

/**
*
* string_1 String
* string_2 String
*/
public InputSource resolveEntity (String string_1, String string_2){


       /**
       * @todo Fill in this method's logic code!
       */
```

```
        return null;

      }
    }
```

- **Emit TODO comments**: check this option to have TODO Javadoc comment generated for each class and each method (if any) in the class.
- **Names Table**: this table shows which implementation class names were specified in the composition diagram and the respective composition items. It further shows which of these classes were found in the classpath or in the specified codebase. The user can select which of these to be automatically generated.