

# GDT: A Toolkit for Grid Service Development

Thomas Friese, Matthew Smith, Bernd Freisleben  
Department of Mathematics and Computer Science, University of Marburg  
Hans-Meerwein-Str., D-35032 Marburg, Germany  
{friese,matthew,freisleb}@informatik.uni-marburg.de

**Abstract:** The inherent complexity of Grid application development is an obstacle to the widespread adoption of service oriented Grid technology. In this paper, a software development environment for service oriented Grid applications integrated into the Eclipse platform is presented. Its design is based on model-driven software development principles, allowing domain experts to rapidly develop Grid applications by hiding the complexities of Grid middleware. The functionality of the three main components supporting service creation, process creation and interactive debugging are presented, and implementation issues is discussed.

## 1 Introduction

Service-oriented Grid computing [FBD<sup>+</sup>04] is a relatively young field of distributed computing which is currently lacking adequate tool support for Grid application development without having to deal with overly complex Grid middleware issues. Only if domain experts (developers of "business logic", i.e. application functionality) can more or less effortlessly integrate a new middleware into their application, will a widespread adoption of service oriented Grid technology be possible.

The development of service oriented Grid applications encompasses the creation of Grid services that form the basis of an application. These services are fine grained components which can be connected in a variety of ways to form complex Grid applications. The workflow of these applications is governed by a high level ("business") process description, resulting from a process composition step. Finally, interactive debugging during the development of components and processes is required to ensure the efficient creation of Grid applications and their workflows.

Integrated development environments (IDE) offering a wide range of interactive development functionality have been widely accepted as tools improving software development time and quality. Many software developers facing the task of Grid application development will already be familiar with using IDEs on a daily basis. Grid development tools supporting the above three development steps should be integrated into common IDEs, to support developers with experiences in different application domains to move into Grid application development.

In this paper, we present the overall design of an integrated software development environ-

ment for service oriented Grid applications and its implementation within the Eclipse platform<sup>1</sup>. This *Grid Service Development Toolkit (GDT)* is based on a model-driven approach [Obj03, MM01, Bro04] to software development and is targeted at non-Grid middleware experts, allowing them to rapidly develop Grid applications by hiding the complexities of the Grid middleware from the application domain experts. The design of three main components dealing with service creation, process creation and interactive debugging is presented. For space reasons, only the implementation of the service creation component will be described in detail, since it represents the largest hurdle for domain experts in developing a service oriented Grid application. The implementation of the other two components will only be sketched.

The paper is organized as follows. Important design decisions for the main three components of the IDE are presented in section 2. Implementation issues will be described in section 3. Section 4 presents an evaluation of the system. Section 5 discusses related work. Section 6 concludes the paper and outlines areas of future work.

## 2 GDT Design

Application development in the service oriented Grid can happen on different levels of abstraction. The Grid middleware and Grid solution providers may offer basic building blocks for the component-oriented development of larger applications, but creation of a Grid application solving a particular problem may also be the composition of available services into a complex choreography of services and resources. This component-oriented approach offers the potential for domain experts to develop large scale distributed applications even though they possess only limited knowledge about the intrinsic details of Grid middleware. In this sense, the Grid users become solution producers themselves and usage of the Grid really becomes problem solving by applying component-oriented software development. This process can be regarded as the task of identifying the processes and workflows required to align the solution components in order to allow them to process a set of input data. It often is a collaborative task between different experts, and thus collaboration support in the process development tools of a Grid IDE is required.

This approach is only feasible if the development tools offer sufficient support to hide the middleware complexity from the domain experts, allowing them to concentrate on their primary field of expertise rather than bothering with Grid middleware specific questions. A model-driven development approach [Obj03, MM01, Bro04] is a good approach to support tackling the inherent complexity of the Grid as an application environment.

While interactive development support is a desirable goal, many software development practices, such as automated testing of software and automated build environments, require tools for Grid development to not only support GUI based interactive development. Such tools must also offer the ability to use them in command line oriented environments or automated build systems without a graphical user interface.

---

<sup>1</sup>[www.eclipse.org](http://www.eclipse.org)

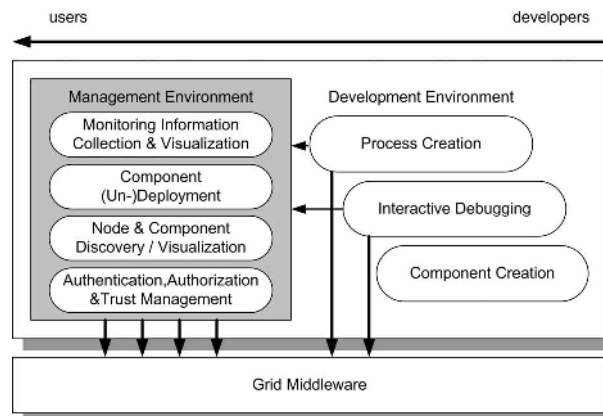


Figure 1: Components in the Grid development and management environment.

Figure 1 shows the basic components for a Grid service development environment. Arrows indicate direct access to the Grid middleware in terms of middleware control or use of the communication facilities (e.g. for collaborative editing of Grid processes). The component ordering from right to left expresses their association with the user in roles ranging from a developer to a Grid user. Note that the Grid management environment is usable on its own but also as part of the Grid development environment, offering additional functionality for interactive testing and debugging purposes (i.e. deployment of Grid services under development).

## 2.1 Model-Driven Development

Model-driven architecture (MDA) [Obj03, MM01, Bro04] has been proposed as an approach to deal with complex software systems by splitting the development process into three separate model layers and automatically transforming models from one layer into the other: The *Platform Independent Model* (PIM) layer holds a high level representation of the entire system without committing to any specific operating system, middleware or programming language. The PIM provides a formal definition of an application's functionality without burdening the user with too much detail. The *Platform Specific Model* (PSM) layer holds a representation of the software specific to a certain target platform such as J2EE, Corba or the service oriented Grid middleware. The *Code Layer* consists of the actual source code and supporting files which can be compiled into a working piece of software. In this layer, every part of the system is completely specified.

MDA theory states that a PIM is specified and automatically transformed into a PSM and then into actual code, thus making system design much easier. The trick, of course, lies in the development of generic transformers capable of generating the PSM and code layers from the PIM [Flo03, Tho04]. The reverse transformation from the code layer up to the more generic platform specific or independent layers is called *architecture driven*

*modernization* (ADM) by the OMG<sup>2</sup>. Since service oriented Grid computing is a young field, there are almost no model-driven development tools and consequently, existing Grid applications have usually not been created using model-driven development. In fact, many existing Grid applications rely not even on service oriented Grid middleware. Therefore, the ADM direction of transformations from existing code into increasingly abstract model representations is a very useful feature for solution providers and application developers.

In previous work [SFF06], we proposed a refinement of the MDA layered approach in order to provide a further separation of concerns between developers focusing on the application domain and developers focusing on Grid specific application development. In the resulting model stack, the service oriented Grid platform specific layer is further separated into an upper business layer and a lower target system specific layer. The term *business layer* refers to the "business" or application logic of a service. To facilitate the separation of the PSM layer, a UML Grid Profile is introduced to model the Grid concerns in the business layer. Three fundamental stereotypes have been defined to mark classes, operations and attributes: GridService, GridMethod and GridAttribute. These are the basic building blocks of all service oriented Grid applications and can be used to build arbitrary real world applications.

The separation of concerns between the upper layer platform specific model representing the *service oriented Grid* as the target platform, and the lower layer PSM capturing details of an implementation for a concrete target system (i.e. a Grid middleware platform such as GT4 or Unicore/GS) further supports the creation of Grid development tools with a high level of reuse. The Grid profile meta-model represents the general concepts of service oriented Grid computing introduced by the WSRF [OAS04]. The GridService corresponds to the service implementation of a WS-Resource with its methods (GridMethod) available to act on the properties collected in the resource properties document (GridAttribute). The representation of a service oriented Grid application using these upper layer modeling elements is independent of any specific implementation of the WSRF standard. Similarly, the concrete implementation of such an application can be separated into two sub-layers representing the core application logic on the upper-layer and target platform binding code corresponding to the lower layer platform specific model of the Grid applications. A model-driven development tool for the service oriented Grid should separate its internal representation of a Grid application on the platform specific level.

## 2.2 Component Creation

A first step towards support for rapid development of Grid applications is a component allowing fast creation of Grid services as the basic components of a service oriented Grid application. Figure 2 shows the relationship between different model representations and the actual platform specific source code for such an application. In a typical model-driven software development approach, software architects start by modeling an overall application in a platform independent manner often by creating an UML model of the software

---

<sup>2</sup><http://www.omg.org/>

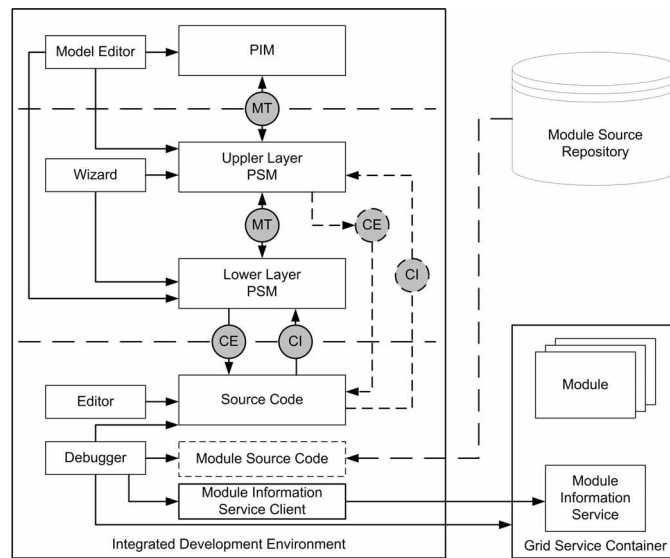


Figure 2: Model representations with transformation and debugging components in the Grid development tools.

(upper left corner). This model can relatively easily be transformed into a Grid platform specific model: by using the UML Grid profile presented in the previous section, developers can directly model the upper layer PSM of the software using stereotypes for the different components and aspects of their solution.

The step from a Grid PSM to a target platform specific model can in many cases be automated or at least be supported by development tools. Strong separation between target platform specific binding code and application logic carrying code makes the mapping from the lower layer PSM to the actual source code a straightforward mapping that can be fully automated. Many integrated development environments offer facilities for the specification and execution of such trivial mappings.

Actual source code carrying the application logic is often not fully generated from a pure model but rather attached to the structural model of the application and carried on through the different model transformations to the lower level model representations. As an alternative path from the platform independent model, an application may be annotated and used as the input to a transformation tool that generates a platform specific model for the Grid that can then be further transformed into the actual implementation of the Grid service for a concrete application, carrying the original application logic into the Grid enabled implementation. Development tools implemented using this abstract architectural model should also be capable of transforming a specific implementation upwards into more abstract representations while keeping the binding between model elements and their implementation.

Figure 2 shows the different components used to represent and transform models throughout the Grid development tools. The gray "MT" circles denote general model transfor-

mation tools. Since they are applied to model representations to be developed or based on a common meta-model language, they are likely to be implemented using a model-to-model transformation framework for that meta-model language such as ATL [JK05]. The transformation from model to source code and vice versa is modeled using distinct *Code Emitters* (CE) from model to source code and *Code Interpreters* (CI) from source code to model. Since in our model the annotated application logic carrying code is a source for upper layer model information, a direct relationship (path in the model transformations) exists between the upper layer PSM and the source code (shown by the dashed lines).

The transformation from the upper layer PSM to the target platform specific lower layer PSM can often be fully automated using common patterns for the lower layer PSM model (an implementation for the Globus Toolkit 4 as the target platform is presented in section 3; Globus was chosen since it is the de facto standard for service oriented Grid middleware). To support another target platform implementation, only this target platform specific mapping model must be replaced. All the common infrastructural components enabling the transformation, all components bridging the gap to higher level model representations and tools can be reused, using the reverse mapping from actual implementations to model representations. The availability of such target specific mapping models allows easy porting of Grid component implementations to other service oriented target platforms. In this case, a developer starts from an application for one target platform, transforms this implementation back into its upper layer representation keeping the application specific logic, and re-generates all target platform specific binding code for another target platform.

A component creation module of the GDT should support two basic use cases: Interactive development of the service reflecting changes on one level in the other model layers by transformation, and automated use as a build tool acting on previously defined service sources.

### 2.3 Process Creation

The Business Process Execution Language for Web Services (BPEL) [Oas05] has gained much attention and broad adoption for the composition of component based business applications. The focus of the BPEL language is to enable the composition of basic web services into more complex processes. Its popularity in the business application domain makes BPEL very promising and interesting for process creation in the Grid domain, since many process execution, management and creation tools are expected to be developed in the future or are even currently under development. Another benefit of using the BPEL language is the ability to seamlessly integrate resulting Grid processes with other business processes in a corporate environment.

Two main considerations should drive the design of a Grid process editor based on BPEL supporting spontaneous Grid application development: It should provide the ability to *adapt* to the needs of different groups of developers, allowing Grid middleware experts to inspect and manipulate fine details of a Grid process (high-fidelity editing) while hiding complicated details from application domain experts (low-fidelity editing). A Grid process

editor should foster *collaboration* among experts in a distributed environment. Ideally, it should support collaborative work on the process under development, building on the underlying communication infrastructure.

Since composition of the basic Grid services offered in the Grid middleware is the main focus of this GDT component, it should be more tightly integrated with certain management components such as the service discovery component.

For the overall design of such a process editor, we use the model-view-controller pattern, allowing the implementation of different view and controller components on a common model representation of the process. Thereby, only a single implementation of all internal components acting on the model (e.g. for export of the model to a concrete process execution engine) is required while views on the model can be provided for the different levels of detail. To fill the gap between high- and low-fidelity editing, a collection of wizards assigns values to the hidden properties in the model elements, based on certain patterns and heuristics defined for the overall system. Mapping heuristics represent best practices that should be defined either explicitly or implicitly by Grid middleware experts and are also shared in the collaborative Grid environment. Selection of a certain best practice wizard for an assignment may either be automatic based on an editing policy set for the application expert or based on a selection of choices made by the expert.

## 2.4 Interactive Debugging

The ability to discover errors in component implementations through interactive debugging greatly improves successful identification and correction of such implementation problems. An interactive debugger is usually part of most modern integrated development environments. For stand-alone applications, most IDEs allow the user to start an application under control of the debugger. The situation for large scale distributed applications is somewhat complicated by the distribution over different nodes in the network. Most integrated debuggers in modern IDEs nevertheless offer the ability to attach even to remote processes and enable developers to control the process execution and inspect in-memory value and execution states of the processes.

Many errors during the development of service oriented Grid applications are related to the platform binding code or happen during invocation of the service within the middleware layer before the actual invocation reaches the user code or after the result is handed to the middleware. In such a case, it is desirable to not only control the custom application logic but to also have the ability to set breakpoints and inspect attribute values and execution states in the middleware itself. Meaningful interpretation of the actual execution state, on the other hand, requires access to the source code of the component under inspection. Many Grid middleware solutions are provided as an open source solution so that the source code is generally available and can be made available to the IDE's debugging component.

The assumption that exactly the same version of a component is deployed on each node involved in the current execution of a Grid application does, however, not hold for large scale Grid environments. Different versions of certain components may successfully be

combined in one Grid environment if they still implement the same or interoperable interfaces. In this scenario, the IDE integrated interactive debugger is extended by a Grid enabled support component. If a developer has the right to control a certain node for debugging purposes he may use a special debugging service to attach to the Grid node, query for detailed version information and even retrieve the component source code for use in the integrated debugger. The debugging service client is then used to supply the interactive debugger with the right information that can then present the right source code for the component under inspection to the user.

The debugging service itself can be offered as a Grid service, and standard access control mechanisms for Grid services can be applied to the platform debugging service. Figure 2 shows the debugging support components in the lower part of the IDE. The integrated debugger is assisted by a client to the debugging support service in the remote Grid service container. This client is used to gather the specific information about the deployed components. Source code for the components can be retrieved from a module source code repository in order to make the source code available to the integrated debugger. Note, however, that the debug information service must be independent of the Grid service container under debugging since access to the service may otherwise be obstructed by the suspension of threads by the debugger.

### 3 Implementation

In this section, we describe our concrete implementation of the GDT for the Eclipse platform. Our current implementation of the GDT includes target system mapping implementations for the Globus Toolkit 4 and the Marburg Ad-hoc Grid Environment (MAGE)<sup>3</sup>.

The entire Eclipse platform is implemented as a small core runtime environment for plugins and a large collection of plugins that provide the functionality of the platform. Plugins define so called *extension points* and implement extension points defined by other plugins to contribute their functionality to each other. The Eclipse platform supports a *headless mode*, i.e. running a so called Eclipse application without the graphical workbench user interface. A standard headless application of the Eclipse platform provides Ant functionality on the command line. In this mode, the Eclipse platform acts like a standard Ant distribution taking a build file as input and making all build tasks defined by plugins in the Eclipse installation available to the build file.

The GDT's functionality is provided to the Eclipse platform through a number of different plugins. We strictly separate core functional components from user interface components to ease the re-use of the core functionality in headless mode of the Eclipse platform. For space reasons, we present only the implementation of the service creation functionality in detail in this paper.

---

<sup>3</sup><http://ds.informatik.uni-marburg.de/mage>

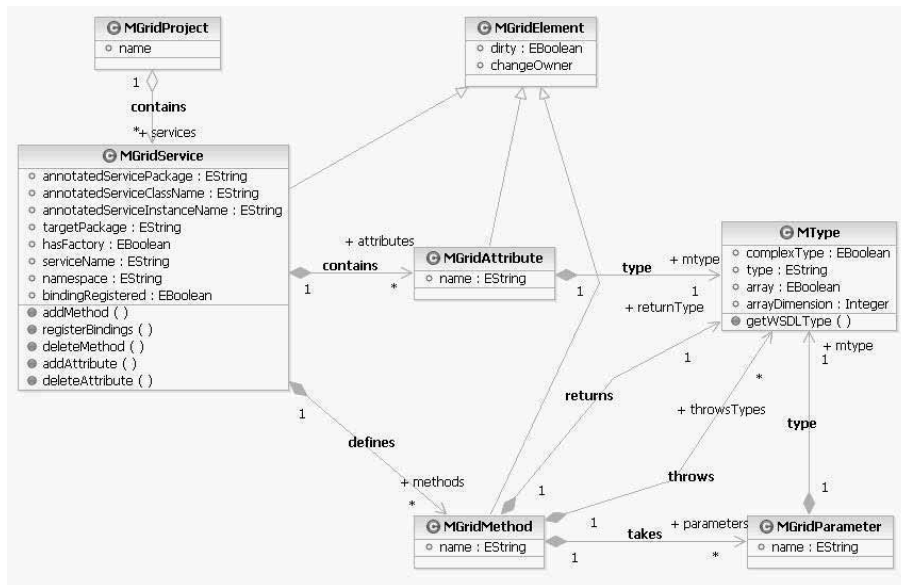


Figure 3: Meta-model for the upper layer Grid PSM.

### 3.1 Component Creation

As a first step towards an implementation of the GDT, we devised the meta-model for the upper layer Grid PSM. The resulting meta-model is shown in figure 3. The basic components of this meta-model are a *Grid project* that can contain many *Grid services* that in turn have a number of *Grid attributes* and *Grid methods*. Following the WSRF definition of a WS-Resource as the union of a stateless web service and a corresponding resource property document capturing the state data of the resource, the Grid attributes are the elements of a service forming the resource property document. Every attribute, method parameter, return type or exception thrown by a method has an associated type that can either be a simple type or a complex type. The Eclipse Modeling Framework (EMF) is used to automatically generate a Java implementation of the meta-model. Instances of this representation are used throughout the GDT for representing concrete upper layer PSMs.

We have defined several Java annotations (*GridService*, *GridMethod* and *GridAttribute*) to represent the connection between application logic carrying code and the upper layer PSM of a Grid service. The Java annotations can be used to define additional meta-data such as the target namespace for the Grid service. The annotations are intended to be used by developers in their applications in order to mark certain logical elements for inclusion in a Grid service. In terms of the GDT, a Java class carrying the *GridService* annotation is called an *annotated service class* and may be interpreted as a model source for the upper layer PSM.

As a first extension to the Eclipse platform, we defined and implemented a project nature

for *Grid projects* (the `GDTNature`) that can be assigned to any Java project in the user's workspace to allow associating our GDT plugin with the project. Apart from the project nature, the Eclipse platform assigns an ordered set of project builders to a project. These builders are invoked whenever the change of one or many resources forces an incremental or full build of the project or when the project should be "cleaned". When a new service is added to a Java project, the GDT assigns the GDT nature to the project and a GDT builder which is running after the Java builder of the Java Development Tools (JDT) plugin.

The GDT builder is the central component handling the internal transformation between the upper and lower layer PSM of a project and the central contribution of the GDT core plugin to the Eclipse platform. The builder receives a so called delta set from the Eclipse platform. This delta set contains a reference to all resources that were changed since the last build was performed on the project. For an incremental project build, this is the set of resources actually touched by the user (or an action by the user induced on the project). The GDT builder now performs two operations in sequence. First, a delta visitor is used to check every resource in the delta set for the project whether it is a model source. Second, the model transformations and emitters are invoked to push changes in the model into the actual resources.

To perform these operations, the GDT builder needs to make a connection between resources and the elements of the project model. An internal binding model defines a binding element that is directly connected to a workspace resource and holds references to a service model element, a resource emitter and a resource interpreter. Binding elements are collected by the GDT plugin in a binding registry that can be used to look up bindings for a given service model or a given resource. All resource interpreters in the GDT implement two methods `isSource` to determine whether a resource is a model source and `read` to actually work on the internal models. The GDT uses the annotation processing facility of the standard Java Development Tools (JDT) to process annotations in Java classes by contributing a custom annotation processor.

Most code emitters of the GDT are implemented using Java Emitter Templates (JET) technology, a part of the Eclipse Modeling Framework. JET uses a syntax similar to Java Server Pages, allowing to use regular Java code within special code sections of the template. Everything outside of the special code tags is literally copied into the output. Since there is a trivial mapping between the lower level PSM and the actual source code for the application, the GDT can directly emit source code from the upper layer PSM. This is also true for non-Java resources such as deployment descriptors and WSDL files for the target service. JET is supplemented by *JMerge*, an EMF component allowing to merge newly generated versions of a Java resource with previous versions that might contain user modifications. *JMerge* transfers the user modification into the newly generated resource preventing loss of user source code. Apart from the template based code emitters, the GDT implementation relies on existing code generators for the GT4 framework.

The generators are implemented as a combination of Ant build scripts and small code generators implemented as Java programs that get called from the Ant build files. These tools are used to generate, for example, stubs for the resulting Grid service. They perform their work in a rather long running process (around 20 seconds on a Pentium 4 Mobile 1.7 GHz system) that is hard to include in an automatic and incremental project build.

Fortunately, those long running generation steps target only automatically generated platform binding code a human developer is unlikely to ever modify. Developers will rather focus on changing the application logic or the code corresponding directly to the lower layer PSM generated by our template based approach. Since the stub generation process is such a long running operation, we give developers the opportunity to manually start the stub generation process for a service as a last development step prior to service packaging and deployment or when they see substantial changes in the service model that requires (re-)generation of the stubs. The resulting source code of the stubs is automatically added to the project and can be modified by the developer within the workbench. The second long running operation that can directly be triggered by the user is the final packaging of the Grid service for deployment. The packaging operation for both target systems we implemented also uses existing Ant scripts and integrates them into the workbench.

All transformation components of the GDT are split between the GDT core plugin and so called *target system mapping* modules (TSM). The core plugin provides the meta model for representation of Grid services as well as common and generic services such as annotation processing and model registries, while the TSMs contribute their interpreters and emitters to the core plugin.

Developers are supported in the creation of new services by a multi-page wizard. The GDT user interface component queries the platform for all available TSM contributions. The first page of the wizards queries common information about the new service such as its target namespace, name, the package and class name of the annotated service class to be created. The GDT user interface queries the Eclipse platform for all available TSM contributions and collects a set of TSM identifiers that the user may select a concrete target system from. This selection determines additional wizard pages that are contributed by the individual TSMs and may allow the user to specify additional choices and values. After finishing the wizard execution, the service model is created and all necessary project settings are changed according to the user's choices. Most importantly, the TSM to be used for transformation is associated to the service in the project. This TSM is then used to generate all service artifacts including a base implementation of the annotated service class that the user may fill with his/her application logic.

The addition of a service to a project does not rely on the graphical user interface components. The GDT contributes a custom Ant task `gdt.generator` and a headless application `de.fb12.gdt.Generator` to the Eclipse platform that can be used without the graphical workbench user interface. Their basic intention is to enable the user to automatically create a Java project for an annotated class or add the service to an existing project, generate the target system specific artifacts and package the service. They also offer the ability to initialize a new workspace, and to automatically import any number of Java libraries and Java source files into a service project. This functionality is intended for use in automated build or test environments that are very common for large Grid applications. Both components can be regarded as different user interfaces to the core components of the GDT. Therefore, the implementation of both components is part of the GDT UI plugin. It relies on functionality for project creation and modification that is provided by the core plugin.

### 3.2 Process Creation

We have implemented a distributed, graphical process editor for Grid processes based on the BPEL language. The editor is implemented strictly using the MVC pattern. For the user interface part, the Eclipse Graphical Editing Framework (GEF) was used. Modification of the underlying process model happens through commands emitted by the controller component of the editor. These commands are distributed to other remote process editor instances that are connected to the same group via the Eclipse Communication Framework (ECF). Remote commands are scheduled as asynchronous commands in the user interface thread of the local editor instance, sequence ordering of remote commands is performed by a group controller. The resulting editor supports interactive collaborative editing of a BPEL based process within the Eclipse framework.

We have implemented two controller and view instances allowing for high-fi and low-fi editing of the process model. The low-fi view defines filters for the user interface that expose only certain properties of the model element to the user. Missing values are assigned by associated wizard elements that implement the assignment of default values.

### 3.3 Interactive Debugging

Our implementation of the interactive debugging support component is based on the Eclipse platform debug component and the integrated graphical Java debugger. This component already allows to connect to a target JVM over a network connection when the target VM has been instructed to load a custom *agent library*. This library interacts with the JVM via the *Tools Interface* (JVM-TI). In addition to the regular debugging agent library, we have implemented a custom *Grid information agent library* that may be loaded additionally to the debugging agent library. The GDT debug plugin contributes a *source locator* component to the platform that queries the target VM for detailed version information about the module under inspection when the debugger suspends execution at a certain stack frame. The additional information is used to gather source archives for the component and presents the source code of the current point of inspection to the debugger. This ensures that the developer can suspend execution also in platform components and has access to the component's source code in the right version. Source archives are published in the Grid environment and may be retrieved from either the node under inspection or other development environments that re-publish the copies they obtain.

## 4 Evaluation

To evaluate the GDT, a video cut detection application was developed as part of the Mediana Project<sup>4</sup>. The cut detection algorithm requires a two-phase processing of the input

---

<sup>4</sup><http://www.fk615.uni-siegen.de>

data. First, a number of features are extracted from the video data. Second, these features are used to determine the location of cuts in the input video. Since feature extraction is the most time consuming part of the application, this step in the algorithm workflow should be distributed over different nodes in the Grid environment.

The core service implementation was created to implement feature extraction using the GDT within the Eclipse workbench. The resulting `FeatureExtractor` service has one public method `extractFeatures` that takes a collection of references to video segments as input and returns the corresponding feature lists as a collection of references to the client.

After having created the annotated service implementation stub with the Grid service wizard, the developer is required to add 9 lines of code manually. This code is required to receive the input data and make a call to the existing feature extraction code in the video analysis library. Upon saving of the source file, the GDT builder automatically generates 5 new classes with 311 lines of code and 6 interface description and configuration files with 151 additional lines of text, that are required as an input to the existing tools for creating the final Grid service binding code and packaging of the service implementation. The developer then needs to invoke the last step in the integrated Grid service development process to build and package the distributable Grid service archive.

On a Pentium 4 Mobile 1.7 GHz system with 1GB of RAM running Windows XP, the transformation from upper layer PSM to lower layer PSM and the creation of the corresponding artifacts takes roughly 1 seconds. Even for a Grid service containing 25 methods and attributes, which is more than most Grid services need, the time does not increase significantly. This time constraint is acceptable for performing interactive development work on the annotated service class or the generated lower layer PSM without too much interference to the developer. The existing build tasks for stub generation and packaging of the service are long running jobs that take between 15 and 25 seconds to finish, which is another reason to only carry them out on explicit request by the user. Headless operation and the use of the Ant target from the command line adds approximately 10 seconds for the initialization of the Eclipse platform components. The overall time required to automatically import an annotated service class into a newly created workspace and project, generate, build and package the service is around 50 seconds. In our automated test environment, the same task takes about 35 seconds on a desktop Pentium IV 3GHz systems with fast S-ATA drives. In both cases, the performance of the generator supports the intended use cases for automatic service generation and interactive development.

In addition to the feature extraction service, two other services are required for the distributed execution of the feature extraction, a video splitting service that can separate a single input video into smaller chunks for feature extraction and a feature merging service that combines the results from the individual feature extraction steps. Again, the two services were created using the Eclipse workbench. Both contain roughly 20 to 30 lines of application logic carrying code and are marked as Grid services with the corresponding annotations.

After having finished the creation of the component services, the developer can compose them into a higher level application workflow using the Grid process editor of the GDT.

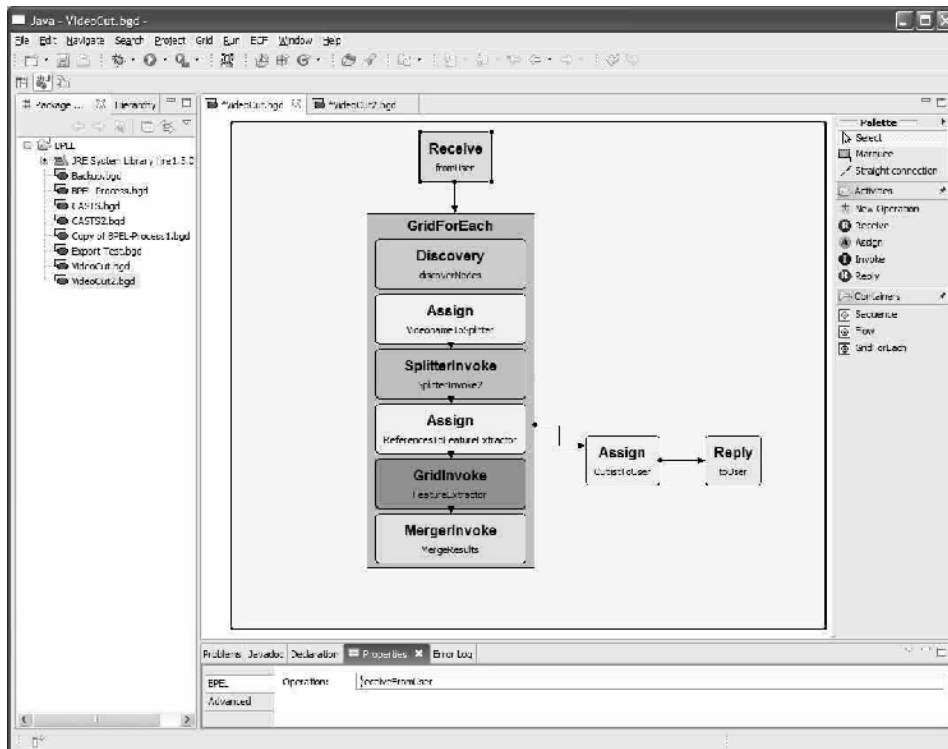


Figure 4: Video cut workflow in GDT

This process can contain steps to invoke the hot deployment service of MAGE to prepare the execution environment and distribute the feature extractor service to as many nodes in the Grid environment as possible. The application developer can also use the integrated management functionality in the Eclipse workbench to directly deploy the services to appropriate nodes in the ad hoc Grid environment. A screenshot of the actual workflow in the GDT workflow editor is shown in Figure 4.

Using an existing video analysis library and the GDT, it was possible to create the Grid application described above in a matter of hours not days or weeks compared to the traditional approach of hand coding all Grid services and the corresponding workflow.

## 5 Related Work

Since service oriented Grid computing is an extension of the service oriented computing paradigm, there are several relevant papers dealing with applying MDA in a web service environment.

A WSDL centric approach is presented in [Mul05]. The paper discusses which compo-

nents of a service belong to which layer of the MDA approach. Definitions, Operations, Port type(s), Messages, Parts and Part type(s) are placed in the PIM layer. Service, Ports and Binding(s) are placed in the PSM layer. We do not agree with this mapping since most of the components placed in the PIM are specific to a service oriented approach and should therefore be placed in the PSM layer. The paper goes on to suggest that a document centric view is better suited to service oriented models than an UML centric view.

A WSDL free approach is presented in [GSSO04]. A WSDL independent UML model is proposed because it offers a much clearer view of the system functionality. Automatic transformations from UML to WSDL are used to create the actual WSDL document. Since the UML model is completely free of any WSDL specific components, the developer is free to concentrate on actual business concerns. The downside is that an integral part of service oriented systems is no longer visible in the MDA models and thus outside of the development scope.

Manset et al. [MMOV05] combine a formalized architectural description with the model-driven engineering process proposed by MDA. Great emphasis is placed on the formal specification and verification of a large number of sub-models using a special architecture description language. The complexity of the formal specification and the large number of sub-models make this a hard to adopt approach not suitable to application domain experts.

Mizuta and Huang present an approach to Globus GT3 service generation using a model-driven approach based on the AndroMDA toolkit [MH05]. This proposal is only geared towards generation of a Grid service, not round trip engineering. Furthermore, the approach targets a single Grid platform, whereas our approach is explicitly designed to allow fast adaptation to other service oriented Grid middleware target systems and includes more development tools for e.g. debugging an process creation not addressed by Mizuta and Huang.

Furthermore, there are several related software projects that deal with development support especially in service creation.

The Web Tools Project (WTP)<sup>5</sup> provides wizards for the creation of Java Web Services and models for the representation of standard web service artifacts. These tools do not deal with Grid specific issues at the moment, but are geared towards tight AXIS (a Java Web Service Framework) integration. Furthermore, *round trip support*, i.e. allowing developers to change the model or the generated sources and reflect the changes in all corresponding model or source elements, is currently not supported by WTP. However, the WTP offers many useful editors for web service related formats such as WSDL files that can be seamlessly used for artifacts created and managed by the GDT.

The Generative Model Transformer (GMT)<sup>6</sup> project's aim is to "produce a set of prototypes in the area of Model-Driven Engineering". ATL is an emerging subproject in the GMT project aimed at providing development tools for the Atlas Transformation Language [JK05]. Unfortunately, the project has not released any usable software component yet.

---

<sup>5</sup><http://eclipse.org/webtools>

<sup>6</sup><http://www.eclipse.org/gmt>

The GT4IDE<sup>7</sup> project's aim is to integrate a service creation tool to Globus Toolkit 4 based Grid services into the Eclipse platform. The GT4IDE is focused on a single Grid platform (the GT4IDE), it does not use a model-driven approach to Grid service development and does not support full round-trip-engineering (only limited support for synchronisation between WSDL artifacts and Java service implementation is provided), process creation or Grid integrated remote debugging.

## 6 Conclusions

In this paper, we presented the overall design of an integrated software development environment for service oriented Grid applications and its implementation within the Eclipse platform. Our system is targeted at non-Grid middleware experts, allowing them to rapidly develop Grid applications by hiding the complexities of the Grid middleware. Three main components dealing with service creation, process creation and interactive debugging were presented and a detailed description of the implementation of the service creation component was provided.

Using our Grid Development Toolkit in several projects (InGrid<sup>8</sup>, MediGrid<sup>9</sup>, Mediana<sup>10</sup> and several student projects), we have observed improvements in the development time required by various application domain experts (e.g. engineers, media scientists, medical research) to bring their application into a service oriented Grid environment. Developers were able to reduce the time required to develop the necessary Grid services for their application within hours instead of weeks they needed prior to using the GDT.

There are several areas for future work, such as providing support for further Grid computing platforms like Unicore-GS and further integration of management components for different Grid environments with the development support components (especially debugging support). Additionally, we plan to investigate the possibility of integrating general model-to-model transformation languages and systems into the GDT for the various model transformations in a transparent way for the end user. Finally, since the implementation of the distributed Grid process editor showed great potential for non-Grid experts, we intend to develop additional wizards for process and property assignment in different application areas.

## 7 Acknowledgements

This work is partially supported by IBM (Eclipse Innovation Grant), the German Ministry of Education and Research (BMBF) (D-Grid Initiative, In-Grid Project) and Siemens AG (Corporate Technology, München).

---

<sup>7</sup><http://gsbt.sourceforge.net/content/view/12/29/>

<sup>8</sup>[www.ingrid-info.de](http://www.ingrid-info.de)

<sup>9</sup>[www.medigrid.de](http://www.medigrid.de)

<sup>10</sup><http://www.fk615.uni-siegen.de>

## References

- [Bro04] Alan Brown. An Introduction to Model-Driven Architecture Part I: MDA and Today's Systems. *IBM Whitepaper*, pages 1–15, 2004. IBM The Rational Edge.
- [FBD<sup>+</sup>04] I. Foster, D. Berry, A. Djaoui, A. Grimshaw, B. Horn, H. Kishimoto, F. Maciel, A. Savvy, F. Siebenlist, R. Subramaniam, J. Treadwell, and J. Von Reich. The Open Grid Services Architecture, Version 1.0. Whitepaper GGF, 2004.
- [Flo03] Franco Flore. MDA: The Proof is in Automating Transformations Between Models. In *OptimalJ White Paper*, pages 1–4, 2003.
- [GSSO04] R. Gronmo, D. Skogan, I. Solheim, and J. Oldevik. Model-Driven Web Services Development. In *Proceedings of the 2004 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE04)*, pages 633 – 649, 2004.
- [JK05] F. Jouault and I. Kurtev. Transforming Models with ATL. In *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*, pages 128–138, Montego Bay, Jamaica, 2005.
- [MH05] Sachio Mizuta and Runhe Huang. Automation of Grid Service Code Generation with AndroMDA for GT3. In *Proc. of the 19th International Conference on Advanced Information Networking and Applications (AINA 2005)*, pages 417–420, 2005.
- [MM01] Jishnu Mukerji and Joaquin Miller. Overview and Guide to OMG's Architecture. *IBM Whitepaper*, pages 1–62, 2001. IBM The Rational Edge.
- [MMOV05] David Manset, Richard McClatchey, Flavio Oquendo, and Herve Verjus. A Model-driven Approach for Grid Services Engineering. In *Proc of the 18th International Conference on Software and Systems Engineering and Applications*, pages 51–58, Paris, France, 2005.
- [Mul05] Ranjit Mulye. Modeling Web Services Using UML/MDA, 2005.  
<http://lsdis.cs.uga.edu/~ranjit/academic/essay.pdf>.
- [OAS04] OASIS. Web Services Resource Framework, 2004.  
[http://docs.oasis-open.org/wsrp/wsrp-ws\\_resource-1.2-spec-os.pdf](http://docs.oasis-open.org/wsrp/wsrp-ws_resource-1.2-spec-os.pdf).
- [Oas05] Oasis WSBPEL TC. Web Services Business Process Execution Language Version 2.0 - draft, December 2005.  
<http://www.oasis-open.org/committees/wsbpel/>.
- [Obj03] Object Management Group, Inc. MDA Guide 1.0.1, omg/2003-06-01, June 2003.  
<http://www.omg.org/docs/omg/03-06-01.pdf>.
- [SFF06] M. Smith, T. Friese, and B. Freisleben. Model Driven Development of Service-Oriented Grid Applications. In *Proc. of the International Conference on Internet and Web Applications and Services*, pages 139–146, Guadeloupe, 2006. IEEE Press.
- [Tho04] Dave Thomas. MDA: Revenge of the Modelers or UML Utopia? In *IEEE Software, May/June*, pages 15–17, 2004.

