

The Otho Toolkit: Generating Tailor-made Scientific Grid Application Wrappers

Juergen Hofer Alex Villazón Mumtaz Siddiqui
Thomas Fahringer

Distributed and Parallel Systems Group,
University of Innsbruck, Technikerstrasse 21a, 6020 Innsbruck, Austria

Abstract: Grid technologies and middleware components have shorter lifecycles than most scientific applications. The smooth and automatic migration of existing codes to modern environments continues to be a major research issue. In our work we address this problem from a novel perspective (MDA and for a specific problem domain (scientific grid applications)). We demonstrate how we apply results and techniques from the Model-driven Architecture and Code Generation fields to automatically create Scientific Grid Application Wrappers in form of different types of services among them plain Web services or WSRF services. The wrappers are enriched with non-functional property extensions such as performance analysis, resource-accounting, file-staging or execution management for instance. Design and implementation of our prototype with accounting of CPU consumption as non-functional property is exhibited. The "*Invisible Grid*" is an ideal Grid environment where users are completely shielded from any Grid specific details. Our research is ongoing work. We present our achievements so far on getting one step closer towards this vision.

1 Introduction

Scientific applications continue to be the main drivers for the rapid development of many large-scale Grid projects such as EGEE¹, CrossGrid² or Kwf-Grid³. In their contexts, the Grid serves primarily as environment for the execution of large-scale scientific applications with high resource demands. Those applications are often specific to a scientific domain and user community (e.g. astrophysics, chemistry, bioinformatics), efficiently implemented using performant programming languages (C/C++, Fortran), support computational and/or data parallelism and require high computation, communication and storage capacities. In addition, such kind of applications are often difficult to maintain, reuse and extend because often they have complex and monolithic structures.

One of the central objectives of *Grid Computing* efforts, such as Globus [FK99] or UNICORE [AS99] is to provide a uniform user and programming interface to heterogeneous

¹<http://egee-ei.web.cern.ch/egee-ei/2004/index.html>

²<http://www.eu-crossgrid.org>

³<http://www.kwfgrid.net>

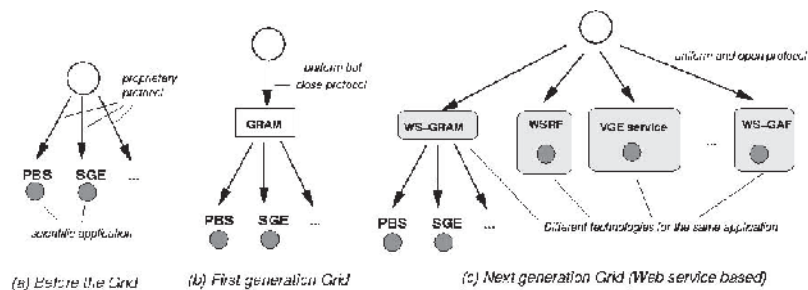


Figure 1: Evolution of Grid technology to run scientific applications

computational and storage resources. Currently, Grid computing is evolving from closed and proprietary *batch-oriented* protocols, to open and service-oriented environments. The Grid community noticed the large impact of service-oriented architectures and Web services and developed the Open Grid Service Architecture (OGSA) [FKNT02] as an open, service-oriented architecture for Grid computing, where every resource is represented by *Grid services*. Recently, the *Web Service Resource Framework* (WSRF) [C⁺04] was proposed to fully align Grid and Web service communities. Other efforts such as Web Service Grid Application Framework (WS-GAF) [PWWR03] or WS-I⁴ however compete to be used to build WS-based Grids. Figure 1 shows the evolution of Grid technology and how scientific applications can be executed. Before the Grid (Figure 1.a) scientists had to interact with different batch systems to execute their applications each having different languages, protocols and tools. With the first generation of Grid middleware (Figure 1.b) the complexity of the underlying systems was shielded through a uniform interface (e.g. GRAM), but still was based on a proprietary protocols. Later on, the use of Web service technologies introduced standard interfaces and protocols (Figure 1.c). This allows to implement the scientific application in form of a *Grid Service* in two different ways: Through a generic web service interface for job submission, such as WS-GRAM (i.e. the WSRF version of GRAM in Globus 4) or through tailored web service wrapper for the application, which are specific to the different Grid environments that are used.

In order to execute scientific applications users usually have to write rather complex *scripts* or programs to coordinate the execution. These scripts are tailored to use specific features of the different grid environments and are usually not reusable. In addition, the diversity of technologies and platforms has a major disadvantage for the application developer and user, since the adaptation of the application to different grid platforms is time-consuming and error-prone. Ideally, the user should be able to run his application without caring about the underlying grid technology⁵. Unfortunately, there do not exist any tools to automate the adaptation to different target Grid environments.

In this paper, we propose *Otho*⁶, which is a step towards the automatic generation of Grid

⁴<http://www.omii.ac.uk>

⁵This vision is commonly referred to as the *"invisible Grid"*, where the user is shielded from Grid specific details.

⁶Otho is a member of starship Comet's crew commanded by Captain Future from the US Pulp magazine

services to *wrap scientific application* components for different service-oriented grid environments. Our approach is based on ideas from Model Driven Architecture (MDA) [MM03], which allows to specify transformation rules of models, from a platform-independent specification of a system, into different platform-specific systems, i.e. for particular target platforms. Figure 2 shows the overall approach. Input to the first step is a black-box description of the functionality of the programs that is internally parsed and converted to a platform-independent model (PIM). In *Otho*, the PIM abstracts from all platform details (languages, artefacts, libraries, APIs protocols). Model transformation rules describe how to transform the PIMs to different platform-specific models (PSM). The PSM holds complete information of the code that will wrap the legacy application on different grid environments. In order to transform from the PIM to the PSM in *Otho* additional information is needed. Descriptions of the required non-functional properties, information on the target platform and target system. The transformation is done using a model compiler. In addition to the functional description of the application to be wrapped, *Otho* also allows to specify non-functional requirements (i.e. constraints that are orthogonal to the main functionality of the application such as logging, performance measurement, accounting of resource consumption, security, etc.) in a platform independent manner. This allows for example to measure the performance of the wrapped application without modifying the original code.

The output of *Otho* are deployable services in an appropriate format defined by the target platform (e.g. a GAR archive in the case of Globus Toolkit 4). The services are then deployed by and registered in GLARE [SVHF05], a Grid activity registration, deployment and provisioning framework developed by our group. We have applied our approach to a simple grid workflow application, where each component is wrapped to a different web-service based grid environment. Our prototype uses the Eclipse Modeling Framework (EMF) [BSM⁺03] as MDA environment and the included code generation utilities Java Emitter Templates (JET). The PIM to PSM model transformation is currently implemented using EMF APIs. We have chosen as target platforms Globus Toolkit Grid service and Apache Axis Web service platform. We show how to measure the execution time of the *wrapped legacy application*, and the consumption of CPU resources of the wrapper web/grid service as examples of non-functional requirements that can be specified in *Otho*.

The rest of this paper is structured as follows. Section 2 describes our proposed solution and architecture, design and implementation of *Otho* prototype. We show our prototype platform-independent and platform-specific models and show examples of code templates and generated code. Section 3 shows how we adapt the code generation with non-functional properties. Section 5 concludes the paper.

published in the early 1940s. *Otho* is a shapeshifter android able to change appearance at will.

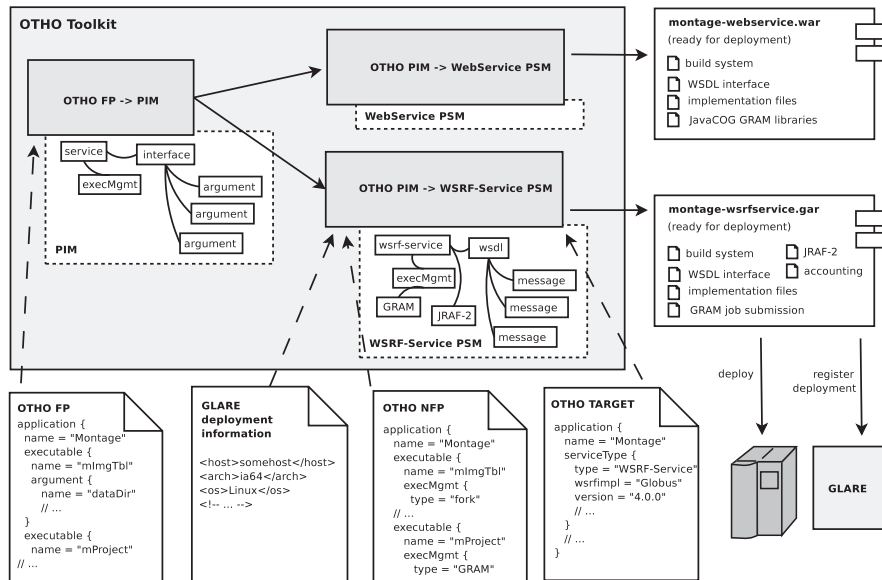


Figure 2: Adaptive generation of Grid services with *Otho*

2 Model-driven Grid Service Wrapper Generation

Model Driven Architecture by Object Management Group (OMG)⁷ describes how to create standards-based, technology independent models of concepts and then map them to different specific technologies [MM03]. Recently, some work has been done which tries to automate the generation of Grid Services code, based on transformation of UML models to Globus Grid services [MH05]. Contrarily to such approach where the full functionality including all internal aspects need to be captured in the models to generate full service implementations we take a more specific and practical approach. The models in *Otho* do not describe the complete internal functionality of a service, but only the specification of the interface of the existing components to be wrapped. Usually these component interfaces are only defined by input (parameters and input artefacts such as files) and output (return values and output artefacts) and metadata how to invoke/call it which keeps our models concise and handy. Moreover this simplifies the model transformation, since the target platform-specific model can be easily mapped from an abstract specification to platform-dependent interface invocations.

To better understand our approach, let's consider the *Montage* scientific application. *Montage* is a software system for generating astronomical image mosaics applying sophisti-

⁷<http://www.omg.org>

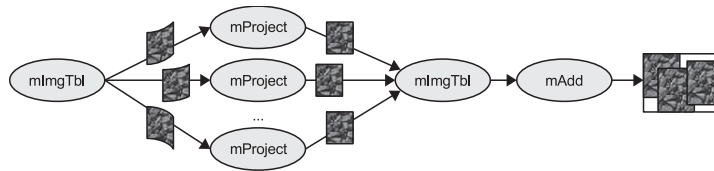


Figure 3: Simplified Montage workflow

cated projection, background modification and rectification algorithms [SD04]. A simplified version of the Montage workflow is shown in Figure 3. Each of the software components takes as input one or several images, and produces images or metadata as output file artefacts. First, image metadata are generated using the `mImgTbl` executable. Then, all the images are transformed using the `mProject` executable. This program makes a projection of the images using spatial coordinates based on the previously produced metadata. Each image can be independently treated, so that the execution can be done in parallel in several different grid sites. After the projections are calculated, image metadata have to be regenerated (`mImgTbl`) and the final mosaic image can be assembled using the `mAdd` component. `mProject` and `mAdd` are rather CPU-intensive tasks and therefore ideal candidates for execution on the grid.

The workflow consists of *computational tasks* and *data-transfer tasks* (file staging) that move image and metadata files between grid sites. In order to execute the workflow on the Grid, we want to wrap the execution of each component on a service-oriented interface, i.e. create a standard Grid or Web service that hides the actual execution on the underlying grid infrastructure. The wrapper itself does not realize the application functionality, but rather executes the code by using a job submission system (such as GRAM or WS-GRAM), or spawning a new process that executes the application. The following code fragments show the final result that we want to obtain in order to wrap Montage components (a) through Job submission script using GRAM, i.e. a platform-specific interface for job submission in Globus, and (b) by local subprocess forking. These code fragments could then be part of different types of java-based Wrappers such as an Java Web service or a Globus Grid service.

The Web Service provides an execution interface for the different components of Montage by combining the Adapter and Proxy design patterns [GHJV94], which have been proven to be useful architectural models to wrap executable units [JK98]. The Adapter allows components to communicate in spite of incompatible interfaces and the Proxy acts as intermediary to a remote object possibly adding functionality. Each activity in the workflow can be wrapped by a different service type supporting different non-functional properties and to adapt to the environment. For example, the `mProject` and `mAdd` components are wrapped with a Grid service that submit jobs through GRAM, since those are the most CPU intensive tasks on the workflow, and the file-staging is also added to execute these wrappers. On the other hand, the `mImgTbl` is wrapped through a simple Web service, which only executes the command locally because it is lightweight.

(a) Wrapper that submits a job to GRAM

```
public void mProject(String dataPath, ...) throws RemoteException {
    RslAttributes rsl = new RslAttributes();
    rsl.add("executable", "/myPath/mProject.sh");
    rsl.addMulti("arguments", new String[] { statusfile, infits,
                                             outfits, hdrtemplate });
    GramJob job = new GramJob(proxy, rsl.toRSL());
    job.request(myhost.mydomain.org);
    // ...
}
```

(b) Wrapper that uses local subprocess forking

```
public void mProject(String dataPath, ...) throws RemoteException {
    String[] command = new String[] { "/myPath/mProject.sh",
                                       statusfile, infits, outfits, hdrtemplate };
    Process process = Runtime.getRuntime().exec(command);
    // ...
}
```

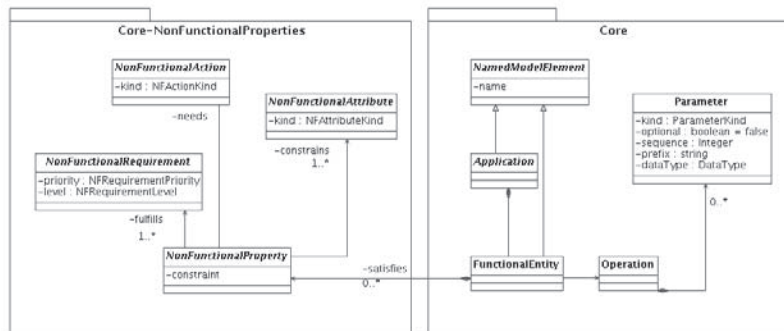


Figure 4: Subset of the Otho Metamodel

2.1 Wrapper Generation metamodel

In order to minimize the amount of interference and manual code construction, a maximum of information should be encoded in the appropriate models (platform-independent, platform-specific) and their transformations. The existing scientific application and its executable components are treated as black boxes with a well-defined interface and their source code is not modified. The only necessary knowledge about the executable components is how to use them and which artefacts they consume or produce. Internals such as what they do and how they work is not of interest in this context.

The first step for the automatic wrapper generation is the definition of the metamodel for our PIM and PSM. The metamodel relates to the model like a grammar to a formal language in that it restricts the space in which modeling elements can be arranged. The requirements and concepts that we identified, start from an abstract representation, i.e. the *platform-independent model* (PIM), that was designed to cover all the relevant information allowing the generation of the artefacts used to build our wrappers. The central generic part of this metamodel is depicted in Figure 4.

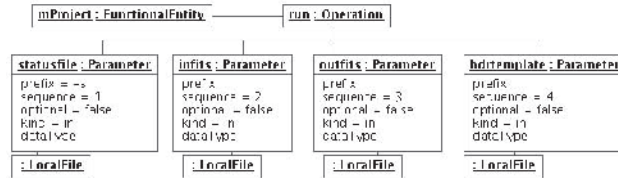


Figure 5: Subset of an instance for Montage mProject of the Otho Metamodel

Part of our metamodel contains generic and abstract model elements that will be further refined for the concrete application context. The central model element is *Application* that consists of *Functional Entities*, characterized by attributes, and a single *Operation*. The *Operation* may have zero or many *Parameters* and each one has a *dataType* associated to it. For example, in the case of Montage, the *Application* will be mapped to *Montage*, the *Functional Entities* will be mapped to each of the single component (i.e. mImgtbl, mProject and mAdd). Depending on the platform that is used, the *Operation* will be mapped to *run*, *execute* or *submitJob* for example, and of course each of the operations will accept different *Parameters*, such as arguments or files that are required for their execution. We create model instances of this metamodel, for each application that we want to wrap. The corresponding instance for the mProject tool is depicted in Figure 5.

In opposition to functional requirements (i.e. what an application is expected to do), non-functional requirement and properties (NFRs) specify global constraints that must be satisfied by the application [NSRC00]. Dealing with non-functional properties is problematic because of their complexity, implementation hurdles and difficulties in formalizing and expressing them. For instance, depending on the application, *security* may or may not be considered as a non-functional property, and often finds itself in conflict with others non-functional properties like *performance* (e.g. authenticating users or encrypting messages costs time hence reducing the system performance). Designing a model to be generic enough to cover the needs of a large variety of scientific applications across several domains, and still being concise and easy to use is a hard challenge. In our work, we restrict to some of non-functional properties that are meaningful for the type of applications that we are targeting. Additionally, the usage of wrappers simplifies the integration of non-functional properties such as performance measurements and resource-accounting, since we do not need (and want) to change the original application code. We discuss in detail the integration of non-functional requirements in *Otho* in Section 3.

From the PIM perspective, we started with a simple metamodel for non-functional requirements. As shown in Figure 4, the non-functional metamodel has four main components: *properties*, *requirements*, *attributes* and *actions*. Each *Functional entity* should satisfy 0 to n non-functional properties. *Non-functional properties* are constraints over *non-functional attributes*, e.g. the non-function property "availability > 0.95" constrains that service availability needs to be higher than 95%. *Non-functional actions* are implementation techniques, code or configuration artefacts that realize non-functional properties. At the PIM level, non-functional actions are abstract and can only get concrete at the PSM level. Finally, *non-functional requirements* express the users additional non-functional

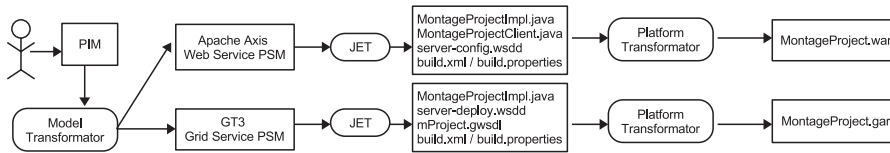


Figure 6: Steps involved in the Wrapper Service Generation

needs on the system. In the following section we concentrate on the process to generate the wrappers based on the platform-independent model.

2.2 Wrapper Generation for Montage

We implemented a prototype of *Otho* using the Eclipse Modeling Framework (EMF) [BSM⁺03] as MDA environment and Java Emitter Templates (JET) as code generation utility. We designed a platform-independent metamodel (PIM) to cover all aspects of the application units (functional entities), which are relevant for the wrapper generation. Two platform-specific metamodels (PSM) were defined: (a) for Grid services using Globus Toolkit that use GRAM job submission (b) and for Web services using Apache Axis and local subprocess forking.

Figure 6 depicts the overall process for the generation of the code for wrapping Montage components on the different target platforms. The user creates the description of the applications, i.e. creates a model instance, using graphical editors. These editors are automatically generated from the metamodel using EMF facilities and allow to modify the elements of the tree-based representations of the model. After creation of the PIM instance, the user saves to model in its default EMF serialization to a file and starts the *model transformation tool* (i.e. the model compiler) that deserializes the PIM and writes its results to the new PSM serialization file. The PIM to PSM model transformation tool was implemented Java using the EMF APIs. The PSM contains all the relevant information from the PIM, and some default values for new settings.

Since the PSM model generated by the model transformation tool is in XMI format, the user has the possibility to use the EMF model editor to modify or adapt some of the PSM attributes, if required. Figure 7 shows the data management subset of the PIM (a), the WS-I Web services PSM (b) and the eclipse-based model editor generated from the EMF metamodel (c). The WS-I Web services PSM contains all the necessary artefacts for our wrapper Web service that we want to generate. For instance, the necessary Apache Ant *build scripts* and configuration files (*build.xml* and *build.properties*) that have to be generated are shown in Figure 7 (c).

```

public static Application createMontageInstance() {
    Application application = coreFactory.createApplication();
    application.setName("Montage-2.2");
    // ...
    FunctionalEntity clie2 = coreFactory.createFunctionalEntity();
    clie2.setName("mProject");
}
  
```

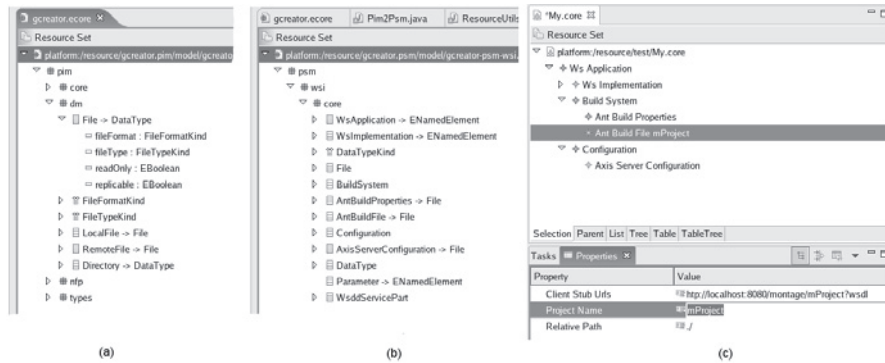


Figure 7: (a) Subset of data management part of the PIM (b) Subset of WS-I Web services PSM (c) Generated Model Editor for the WS-I Web services PSM

```

clie2.setExecutable("/opt/Montage-2.2/bin/mProject");
clie2.getInParameters().add(createParameter("statusFile",
                                           "-s", stringDataType));
clie2.getInParameters().add(createParameter("infits",
                                           null, stringDataType));
//...

```

The code fragment printed above is part of the model transformer and illustrates how the EMF APIs generated from the metamodel are used to build and manipulate model instances programmatically. The `Application` and `FunctionalEntity` classes were generated from the EMF metamodel. Once the platform-specific models are created, JET is used to generate the source code and the configuration file artefacts. For each file artefacts, we created a JET template file like the following.

```

<%@ jet package="org.gcreator.generators" class="WsImplementationTemplate" ... %>
<% WsImplementationPsm model = (WsImplementationPsm)argument; %>
package <%= model.packageName %>;
public class <%= model.className %> {
    private static HashMap instances = new HashMap();
    public int <%= model.getMethodSignature() %> throws RemoteException {
        String[] command = new String[] {
            <%= model.cliExecutable %>,
            <% for (int i=0; i<model.params.length; i++) { %>
                <%= model.params[i] %><% if (i<model.params.length-1) %>
                { %><%= NL %><% } %>
            <% } %>
        };
        Process process = Runtime.getRuntime().exec(command);
        // ...
    }
}

```

JET uses a template based mechanism similar to Java Server Pages (JSP)⁸ with a template language that is transformed into a Java classes with a generate method, that constructs and returns the generated code as string. The template language provides *scriptlets* to

⁸<http://java.sun.com/products/jsp>

embed any Java code and expressions that are evaluated marked with special tags. The EMF model (or in fact arbitrary class) can be accessed to retrieve the model information. We decided not to directly access the EMF model from JET but to use helper classes that internally access the EMF model instead. The helper classes provide convenience and utility methods such as the `getMethodSignature()` that returns a full method signature in Java syntax as string.

The last step in our process of Figure 6 is the actual deployment of the generated wrappers. All the necessary steps (compilation, packaging, etc.) are automated using Apache Ant build files, that themselves are generated from EMF-model via JET. Most of these build-files are generic and require only small adaptations to be used on different target platforms. We have implemented a Java-based tool that produces deployable packages in platform specific formats i.e. web application archive (`.war` file) for web service deployment, and Grid application archive (`.gar` file) for the deployment of Globus Grid service.

3 Adapting Generation with Non-Functional Properties

The rationale behind our wrappers is twofold. First we address the semi-automatic integration into different Grid environments. This is the functional aspect described in the previous section. Second we augment utility of our wrappers by enriching them with non-functional properties (NFP) to add *additional benefit* to users. Non-functional characteristics are orthogonal to the main functionality of the application. We restrict ourselves to non-functional properties that manifest themselves in code/configuration artefacts and that are of high relevance for scientific Grid applications such as logging, performance analysis, accounting of resource consumption, security, etc. For our prototype we selected execution time analysis and accounting of resource consumption of Java applications. These are representative examples to demonstrate their integration into our wrappers and potentially reveal necessary improvements of our modeling and implementation approach.

3.1 Case 1: Monitoring Execution Time

In our first case study we are interested in collecting the real execution of our applications. This requirement is simplistic but it allows easy modeling and rapid prototyping. Table 3.1 shows the non-functional part of our metamodel for monitoring execution time. The requirement expresses the user's wish to know the execution time. This requirement is satisfied by `MONITOR_EXEC_TIME = REAL_TIME, MILLISECONDS`; the constraint 'real time in milliseconds' over the attribute `MONITOR_EXEC_TIME`. Actions describe how properties are realized in a specific environment and therefore manifest themselves only at the PSM level. In our example we assume a Java-platform allowing us to use the `System.currentTimeMillis` API to retrieve and compare timestamps.

```
private long execTime = 0;
public void mProject(String dataPath, ...) throws RemoteException {
```

Monitoring execution time	
NonFunctionalRequirement	"how long does my program need to finish"
NonFunctionalAttribute	MONITOR_EXEC_TIME
NonFunctionalProperty	{REAL_TIME, MILLISECONDS}
NonFunctionalAction	retrieve/compare timestamps via System.currentTimeMillis

Accounting of CPU consumption	
NonFunctionalRequirement	"how cpu-intensive is my program"
NonFunctionalAttribute	CPU_CONSUMPTION
NonFunctionalProperty	{BYTECODE_INSTRUCTIONS, ABSOLUTE_COUNT}
NonFunctionalAction	apply JRAF2 bytecode rewriting, add JRAF2 libs and GridService

Table 1: Declaring Non-functional properties

```

long startTime = System.currentTimeMillis();
// execute mProject as usual
this.execTime = System.currentTimeMillis() - startTime;
}
public long getExecutionTime() {
    return this.execTime();
}

```

In the code fragment above, we see the manifestation of Table 3.1 in Java source code, where `System.currentTimeMillis` is used to retrieve timestamps and the result of the comparison is stored in a member variable that is accessible via an additional public method that is also added in our generation process. If we use another target platform, e.g. submission of the application to a local or grid resource manager, then the script that runs the application needs to be adapted. The following extended script uses the Unix `date` utility to retrieve timestamps and writes the comparison result to a file.

```

#!/bin/sh
start="date +%s"
mProject.sh $*
end="date +%s"
echo "startTime=$start" >> time.stat
echo "execTime='expr $end - $start's" >> time.stat

```

Likewise to our first case study more advanced methods and utilities such as e.g. `perfex` can be used in the same manner.

3.2 Case 2: Accounting of Resource Consumption

In our second case, we experimented with white-box accounting of resource consumption of Java programs. We selected for this purpose the Java Resource Accounting Framework 2 (JRAF2), a set of tools to enhance standard Java runtime systems with resource management features [HB04]. The current version of JRAF2 allows for non-intrusive monitoring of the CPU consumption in terms of Java Virtual Machine (JVM) bytecode instructions. By applying bytecode rewriting techniques, Java class files are transformed and instrumented so that during execution each thread maintains an instruction counter indicating

jdk	mandelbrot25			mandelbrot50			wekaj48-1e3			wekaj48-5e3			wekaj48-1e4		
	\bar{x}	δ	<i>c.v.</i>	\bar{x}	δ	<i>c.v.</i>	\bar{x}	δ	<i>c.v.</i>	\bar{x}	δ	<i>c.v.</i>	\bar{x}	δ	<i>c.v.</i>
sun14	134.4	47.5	0.4	131.7	35.8	0.3	119.6	27.7	0.2	134.8	52.2	0.4	388.6	208.3	0.5
sun14rw	138.2	36.1	0.3	151.0	61.7	0.4	124.0	26.7	0.2	178.6	112.0	0.6	428.7	236.0	0.6
sun14 (b)	1.000	-	-	1.000	-	-	1.000	-	-	1.000	-	-	1.000	-	-
sun14rw (n)	1.029	-	-	1.147	-	-	1.037	-	-	1.324	-	-	1.103	-	-
sun15	155.1	95.6	0.6	155.6	59.8	0.4	122.0	25.9	0.2	156.3	100.0	0.6	266.4	194.4	0.7
sun15rw	166.4	100.2	0.6	269.4	203.0	0.8	124.2	29.2	0.2	208.1	118.2	0.6	391.4	204.1	0.5
sun15 (b)	1.000	-	-	1.000	-	-	1.000	-	-	1.000	-	-	1.000	-	-
sun15rw (n)	1.073	-	-	1.731	-	-	1.017	-	-	1.332	-	-	1.469	-	-

Table 2: JRAF2 Overhead for Mandelbrot and WekaJ48 Example Grid Services.

the number of executed bytecode instructions and bytecode blocks report about the number of JVM instructions. The wrapper generation process now needs to cover the following aspects (1) add JRAF2 runtime libraries, (2) perform bytecode rewriting of all class files using the JRAF2 rewriting tool and (3) extend the wrapper Grid service with additional public methods that expose the resource accounting information collected by JRAF2.

To accommodate for both computational- and data-intensive Grid applications, we implemented two Globus Grid services (a distributed master-slave mandelbrot fractal calculator and a disk-I/O-intensive data mining application based on Weka⁹) in order to demonstrate JRAF2-based accounting as non-functional property in our toolkit. In addition to applying our wrapper generation technique, we were also interested in the runtime overhead caused by using JRAF2 for CPU resource accounting. Remember that the Otho Toolkit is used exclusively to create service that once created and deployed run completely independent of Otho. Therefore these experiments do not analyze Otho itself but the overhead introduced when wrappers exhibit this information using JRAF2.

We prepared four different server environments¹⁰. The unmodified (original) versions of Sun JRE 1.4.2.06 and Sun JRE 1.5.0 were used as basis for our evaluation. We instrumented (jrafed) the complete bytecode classes including all third-party libraries of Jakarta Tomcat, the Globus Toolkit and our two applications. A test client¹¹ simulated varying workload by running n client programs in a single JVM in separate threads concurrently and we analyzed the measured response times¹². Table 3.2 contains the detailed results: the response time mean values \bar{x} , standard deviation δ and the corresponding coefficient of variation for two mandelbrot and three data mining problem sizes. The rows marked with (n) show the response times normalized to the times for the unmodified scenario, that are the basis and found in rows (b) to show relative overhead. The overhead varies depending on application type and parameterizations from 1.7% for wekaj48-1e3 (the smallest dataset we used) to 73% for the most CPU intensive test 'mandelbrot50' that requires the most iterations over nested loops containing arithmetic operations. Interesting is that in almost all cases the standard deviation of the jrafed mean response times is larger than in the test cases where the unmodified environment was used.

⁹<http://www.cs.waikato.ac.nz/ml/weka>

¹⁰Intel Pentium 4, 2.6GHz running FC2 Linux and Globus 3.2.1 WS-Core in Tomcat

¹¹Intel Pentium 4, 2.4GHz, running RH9 Linux; 100Mbit Ethernet to server

¹²All experiments were repeated 130 times with the first 30 iterations discarded to allow application warm up.

4 Related Work

Pierce and Fox [PF04] describe a simple approach where legacy binaries, in their case an earthquake simulation tool, are wrapped by Java Web services and executed in forked subprocesses. They treat the wrapped application as black-box and expose web-service interfaces to allow easy integration in web portals and problem solving environments. However their approach is a largely manual technique and issues such as service discovery, service metadata, error and exception handling are ignored. In contrast our approach applies MDA-techniques for fully or semi-automatic code generation and allows extensions with non-functional properties.

M. Li et al [LQ04] describe an approach to generate Web services for non-networked high performance MPI scientific legacy codes from XML-based descriptions of the interface and platform-specific information of the legacy component. In contrast to our approach they do not use MDA techniques and do not separate platform-independent from platform-specific information in the component description therefore reducing flexibility and restricting themselves to a single platform. Moreover they do not consider non-functional properties.

Mudiam et al [MGL04] wrap Web services and commandline programs on-the-fly to facilitate their integration and so that they can be discovered and used with Jini technologies. As in our approach they rely on the Proxy and Adapter patterns for their wrappers. They use an architecture description language for describing the legacy components. Although they wrap different source components including commandline programs and Web services their solution is restricted to a single platform without incorporation of non-functional properties.

Mizuta and Huang [MH05] developed a GT3 UML profile and AndromDA cartridge that translates grid services as UML class models to generate source code and settings for GT3. They differ in their approach in that they rely on UML as modeling language and a GT3 UML-Profile to enrich the models whereas we use lighter specialized models for our problem domain. We see their approach as complementary in that we could address their UML-based input models as a target of our tools to reuse their code generation infrastructure.

Sang et al [SFKL02] focus on the migration of existing scientific codes to CORBA-based environments. High-performance legacy codes written in Fortran and C++ are wrapped to build distributed CORBA objects in form of an client/server environment. Again the major difference to our work is that we are using MDA-techniques to target several platforms and our focus on non-functional aspects.

5 Conclusions and Future Work

Distributed computing and also Grid environments have lifecycles much shorter than the lifespan of many scientific applications. The need for smooth and automatic migration of existing codes to modern environments has been a research issue for some time and

continues to be a major research interest. In our work, we address this well-known problem but from a novel perspective and for a novel problem domain. We demonstrated how we apply results and techniques from Model-driven Architecture and Code Generation field to create wrappers for scientific applications. The benefit of these wrappers is founded in that we enrich them with non-functional property extensions such as performance analysis, for instance. We presented design and implementation of our early prototype. As non-functional property we selected, in the context of this work, CPU-resource-accounting with JRAF2 and presented its integration and we performed performance experiments to investigate its effects.

Our work is in an early stage with still many open questions. As next steps we plan to add additional platforms and features which will give us feedback on our metamodels. We expect to arrive at a broadly-applicable and generic metamodel after a couple of other environments and platform features have been integrated. We have several ideas regarding additional non-functional properties for performance analysis and reporting that we are going to tackle in future work. Data management and execution parallelism are a central issue for Grid applications in general and for workflow applications in particular. We plan to extend our metamodels to capture file-staging requirements and generate the necessary code to perform file-staging, e.g. by using GridFTP, as part of the wrapper. For non-java-based applications we plan to use Scalea-G [FJP⁺05] an application-level monitoring environment for parallel and distributed applications that can provide similar run-time and post-mortem information as JRAF2 although designed for a different purpose.

References

- [AS99] J. Almond and D. Snelling. UNICORE: Uniform access to supercomputing as an element of electronic commerce. *Future Generation Computer Systems*, 15:539–548, 1999.
- [BSM⁺03] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose. *Eclipse Modeling Framework*. Addison-Wesley, August 2003.
- [C⁺04] K. Czajkowski et al. The WS-Resource Framework 1.0, May, 3 2004.
- [FJP⁺05] Thomas Fahringer, Alexandru Jugravu, Sabri Pllana, Radu Prodan, Clovis Seragiotto Junior, and Hong-Linh Truong. ASKALON: A Tool Set for Cluster and Grid Computing. *Concurrency and Computation: Practice & Experience*, 17(2-4), 2005.
- [FK99] Ian Foster and Carl Kesselman. The Globus Toolkit. In Ian Foster and Carl Kesselman, editors, *The Grid - Blueprint for a new Computing Infrastructure*, pages 259–278. Morgan Kaufmann, 1999.
- [FKNT02] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Technical report, Open Grid Service Infrastructure Working Group, Global Grid Forum, June 2002.
- [GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, October 1994.

- [HB04] J. Hulaas and W. Binder. Program Transformations for Portable CPU Accounting and Control in Java. In *Proceedings of PEPM'04 (2004 ACM SIGPLAN Symposium on Partial Evaluation & Program Manipulation)*, pages 169–177, Verona, Italy, August 24-25 2004. ACM Press.
- [JK98] H.-Arno Jacobsen and Bernd J. Kramer. A design pattern based approach to generating synchronization adapters from annotated IDL. In *Proceedings of the Automated Software Engineering Conference*, pages 63–72, Hawaii, USA, 1998. IEEE Computer Society Press.
- [LQ04] M. Li and M. Qi. Leveraging legacy codes to distributed problem-solving environments: a Web services approach. *Software: Practice and Experience*, 34(13):1297–1309, August 2004.
- [MGL04] Sudhakiran V. Mudiam, Gerald C. Gannod, and Timothy E. Lindquist. Synthesizing and Integrating Legacy Components as Services Using Adapters. *Journal of the Science of Computer Programming*, 2004.
- [MH05] Sachio Mizuta and Runhe Huang. Automation of Grid Service Code Generation with AndroMDA for GT3. In *Proceedings of 19th International Conference on Advanced Information Networking and Applications (AINA'05)*, pages 417–420, 2005.
- [MM03] Joaquin Miller and Jishnu Mukerji. MDA Guide Version 1.0.1. Technical Report omg/2003-06-01, OMG, June, 12 2003.
- [NSRC00] George R. R. Justo Nelson S. Rosa and Paulo R. F. Cunha. Incorporating Non-Functional Requirements into Software Architectures. In *In Fifth International Workshop on Formal Methods for Parallel Programming: Theory and Applications (13th IEEE International Conference on Distributed and Parallel Processing)*, Cancun, Mexico, May 2000.
- [PF04] Marlon Pierce and Geoffrey Fox. Making Scientific Applications as Web Services. *Web Computing*, January/February 2004.
- [PWWR03] Savas Parastatidis, Jim Webber, Paul Watson, and Thomas Rischbeck. A Grid Application Framework based on Web Services Specifications and Practices. Technical report, North East Regional e-Science Centre, University of Newcastle, August 12 2003.
- [SD04] Gurmeet Singh and Ewa Deelman. Montage on the Grid. Nvo technical report, NVO, April 3 2004.
- [SFKL02] J. Sang, G. Follen, C. Kim, and I. Lopez. Development of CORBA-based Engineering Applications from Legacy Fortran Programs. *Information and Software Technology*, 44(3):175–184, 2002.
- [SVHF05] Mumtaz Siddiqui, Alex Villazon, Juergen Hofer, and Thomas Fahringer. GLARE: A Grid Activity Registration, deployment and provisioning framework. In *Proceedings of the International Conference for High Performance Computing, Networking and Storage (Supercomputing 2005)*, Washington, USA, November 12-18 2005.