

# V-Grid - A Versioning Services Framework for the Grid

Jernej Kovse, Theo Härder

Department of Computer Science  
University of Kaiserslautern  
P.O. Box 3049, D-67653 Kaiserslautern, Germany  
{kovse,haerder}@informatik.uni-kl.de

**Abstract:** A large variety of emerging Computational Grid applications require versioning services to support effective management of constantly changing datasets and implementations of data processing transformations. This paper presents V-Grid, a framework for generating Grid Data Services with versioning support from UML models that contain structural description for the datasets and schema tuning information. The generated systems can be integrated using active rules to support dynamic composition of versioning services and large federated workspaces consisting of objects that reside in the individual systems.

## 1 Introduction

A *Computational Grid* is a hardware and software infrastructure providing dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities [FC98]. In scientific applications, its usage roughly follows four steps [EUD03]: (i) The user initiates a request for a computational job to the Grid and provides input data, (ii) the Grid allocates the required computational and storage resources, (iii) the Grid monitors request processing, and (iv) the user is notified by the Grid as the results of the job become available. Typical Grid applications include processing large volumes of experimental data from high-energy and nuclear physics experiments [PPD03], genomics, proteomics and molecular biology [IBM03], and earth observations (e.g. for tracking large-scale climate changes) [UNE03].

In our opinion, the current state of Grid-related research lacks a concise study of how the Grid can benefit from versioning services. Therefore, the main objective of this paper is to determine what kind of entities in the Grid require versioning services and how these services may be provided. We try to build on the existing standardization efforts including the *Open Grid Services Architecture (OGSA)* [GGF03], *Open Grid Services Infrastructure (OGSI)* [GGF03a], and *Database Access and Integration Services (DAIS)* [GGF03b].

This paper is structured as follows. In Sect. 2, we discuss the general concepts related to the so-called Grid services with particular focus on services used for data management. The section also provides an overview of work that substantiates the need for versioning for Grid applications. Sect. 3 introduces *V-Grid*, which is our framework for model-

driven development of Grid data services that support versioning. Finally, Sect. 4 presents our conclusions and outlines some ideas for the future work.

## 2 Definition of Terms and Related Work

In this section, we define the terms needed throughout the rest of the paper and give an overview of related work.

### 2.1 Grid Services

The goal of the Grid is the efficient integration of distributed computational resources through *virtualization*, i.e. a transparent access to these resources. Each resource is represented as a *Grid service*, which is a service specified using OGSi [GGF03a] extensions to WSDL. Thus, a Grid service is a Web Service conforming to a special set of conventions (i.e., interfaces) that the clients in the Grid can rely on. OGSi defines these conventions by specifying WSDL portTypes and describing the required behavior of Grid services implementing these portTypes (see [GGF03a] for a detailed overview of operations defined by the portTypes). An implementation of such a service runs on a server called *hosting environment* to serve the requests posed by other services (clients). OGSi defines mechanisms for creating, managing, and exchanging information among Grid services. In the following list, we give a brief overview of the concepts covered by OGSi.

- *Grid service lifecycle*. A client can request the creation of a Grid service instance through a *factory*, which itself is a Grid service. An instance can be terminated in two ways: In the case of *explicit destruction*, the *destroy* operation is invoked on the instance. In the *soft-state approach*, the client expresses interest in the instance for a given period of time. As this time expires (it can, however, be extended by the client), the instance will be automatically destroyed.
- *Naming*. A Grid service instance is named globally by one or more *Grid Service Handles (GSH)* in the form of a URI. In order to communicate with the instance, the client has to resolve (either by itself or by using a *handle resolver* service) a GSH to a *Grid Service Reference (GSR)*, which includes the information required for accessing the service instance over one or more protocol communication bindings (e.g. RMI/IIOP or SOAP).
- *Notification*. The OGSA notification framework allows asynchronous delivery of notifications, i.e. messages interesting for services cooperating in a given domain. Grid services that act as message senders are called *notification sources*, while services that wish to accept messages are called *notification sinks*.

### 2.2 Grid Data Services

Grid services that process large data volumes obtain this data from Grid services that provide data access and management capabilities, the so-called *Grid data services (GDSs)*.

The general requirements for facilitating data management for Grid applications by a GDS are discussed by [GGF03b], [Wa02], and [RNC+02]. These requirements, determining the core properties of a GDS, can be summarized as follows.

- *Heterogeneity transparency*: Accessing the data is independent of the implementation of the data source, e.g. a DBMS or a file system.
- *Location and name transparency*: A client is shielded from the actual location of data it accesses.
- *Distribution transparency*: A GDS integrates distributed data and allows the client to access it in a unified fashion.
- *Replication transparency*: A GDS may cache and replicate data to improve performance and availability.
- *Ownership and costing transparency*: Clients are spared from separately negotiating access authorization and costs.

A GDS should provide substantial metadata on the underlying data management system (e.g. supported query languages). However, metadata describing the *structure* (for example, relational schema or XML schema) of data stored by the GDS is no less important since it allows metadata-driven tools to discover schema information at runtime. Typically, diverse query languages (e.g., SQL, XPath) will be supported by a GDS. A GDS also supports high-performance bulk loading, streaming of query results to an external node for further processing, and a function that estimates query execution costs without actually running the query. In accordance with the OGSA notification framework described in Sect. 2.1, a GDS can act as a notification source for insert, update, deletion, query, and schema modification events. Finally, some clients will desire to access large datasets connected by relationships much like objects in an OO database. Thus, if this is a requirement, a GDS should provide object-at-a-time navigational access to its data.

### 2.3 Why Versioning Services are Required?

Processing of large amounts of data in scientific experiments requires versioning capabilities. For example, Jurisica et al. [JRG+01] describe Max, a prototype used to speed up the process of crystal growth for proteins to enable the determination of protein structure using single crystal X-ray diffraction. A robotic setup prepares and evaluates over 40 thousand crystalization experiments a day. Digital images of the crystalization are processed using the two-dimensional Fourier transform to perform automated classification of the experiment outcome. According to the authors, since the image-feature extraction algorithm is in gradual improvement to increase classification accuracy and the imaging settings may change as well, versioning of images and the processing code is required.

Holtman [Ho01] provides an overview and requirements of the data grid system used for the Compact Muon Solenoid (CMS) experiment. The prime goal of the experiment is to confirm the existence of the Higgs boson particle, which is the origin of mass. The author notes that the analysis of (pre-filtered) data from *events* (collisions of particles in the CMS detector) in the system is an iterative, collaborative process. Subsequent versions of event feature extraction and event selection functions have to be refined until their effects

are well understood. A typical job issued by a physicist will be to run the next version of the algorithm he developed to locate the Higgs events and later on, based on the output data, examine the properties of the version.

To summarize, in experimentation environments, data to be analyzed can originate from diverse sources with changing observation conditions. These conditions relate both to equipment – cameras, radiometers, spectrometers, chromatographs, which can be calibrated for various degrees of precision – as well as the observation environment, e.g. temperature, humidity, air pressure, illumination (these factors can as well be simulated). In such cases, *versioning of input data* is required. We expect that there will be a Grid service that processes this data to obtain some output data. However, different versions of the implementation of this Grid service can be available (e.g., as mentioned by [RJS01], there may be a fast version that produces only approximate results and a slow version that produces more precise results). Some of the implementation versions can be marked as stable and some can be early releases of implementations still under development. Additionally, versioning can be applied to distinguish among service implementations that perform the same data transformation but require different hosting environments. In this manner, we view *implementations themselves as versioned data* that is stored by Grid data services and can be deployed on demand. Often, transformations are chained meaning that the output data produced by one service will be used as input data for another service. Typical examples of this are data preprocessing services well known from data mining applications [HK01]: *data cleaning* (automatic dealing with missing values, e.g. by inserting global constants or calculated attribute means; dealing with noisy data, e.g. by regression), *data pre-transformations* (aggregation, generalization, normalization, or feature construction), or *data reduction* (dimensionality reduction, data compression, numerosity reduction, discretization). Thus, the main purpose of versioning services for the Grid is to allow the tracking of what *version of what input data* has been processed by a chain of *particular versions of some Grid services* to produce a *version of some output data*. Additionally, if transformation services are parameterized, we want to know what input parameters have been submitted to them to configure the transformation. Such tracking records in the Grid are called *provenance (lineage)* [ADG+03] and are very important for consistently repeating experiments used to derive some input data and later processing of this data, as well as discovering reliable data sources and useful calibrations of instruments. For example, if a smaller sample produces interesting results, we may choose to repeat the experiment and invest more processing resources to run the transformation on a larger dataset.

Raman et al. [RNC+02] also mention the need for special *collaboration services* in data-intensive Grid applications, which will facilitate sharing of data between users at different sites. These services encompass checkout/checkin functionality and annotation of objects in Grid data sources with versioning information.

Sometimes it is easier for Grid users and applications to view their data in a version-free manner, although data is versioned. This makes interactive manipulation of data easier and implementations of algorithms that manipulate the data less verbose. The first approach to this problem is to support a version of data in the GDS to be marked as the default version meaning that it will be returned in case we do not exactly specify what

version we want. Bernstein et al. [BBC+99] refer to this behavior as *pinning*. Another solution is to return the version determined by a rule that chooses the version from the version graph according to some properties (the most common rule is to return the latest version from the graph). Another well-accepted solution is to support *workspaces* (*configurations*), where each workspace is allowed to attach no more than a single version of data. Thus, once a client chooses a workspace to work with, it can manipulate the objects within the workspace without explicitly referring to versions.

Versioning can affect replication policies in the Grid. Some versions can be marked as *read only* meaning that they can be replicated without having to assure change propagation back to the master copy. This will always be the case with versions we have *frozen* (made *immutable*) to prevent further changes to the data. As mentioned by Guy et al. [GKL+02], special policies are needed to determine how a GDS with an installed replica behaves on changes committed either to the master or to other replicas. For example, creating a successor version to the master may automatically replace the existing replicas with the new version. An alternative is to install new versions upon request.

### 3 V-Grid

The purpose of our *V-Grid framework* is two-fold:

- First, V-Grid acts as a *generation platform*. A user that requires a GDS with versioning support has to provide a model for the datasets and define versioning semantics that should be used for the data (e.g. what types of datasets are versioned, how do operations on this datasets like *createSuccessor*, *copy*, or *freeze* propagate among datasets). The V-Grid generator takes the model and generates a complete GDS implementation, which we call *V-GDS* (a GDS with versioning support). The generated V-GDS is a complete, running J2EE application with a corresponding database schema, middleware enterprise components and its operations exposed as Web services so that the V-GDS can be accessed by other entities in the Grid. A V-GDS can be deployed automatically on a selected remote application server from a server pool.
- Second, V-Grid acts as an *integration platform* for generated V-GDS systems. This integration platform allows an integration of generated V-GDS systems by applying *rule-based service composition*. Such an integration platform is needed since the requirements for storing data in complex Grid applications will rarely remain static: Often, additional datasets and transformation implementations that require storage and versioning services will emerge. This implicates the need for large federated workspaces with datasets stored in diverse participating V-GDS systems. Active rules are used to dynamically compose versioning services across these systems and assure referential integrity for the federated workspaces.

#### 3.1 V-Grid Generation Platform

The purpose of the V-Grid generation platform is to support the generation of V-GDS systems on a basis of formal system specifications provided in the UML language. In this

sense, the platform is motivated by the OMG's *Model Driven Architecture (MDA)* [OMG01]. MDA is an approach to software system development that separates a formal specification of a system from the implementation of the system on a particular platform. It is desired that formal specifications that capture both static and dynamic (behavioral) properties of a system are provided using existing OMG's modeling languages (i.e., UML and CWM). Given a formal specification in form of a model, a *generator* will be used to map the model to the system implementation that executes on a particular platform.

The V-Grid generation platform can be seen as a *product line* [CN02] for V-GDS systems. The product line is implemented as a *system family*, where different V-GDS systems that can be generated using the generation platform are seen as members of this family. All V-GDS systems share a certain amount of base functionality: They all support storing datasets in a relational database, provide versioning and workspace management services for these datasets, and enable set-oriented and navigational access. However, each member is still a unique system, since it possesses a unique relational schema for its datasets and may have the semantics of its versioning services optimized for its clients. Thus, the member is specified in two consecutive steps, *type definition* and *schema tuning*.

**Type definition.** V-Grid adopts the object-oriented approach described by Bernstein [Be98] to representing versioned data. Classifications of data stored by a V-GDS are represented as object types and modeled as *UML classes* using a UML class diagram. Properties of datasets are represented as attributes of Java data types. A mapping of these types to the type model of the target DBMS (e.g. DB2, Oracle) can be defined to customize the output DBMS schema, where large data sequences are typically represented as byte arrays in Java and BLOBs in the DBMS. Within a V-GDS, semantic relationships among data (objects) may exist. For example, a relationship may be used to connect the *source code of a transformation algorithm* (represented as the first object) to the corresponding *executable* (represented as the second object); similarly, a relationship may connect the *calibration parameters* of an instrument (first object) to the *dataset delivered in the experiment* (second object); finally, each applied transformation will typically result in a relationship among the *input dataset* (first object) and the *output dataset* (second object). Each relationship is an instance of a relationship type that exists among two UML classes and is defined as a *UML association*. UML class diagrams for this step can be developed using any existing UML modeling tool, such as Rational Rose or Genteware Poseidon.

**Schema tuning.** Type definitions defined in the previous step can support versioning in a variety of ways. For this reason, we allow the schema represented as the UML class diagram to be fine-tuned (optimized for convenient use as well as performance). This is possible by *branding* UML classes and associations by stereotypes and choosing tag values for tag definitions provided by these stereotypes. *Stereotypes, constraints, tag definitions* and *tag values* constitute a built-in extension mechanism of the UML language and are defined in form of *UML profiles*. Again, since the majority of UML modeling tools support profiles, the schema tuning step can be fully accomplished using these tools. Branding a UML class or associations with a stereotype and choosing tag values implicates that the corresponding object or relationship type in the V-GDS will possess special proper-

ties. Stereotypes and tag values are used to drive the V-Grid generator to consistently include these properties in the implementation of the V-GDS. The properties that can be defined are classified as follows.

- *Variability in object management.* As noted by Rumbaugh [Ru88] and Zhang et al. [ZRH01], relationships are a convenient spot for capturing propagation behavior of operations on objects. In V-Grid, tag values on each end of a relationship type define whether basic object management operations on datasets (objects), namely *create* and *initialize*, *copy*, and *delete* are executed in a *propagated* or *isolated* fashion. For example, copying an existing input dataset may cause the output dataset associated with it to be copied as well.
- *Variability in relationship management.* These properties allow the users to define whether a relationship can be created in case one or both objects it connects do not yet exist. Similarly, it is possible to specify whether manual deletion of relationships, which will delete a relationship but not the objects the relationship associates, is permitted. Finally, connecting or disconnecting a relationship end to a dataset version that has already been frozen can be allowed or prevented.
- *Variability in version management.* It is not a requirement that all dataset types defined in the schema support versioning. Versioning of some types may be prevented, both for simplicity of use and storage optimizations. As a consequence, these types will always support merely a single version of its instances and will not define the *createSuccessor* operation and operations used to traverse the version graph (*getRoot*, *getSuccessors*, *getPredecessors*, and *getAlternatives*) that are normally supported by types that support versioning. Similarly as it is the case with object management operations, *createSuccessor* and *freeze* operations can be executed in a propagated or isolated fashion across relationships. For example, freezing a given dataset can also freeze the associated datasets. Another versioning feature that can be selected or omitted for relationship ends that connect to versioned datasets are *floating relationship ends*, which are used in the following way: In case a dataset A is versioned, it sometimes does not suffice for a dataset B that is related to A to merely identify A when navigating across the relationship. This is the fact since B does not necessarily connect to all versions of A, but rather to a user-managed subset of versions of A, which we call a *candidate version collection*. In case a floating relationship end is chosen for a given relationship type, the V-GDS will provide operations for manipulating candidate version collections, pinning and unpinning a certain version in the collection (in case the client does not want to review all versions in the collection, the pinned version will be returned by the V-GDS automatically), or selecting a version on the basis of some predefined rule, the most common case being to return the latest version from the collection. Again, for simplicity of use as well as performance and storage optimizations, the use of a floating relationship end can be omitted.
- *Variability in workspace management.* These properties allow the user to define whether the *attach* operation on an object that makes this object a component in a given workspace is propagated across existing relationships from this object. In a similar fashion, the *detach* operation can also be propagated across relationships of a given type. Additionally, users can define whether objects of a given type should be exclusively owned by workspaces of a specific type. Alternatively, objects may be

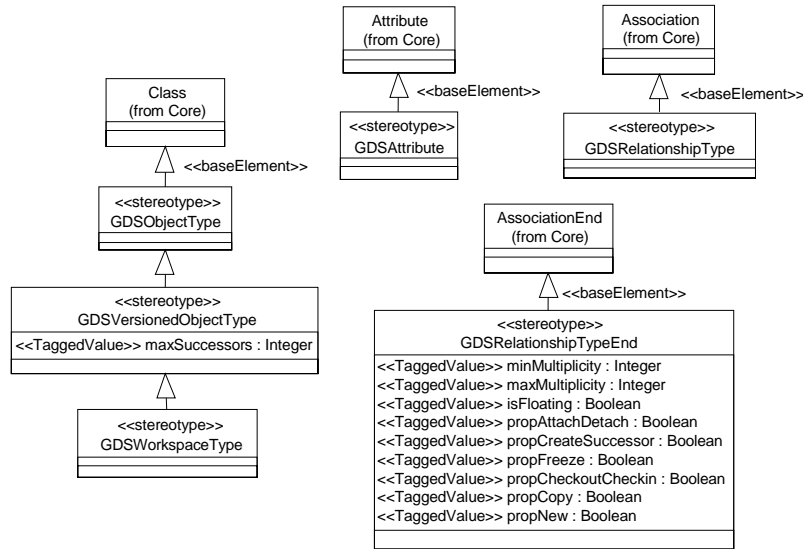


Fig. 1: UML profile for V-GDS type definition and schema tuning

shared among workspaces. Finally, the invocation of the *createSuccessor* operation on an object within a workspace may replace the existing version in the workspace, or create a new version of the entire workspace.

- *Variability in checkout/checkin*. The *checkout* and *checkin* operations, which are used for setting and releasing long-term locks on repository objects, can be propagated across relationships of a given type or executed in an isolated fashion.

Fig. 1 illustrates a simplified version of our UML profile for V-GDS type definition and schema tuning that supports the described variation points.

### 3.2 V-Grid Generator

As the schema tuning step is completed, the UML class diagram is exported from the modeling tool as an XMI document. This document serves as an input to the V-Grid generator, which will examine the type definitions and user decisions on variable features. The main advantage of the V-Grid's generative approach is that these decisions become directly hardwired into the implementation of the V-GDS. For example, it would equally be possible to provide *generic* database tables (i.e., tables that would be present in every single database schema for a V-GDS, irrespective of type definitions) for maintaining information on candidate version collections and currently pinned versions. However, this solution requires separate access to the generic tables each time the versions in the collection are accessed by the application. In our approach, the generator will normalize the schema to support direct joins in the queries that access candidate version collections. In a similar way, it would equally be possible to define operation propagation rules that



would determine how operations are propagated across relationship types using a separate base of ECA rules (see [BD94] for a detailed description of this *notification approach* in the context of repository systems). However, this requires the V-GDS to act as a rule interpreter, decreasing its performance. In our case, all *operation propagation rules* can be automatically derived from the tag values selected in the schema tuning step. For this reason, the generator can integrate them directly in the implementation of the V-GDS, eliminating the need for run-time interpretation.

The V-Grid generator adopts a *template-based code generation* approach proposed by Sturm et al. [SVB02]. As mentioned by the authors, similar template-based approaches have become popular for the dynamic creation of HTML pages. In the proposed approach, templates act as skeletons for generated code artifacts and are filled with information extracted from the UML model in the generation process.

Following the idea presented by [SVB02], V-Grid templates have been implemented using the open source project Velocity [Ap03], which comes with a language for defining templates, called the Velocity Template Language (VTL) and a Java-based template engine. The purpose of the engine is to merge a template written in the VTL with a *context*. As described by [Ap03], the context is basically a hash table (a set of key-value pairs) that makes Java objects of various types (*values*) accessible from within a template using *keys*. As most template languages, including XSLT, VTL supports *looping through a list of objects* (which is very convenient in case a certain code segment in the generated code is sequentially repeated for each of the objects in the list) and *conditional statements*. In our case, we fill the Java objects that act as context values with information obtained by parsing the XMI document that corresponds to the UML model containing type and relationship definitions for the V-GDS as well as schema tuning decisions. A large set of VTL templates is used in the V-Grid generation approach, where each of the templates typically accesses only a part of the information from the UML model. This information is fetched from the UML model using the so-called *prepared elements* introduced by [SVB02]. For example, a single prepared element provides information (as strings) on the name of the class that represents an object type in the UML model, stereotype the class has been branded with, attribute names, names of relationships the class participates in, corresponding multiplicities, etc. Without a prepared element, this information would have to be gathered by the VTL template from many fine-grained objects that correspond to UML model elements, which would make the template excessively verbose. Fig. 2 illustrates a part of a VTL template used to generate the *createSuccessor* method for a versionable object type. The `#foreach` directive is used for looping, while the `$` characters denotes references to Java objects (*values*) in the context. An example for a prepared element for easy access (methods for returning values for visibility, stereotype, etc.) to UML model elements that represent classes is given in Fig. 3.

### 3.3 Generated Artifacts

In the current state of our project, the VTL templates used by the generator produce Java code for the J2EE platform. However, a similar generation approach (with modified tem-

```

public abstract class $class.getName()Bean implements javax.ejb.EntityBean {
...
// creates a new version of the current object
public $class.getName()Local createSuccessor() throws Exception {
    if (!getFrozen())
        throw(new javax.ejb.EJBException("object is not frozen!"));
    $class.getName()Local newCopy = null;
    try {
        newCopy = get$class.getName()Home().create(getObjId());
        newCopy.setParent(($class.getName()Local)myEntityCtx.getEJBLocalObject());
#foreach( $attribute in $class.getAttributes() )
        newCopy.set$attribute.getNameUpperCase()(get$attribute.getNameUpperCase());
#end
    }
    catch (Exception ex) { // shouldn't happen
        throw(new javax.ejb.EJBException("couldn't create copy"));
    }
    return newCopy;
}
...
}

```

Fig. 2: Excerpt from a VTL template for generating Java code for versionable objects

```

public class PreparedClassData {
...
public String getVisibility() {
    return mModelClass.getVisibility().toString();
}
public String getStereotype() {
    Collection stereotypes = mModelClass.getStereotype();
    if (stereotypes.size() > 0) {
        MStereotype stereotype = (MStereotype)stereotypes.iterator().next();
        return stereotype.getName();
    } else return "";
}
...
}

```

Fig. 3: Excerpt from a prepared element for accessing Class UML model elements

plates) can be applied to produce code for other execution platforms. Our generated V-GDS systems follow the idea of thick middle-tier applications with most of the application logic (versioning operations with hardwired operation propagation rules, e.g., *create-Successor*, *freeze*, and others, as well as retrieval of objects within a workspace) executed in the application server. The major advantages of this approach with respect to Grid applications are the following.

- *Caching*. As mentioned by [RNC+02], caching functionality is important in Grid applications for replicating an entire dataset or its subset for fast access by the clients and maintaining its state synchronized with the original in the information tier (i.e., the Grid data source). Once a dataset is derived in some experiment and written to the V-GDS, it will typically have many read-only accesses by the clients to be used as input in the transformations. Since read-only accesses do not invalidate the contents of the cache, there is no need for synchronization, which brings significant performance advantages. In general, J2EE application servers implicitly support data caching at the persistence layer using Entity EJBs.
- *Scalability*. Multithreading, data source connection pooling, and instance pooling at the persistence layer increase scalability within a single instance of an application

server. Certain implementations of application servers (e.g. WebSphere Application Server) increasingly support techniques such as vertical and horizontal server instance cloning combined with centralized workload management [IBM00].

- *Remote deployment.* Remote deployment of a generated V-GDS to a server from a server pool is made possible by the so-called *deployment managers* that are part of the application server and can occur without human intervention.
- *Set-oriented access.* Although the persistence layer presents the clients with an object-oriented view to the datasets (which directly supports object-at-a-time navigational access we mention in Sect. 2.2), this does not necessarily exclude the set-oriented access. *Select methods* of Entity EJBs can be specified using special EJB QL query language [Sun02] for set oriented access over the *abstract schema* for the datasets.

The following sections provide a detailed overview of artifacts produced by the V-Grid generator.

**Database schema.** Each object type from the UML model is mapped to an own database table with columns that correspond to the type's attributes. However, the VTL templates assure that additional constructs are added to the tables depending on how the schema has been tuned. For example, in the case of a versionable object type, a table will obtain an `objId` column, which represents the identity for the object, a `verId` column, used to identify diverse versions of an object, and a `globId` column, which stores V-GDS-wide unique identifiers comprised of `objIds` and `verIds`. Moreover, we need a `predecessorId`, which is used for linking a version to its predecessor version to allow traversal of the version graph, a `frozen` column to denote whether a version has already been frozen as well as a `checkout` column referencing the workspace the version has currently been checked out to. Foreign keys are added to diverse tables depending on where floating relationship ends are applied.

**Persistence layer.** Entity EJBs in the persistence layer are used to abstract the control layer from fine-grained SQL access to the V-GDS data source by automatic synchronization of updates to the data source and data caching. V-Grid generates an Entity EJB for each object type definition from the UML model that mirrors both user-defined attributes for the database tables as well as attributes added due to schema tuning.

**Control layer.** Session EJBs in the control layer act as a *business facade* [ACM01] for the persistence layer. They provide the users with a coarse-grained interface to versioning operations and assure that versioning operations are carried out as required in the schema tuning step. For example, operations like *createSuccessor* and *freeze* propagate across the relationships, where desired; specified version selection rules (e.g. selection of the latest version) are applied when the version is to be automatically selected from a candidate version collection. Each client communicates with the control layer by first retrieving a *V-GDS session*, which is a stateful representative of the client on the side of the V-GDS and is typically used to hold the identities of the currently selected workspace and the currently running ACID transaction. This makes the communication with the client less verbose, since these state values do not have to be passed in each client call. Since fine-grained remote access to objects results in high communication costs between the client and the V-GDS, disjoint schema partitions of coarse-grained Java value objects can be

specified in the schema tuning step. These value objects hold data from multiple entities, assure that only user-defined attributes (but not the V-GDS managed attributes like *verId*, *frozen*, or references among entities) are updatable, and provide the client with an object-at-a-time navigational access to a part of the entire object graph. The control layer assembles value objects on demand at each client call and disassembles them (in the case of updates made by the client) to map the modified data back to the persistence layer.

**Web services layer.** Since the V-GDS does not make any assumptions about the execution platform of the client, V-Grid generates Web services endpoints that support SOAP messaging between the client and the generated V-GDS. The endpoints implement the port-Types required by the GDS specification document [GGF03b] as well as provide additional operations specific to the data types specified in the UML model and the tuned schema. Additionally, a WSDL document is generated for each V-GDS.

### 3.4 Accessing Generated V-GDS Systems

There are three main styles of how the generated V-GDS can be used by the client.

- *Direct access and scripting.* In this approach, clients that rely on the generated Web services interfaces are developed to communicate with the V-GDS using these interfaces. For this reason, the client calls are dependent on the object types defined by the UML model. Such a client will typically fetch a version of an object, perform a dedicated transformation and store transformation results to the same or another V-GDS.
- *Generic (meta-data driven) access.* Developing clients that are bound to operation signatures of a generated V-GDS is not efficient, since a client cannot be reused for performing a similar task on a V-GDS with a different information model. The solution is to make the running client access a V-GDS *generically*, i.e., in two steps: First, the client retrieves the entire UML model including the schema tuning information. Based on this model, the client itself at runtime assembles the names of operations it wants to invoke.
- *Interactive access.* Sometimes, V-GDS users will want to explore and possibly update the contents of a V-GDS in an interactive way, i.e. without using a special client. For this purpose, the V-Grid generator produces and deploys JSP pages that allow interactive browsing of V-GDS contents and invocation of version management operations provided by the V-GDS.

### 3.5 V-Grid Integration Platform

There is a wide variety of approaches that successfully address the execution of distributed workflows using rules. For example, the WfMS in the WIDE project [CGS97] uses ECA rules to support exceptions and asynchronous behavior during the execution of a distributed workflow instance. The V-Grid integration platform adopts the rule-based approach to service composition proposed by the *DY<sub>flow</sub>* framework [ZBL+03]. *DY<sub>flow</sub>* supports three different types of service composition rules (see [ZBL+03] for a detailed syntax for rule definitions).

- *Backward-chain rules.* These rules define preconditions (i.e., data and flow constraints) for executing a task. For example, we may want to require that each dataset in a workspace is frozen before the entire workspace is replicated to another V-GDS.
- *Forward-chain rules.* These rules are defined as ECA rules and specify tasks (i.e., actions) that need to be carried out as a consequence of executing a given task. The execution of an action may depend on the condition part of the rule. For example, we may want to create a successor to a version in some V-GDS as soon as a successor to a related version in another V-GDS has been created.
- *Data-flow rules.* These rules specify data flows among tasks. For example, they can be used to automate transformation tasks for new versions: As soon as a new version of a dataset appears, it will automatically serve as an input for a Grid service that performs a selected transformation.

Unlike rules within a single generated V-GDS, which are hardwired by the V-Grid generator into the implementation code to increase performance, the composition rules can be added to the V-Grid integration platform dynamically as new V-GDS systems appear. We alleviate the definition of these rules to the users by parsing the WSDL definitions of each generated V-GDS to automatically identify signatures of operations used afterwards in the definitions of rules. Transactional execution of rules that involve many V-GDS systems is enabled by the two-phase-commit protocol supported by each participating system.

However, the V-Grid integration platform does not serve merely as a rule processing framework. Federation of multiple V-GDS causes the need to support large federated workspaces that span across objects from different V-GDS systems. Unlike local workspaces in each generated V-GDS that use highly specialized (generated) schemas, the data required for the integration (i.e. *logical references* between the workspace and the objects it contains) is stored by the integration platform using a *generic schema* that does not have to be altered as new workspaces are defined. For this reason, the access to the federated workspace is always generic (meta-data driven). A federated workspace itself can participate in the defined service composition rules. For example, we may define a rule that the *createSuccessor* operation on an object that is part of a federated workspace should create a successor to the entire federated workspace. Rules also apply for assuring global integrity constraints. For example, deletion of an object that is part of a federated workspace should delete a logical reference to this object from the workspace.

## 4 Conclusion and Future Work

This paper presented our V-Grid framework, which is used for generating Grid Data Services with tunable versioning support from UML models. Using a dedicated UML domain profile and a template-based generation approach, we are capable of generating complete application code for the J2EE platform with operations exposed as Web services. The V-GDS systems obtained in this way can be automatically deployed on application servers from a server pool and integrated using active rules. The integration

approach supports the dynamic composition of versioning services and use of federated workspaces that contain objects from diverse V-GDS systems.

In the course of our future work, we attempt to:

- Explore the possibility of supporting the *merge* operation, used for reuniting branches in the versioning graph. The semantics of *merge* is more complex than that of other versioning operations, since it requires detailed knowledge of the structure of each object attribute to decide on the priority of one version over another. It is our assumption that reconciliation among versions can be specified by using a dedicated set of constructs at the UML level, which would allow the operation to be fully generated.
- At this moment, the generation process is initiated by the user through an interactive interface to the V-Grid generator that accepts the UML model in the XMI format. Nevertheless, in accordance with the core idea of the Grid, we also expose the generator itself as a Grid service. We will try to explore to what extent a full programmatic invocation of the generation process and deployment of a generated V-GDS system may be interesting to Grid applications.

## References

- [ACM01] Alur, M.; Crupi, J.; Malks, D.: Core J2EE Patterns. Prentice Hall, 2001.
- [ADG+03] Atkinson, M.P.; Dialani, V.; Guy, L.; Narang, I.; Paton, N.W.; Pearson, O.; Storey, T.; Watson, P.: Grid Data Services and Integration: Requirements and Functionalities, DAIS-WG memo. Available from: <http://www.cs.man.ac.uk/grid-db/>
- [Ap03] The Apache Jakarta Project: Velocity. Available as: <http://jakarta.apache.org/velocity/>
- [BBC+99] Bernstein, P.A.; Bergstraesser, T.; Carlson, J.; Pal, S.; Sanders, P.; Shutt, D.: Microsoft Repository Version 2 and the Open Information Model. In: Information Systems 24(2):1999, pp. 71-98.
- [BD94] Bernstein, P.A.; Dayal, U.: An Overview of Repository Technology, in: Proc. VLDB 1994, Santiago de Chile, Sept. 1994, pp. 705-713.
- [Be98] Bernstein, P.A.: Repositories and Object-Oriented Databases. In: ACM SIGMOD Record 27:1, 1998, pp. 34-46.
- [CN02] Clements, P.; Northrop, L.: Software Product Lines. Addison-Wesley, 2002.
- [CGS97] Ceri, S.; Grefen, P.; Sanchez, G.: WIDE: A Distributed Architecture for Workflow Management. In: Proc. RIDE 1997, Birmingham, April 1997.
- [EUD03] EU DataGrid Project. Available as: <http://eu-datagrid.web.cern.ch/eu-datagrid/>
- [FC98] Foster, I.; Kesselman, C.: The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann, 1998.
- [GGF03] Global Grid Forum Open Grid Services Architecture Working Group (OGSA-WG): The Open Grid Services Architecture (OGSA) Platform, Feb. 2003. Available from: <http://www.gridforum.org/ogsa-wg/>
- [GGF03a] Global Grid Forum Open Grid Services Infrastructure Working Group (OGSI-WG): Open Grid Services Infrastructure (OGSI), Version 1.0, draft, Apr. 2003. Available from: <http://www.gridforum.org/ogsi-wg/>

- [GGF03b] Global Grid Forum Database Access and Integration Services Working Group (DAIS WG): Grid Database Service Specification, Feb. 2003. Available from: <http://www.cs.man.ac.uk/grid-db/>
- [GKL+02] Guy, L.; Kunszt, P.; Laure, E.; Stockinger, H.; Stockinger, K.: Replica Management in Data Grids, Global Grid Forum 5, July 2002. Available from: <http://www.isi.edu/~annc/gridforum/papers.html>
- [HK01] Hai, J.; Kamber, M.: Data Mining: Concepts and Techniques. Morgan Kaufmann, 2001.
- [Ho01] Holtman, K.: CMS Data Grid System, Overview and Requirements, CMS Note 2001/037. CMS CERN. Available from: <http://kholtman.home.cern.ch/kholtman/>
- [IBM00] IBM Corp.: WebSphere Scalability: WLM and Clustering, IBM Redbook, 2000, available from: <http://ibm.com/redbooks/>
- [IBM03] IBM Corp.: IBM Grid Offering for Information Accessibility: Life Sciences. Available as: <http://ibm.com/grid/>
- [JRG+01] Jurisica, I.; Rogers, P.; Glasgow, J.I.; Fortier, S.; Luft, J.R.; Wolfley, J.R.; Bianca, M.A.; Weeks, D.R.; DeTitta, G.T.: Intelligent Decision Support for Protein Crystal Growth. In: IBM Systems Journal, 40(2), 2001, pp. 394-409.
- [OMG01] Object Management Group Architecture Board ORMSC. Model Driven Architecture (MDA), OMG document ormsc/2001-07-01.
- [PPD03] Particle Physics Data Grid (PPDG). Available as: <http://www.ppdg.net/>
- [RNC+02] Raman, V.; Narang, I.; Crone, C.; Haas, L.; Malaika, S.; Mukai, T.; Wolfson, D.; Baru, C.: Data Access and Management Services on the Grid. Available from: <http://www.cs.man.ac.uk/grid-db/>
- [RJS01] De Roure, D.; Jennings, N.; Shadbolt, N.: Research Agenda for the Semantic Grid: A Future e-Science Infrastructure, EPSRC/DTI Report. Available from: <http://www.semanticgrid.org/>
- [Ru88] Rumbaugh, J.E.: Controlling Propagation of Operations using Attributes on Relations. In: Proc. OOPSLA'88, San Diego, Sept. 1988, pp. 285-296.
- [SVB02] Sturm, T.; von Voss, J.; Boger, M.: Generating Code from UML with Velocity Templates. In Proc. UML 2002, Dresden, Sept. 2002, pp. 150-161.
- [Sun02] Sun Microsystems: Enterprise JavaBeans Specification, Version 2.1, August 2002.
- [UNE03] United Nations Environment Programme. Division of Early Warning and Assessment (DEWA). Available as: <http://www.grid.unep.ch/>
- [Wa02] Watson, P.: Databases and the Grid. UK e-science Tech. Report UKeS-2002-01. Available from: <http://www.cs.man.ac.uk/grid-db/>
- [ZBL+03] Zeng, L.; Benatallah, B.; Lei, H.; Ngu, A.; Flaxer, D.; Chang, H.: Flexible Composition of Enterprise Web Services. In: Int. Journal of Electronic Commerce and Business Media (2003). Available from: <http://www.cs.swt.edu/~hn12/papers/>
- [ZRH01] Zhang, N.; Ritter, N.; Härder, T.: Enriched Relationship Processing in Object-Relational Database Management Systems. In: Proc. CODAS'01, Beijing, Apr. 2001, pp. 53-62.