Towards Generating Object-Relational Software Engineering Repositories[1]

W. Mahnke, N. Ritter, H.-P. Steiert

Department of Computer Science
University of Kaiserslautern
P O Box 3049, 67653 Kaiserslautern, Germany
e-mail: {mahnke/ritter/steiert}@informatik.uni-kl.de

Abstract

Nowadays the complexity of design processes, no matter which design domain (CAD, software engineering, etc.) they belong to, requires system support by means of so-called repositories. Repositories help managing design artifacts by offering adequate storage and manipulation services. One among several important features of a repository is version management. Current repository technology lacks in adequately exploiting database technology and in being adaptable to special application needs, e. g. support of application-specific notions of versioning. For that reason, we propose new repository technology, which is not completely generic (as current repositories are), but exploits generic methods for generating tailored repository managers. Furthermore, we show that new, object-relational database technology is extremely beneficial for that purpose.

Keywords: Repositories, Object-Relational Database Systems, Software Engineering, Versioning, Reuse, Generic Methods.

1 Introduction

For years the term *repository* has been used in a restricted manner, since it only addressed metadata management in the context of database management systems. Nowadays it is used in a much broader sense and covers multiple services supporting design applications. It is not the goal of this paper to redefine the functional requirements a system has to provide in order to deserve the name repository, but to point out a better, i. e. more flexible, way to make well customized repositories available for users. To reach this goal, we propose a framework approach, allowing users to reuse predefined specifications of repositories services and adapt them to their special needs. Thus, we want to generate specific repository services from predefined service elements and amending user specifications. Before introducing our approach (in the following sections) we first want to recall basic services a repository manager has to provide and briefly discuss the technology a repository manager may be based on.

Frameworks and Repositories

In the literature, two kinds of technology can be found aiming at an adequate support of design applications: frameworks [21, 7, 26] and repositories [2, 3, 25]. Although the goals of both are pretty close, we think, there are some differences between the underlying approaches. Frameworks focus on providing basic services, which may be adapted to special application needs, and, thus, help to provide corresponding design environments capable to support the (special) design application. Repositories or, more pre-

1. This work has been supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the Sonderforschungsbereich (SFB) 501 “Development of Large Systems with Generic Methods”.

cisely, repository managers, on the other hand, emphasize the data control aspect of a certain class of design applications, e. g. software development applications, and offer predefined, generic services. We think, the two approaches may be integrated in a way beneficial for both, system providers and system users. In simple words: we want to provide a framework allowing to *generate* repository managers.

Repository Manager Services

In [2] the term ***repository*** is defined as *a shared database of information about engineering artifacts*. Thus, *a common repository allows* (design) *tools to share information so they can work together.* A corresponding ***repository manager*** *provides services for modeling, retrieving, and managing objects in a repository*. For that purpose, a repository manager has to provide the *standard amenities of a DBMS* (*data model, queries, views, integrity control, access control, and transactions*) as well as some value-added services [2]: *checkout/checkin, version control, configuration control, notification, context management and workflow control*. Since these terms (as well as their respective meaning) are widely known, we do not detail them.

Database Support for Repositories

In [2] the exploitation of a database management systems is favored instead of building repository manager services on top of a file system. Many current repository managers [4], as CMS, SCCS, RCS, and MMS, are based upon a file system. Advantages of these systems are portability and performance. Nevertheless, it is reasonable to exploit the benefits of DBMS, as for example transactions, integrity maintenance and queries. Therefore, some repository managers store descriptions of objects within a database system and leave the objects themselves be managed by the file system. This causes the well-known problems of keeping descriptions and objects consistent and controlling access to objects, which are not under control of the database system. These problems, in turn, may be solved by using a DBMS for managing all the data. Current object-oriented DBMS (OODBMS, [22]) offer some support for complex objects and user-defined data types, but as stated in [2], they often provide limited transaction facilities, limited support for queries, views, and integrity maintenance. These key features of relational DBMS (RDBMS) are now integrated with the advantageous features of OODBMS within so-called object-relational DBMS (ORDBMS, [24, 11]), which are currently considered to be the most successful trend in DBMS development (see the evolving SQL3 standard [8]). Thus, ORDBMS seem to be a good foundation for realizing repository managers. As far as we know, there are currently no repository manager approaches which are based on ORDBMS technology. We will argue that ORDBMS are well suited for that purpose, especially because of enriched modeling concepts and the extensibility property [13], i. e. the possibility of defining user-defined data types (UDT) and functions (UDF). For these reasons, our approach is based on object-relational database technology, as we will especially see in Sect. 4.

Genericity

Usually repository manager services are as generic as database system services are. Let us consider versioning facilities as an example for generic services. Usually a repository manager implements a given versioning model offering a predefined set of data structures and generic operations. As already discussed in [10] there are very many facets, versioning models may differ in. As we think, many of the different concepts which lead to (lots of) different version models are very application-specific. Consequently, a generic version model cannot support *all* applications properly, but serves some more, some less appropriately. Our goal is to be able to support all applications by providing basic versioning facilities which may be refined and which are the foundation for generating application-specific functionality. We think that this approach is not only suited w.r.t. versioning, but also w.r.t. other fields of design domains, e. g. designflow management. Therefore, our repository managers are more generated than generic. As far as we know, our approach is the first one generating extensions of object-relational database systems, which, in turn, implement repository manager functionality.

Overview of this paper

Sect. 2 will give an overview of our approach, which is called the SERUM approach[2]. SERUM can be considered to be an infrastructure for tailoring repository managers to special application needs by exploiting generic methods. Sect. 3 and Sect. 4 will consider versioning, which is one of the key issues of repository management, as an example helping to discuss the generic approach, SERUM is based on. For that purpose, Sect. 3 outlines a basic versioning framework and how it can be adapted to reflect a dedicated version model; Sect. 4 explains how an adapted versioning framework can be used to generate repository manager services and how this process can benefit from ORDBMS technology. Sect. 5 concludes the paper.

2 Overview of SERUM

Obviously, in large product development projects a shared database should be used in order to support cooperation and reuse of design. Usually, a repository is more appropriate than a pure DBMS, because repositories enhance database systems by value-added services. The requirements on repository services vary w. r. t. different ***domains***. Our domain is *computer aided software engineering*, or, more exactly, *development of large (software) systems with generic methods*. This domain spans different ***domain sections***, as for example *management of versioned product data* or *designflow management*. While the former requires facilities for versioning complex structured objects, the latter may utilize active repository services. Hence, each domain section has its own requirements pertaining dedicated repository services. In order to cope with the needs of the different domain sections, we want to provide a *framework for building customized repository managers* instead of a single, universally applicable repository manager. Since SERUM is a framework and not a “stand-alone” system, it offers a set of methods, tools, techniques and “half-fabricated” components helping to create customized repository managers. Thus, the *domain repository* is established by a set of repository managers, one for each domain section. All repository managers are based upon the same (logically) centralized DBMS allowing for data sharing among domain sections.

In Fig. 1 a graphical illustration of SERUM and the steps to be executed for customizing a repository manager is given. If the ***repository designer***[3] wants a new repository man-

2. SERUM stands for: Software Engineering Repositories using UML.

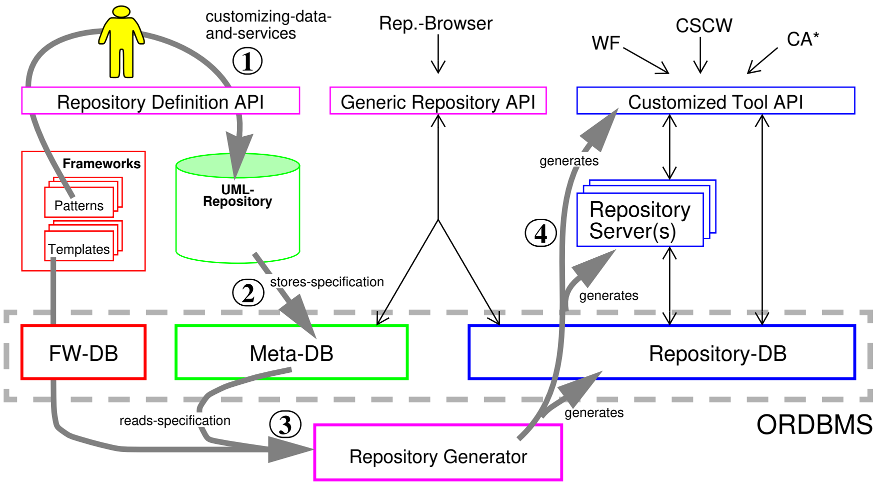

Fig 1: SERUM-Overview

ager to be generated or an existing one to be modified, he first has to choose from the set of half-fabricated components, associated with the particular domain section he has in mind. We call such a half-fabricated component a ***framework*** (see Sect. 2.3). The repository designer has to customize the framework by adapting, refining, completing and specializing the chosen components (1). The customizing process takes advantage of object-oriented concepts, as, for example, subclassing (specialization), overloading, late binding of interfaces. Several of the *reuse* techniques identified in [12] are exploited. The result of the customization process is an *UML*[4] *specification of the new repository manager*, which is stored in the SERUM meta-database (2). This UML-specification is used as input for the SERUM ***repository generator*** (3), which generates the ***repository database schema***, the ***customized tool API*** and (several) ***repository servers*** (4). All these (generated) value-added data management services together establish the new repository manager. Because we are using an ORDBMS, the *repository database schema* not only consists of tables but also includes UDTs and UDFs. Generated *tool API functions* must meet the requirements of the tools associated with the domain section and allow for adequate access to the services of the (generated) repository manager. Additional functionality, which cannot (or should not) be implemented at the API level or the database server level, must be realized by specialized *repository servers*.

In the following, we will have a closer look at the aspects and components relevant for generating repository managers.

2.1 Interfaces

The SERUM system provides a general API for the customization process (*repository definition API, RDAPI*) and a generic API for browsing metadata as well as repository data (*generic repository API, GRAPI*). The latter is needed to provide access to reposi-

3. The repository designer is the person specifying application-specific semantics needed by SERUM to generate repository manager services.
4. Unified modeling language [18, 19].

tory data for common tools, like browsers, which are not written for a dedicated repository model. Additionally, the (generated) repository managers have to come along with customized tool APIs allowing for an efficient access to the value-added data management services (*customized tool API, CTAPI*).

2.2 SERUM Metamodel

As a metamodel for components and user specifications we have chosen the UML metamodel [18, 19] for the following reasons. First, we believe that UML will play an important role as a standard language for specifying and documenting the artifacts of software systems. The Object Management Group (OMG) has integrated the UML into the Object Management Architecture (OMA) and also specified an OA&D CORBAfacility [20] as a standard interface for accessing UML-based tools via CORBA [16]. Furthermore, the current UML standard has become broadly supported by the vendors of graphical modeling tools and is widely used by system architects for analysis and design of software systems. Second, we expect using UML not only during analysis and design but also for specifying value-added data management services will help to reduce the cognitive distance between these two steps and, additionally, may help for bridging the gap between modeling and semi-automatically implementing components of software systems. Third, in comparison with other information models, e. g. the ER-Model, UML has lots of advantages (object orientation, detailed modeling of class structures and semantic relationships, formal object constraint language OCL [17], modeling of behavior and state-oriented aspects, extensibility).

2.3 SERUM Frameworks

The generic methods offered by SERUM to customize repository managers are based on a set of pre-defined frameworks. According to [9] "a framework is a reusable design of all or a part of a system that is represented by a set of abstract classes and the way their instances interact. "SERUM provides a framework for each domain section (versioning, activity support, ...); each framework consists of so-called ***design patterns*** and ***templates,*** which will be discussed in the following subsections.

SERUM Design Patterns

In [5] a design pattern is defined as "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context". SERUM design patterns are used to automatically enhance UML models. A SERUM design pattern consists of two parts, an informal one helping the repository designer to select the pattern, and a formal one used to enhance user specifications. Due to the second part, SERUM design patterns are much more formal than those presented in [5]. Fig. 2 illustrates the process of using SE-

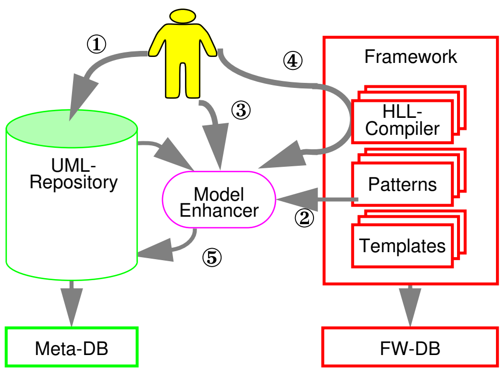

Fig 2: Applying Design Patterns

RUM design patterns. The repository designer specifies an initial UML model of the domain objects (1). Usually, this model is incomplete and mainly contains the application-specific parts. The so-called SERUM model enhancer (SME) is used to apply the *design pattern script* (formal part of a SERUM design pattern) to the user-specified UML model (2). This way, the model is enhanced by new classes and relationship types. Furthermore, already existing classes may be modified (5).

While the SME automatically applies design pattern scripts to UML models, the repository designer must select the patterns to be applied and determine the generic parameters of the pattern application. This can be done either interactively, where the user chooses the patterns and parameters (3), or by external tools (4). These tools may allow the repository designer to express specifications in a more abstract high-level language (HLL) which depends on the corresponding SERUM framework. For instance, the versioning framework, we will use as an example for the discussion in Sect. 3 and Sect. 4, comes along with a specific data definition language (DDL) and a corresponding compiler translating DDL statements into sequences of SME calls.

After the mentioned definition steps are finished, a technology-independent specification of the repository model is stored in the meta-database.

To support the outlined specification process appropriately, a SERUM design pattern must fulfill two requirements. First, searching design patterns must be supported. Several design pattern languages [5] try to cope with this problem. Hence, we want to concentrate on the second requirement, which is the provision of an appropriate design pattern scripting language. As we think, a scripting language must at least offer constructs for specifying the following elements (Fig. 3, in order of appearance):

- *Parameters.* Parameters specify the elements of the input model which are to be manipulated.

```
begin define pattern “ProductDataObject”
  begin parameters
     ClassUList  : aClassSeq;
  end parameters
  begin constraints
    aClasses->forAll( c | exists( “Class”,
                                c.name ))
  end constraints
  begin definitions
    if ( ! exists( “Class”, “BVF_Object” ) )
    {
      begin mdl
        (object Class “BVF_Object”
           // definitions
        )
      end mdl
    };
    if ( ! exists( “Class”, “BVF_PDO”)
    {
      begin mdl
        (object Class “BVF_PDO”
           superclasses
               (list inheritance_relationship_list
                 (object Inheritance_Relationship
                    supplier        “BVF_Object” ))
           // additional definitions
        )
      end mdl
    };
end definitions
begin alter model
  for ( int i; i < aClassSeq.length; i++ )
  {
    Generalization.create_generalization
            (aClassSeq.name(), “BVF_PDObject” );
  }
end alter model
end define pattern
```

Fig 3: Design Pattern Definition Script

- *Constraints.* If a SERUM design pattern is applied in order to enhance an UML model, this model has to fulfill preconditions. The SME checks these preconditions, which we specify in our design pattern definition script using OCL constraints.
- *Default Structures.* Default structures are, for example, abstract base classes with default attributes, relationships and behavior. Within the design pattern script the names of the default structure needed are to be enumerated and the SME subsequently incorporates the corresponding specifications.
- *Model Evolution Operations.* Applying a design pattern to an UML model means integrating the user-specified definitions and the (pre-defined) framework infrastructure used by the pattern. This, actually, happens by manipulating the user specifications: Class definitions may be enriched by new attributes, new relationships, new superclasses, new methods, etc. For that purpose, the script may contain calls of model evolution operations; the parameters of these operations are usually elements of the user specification. In the OMG OA&D CORBA facility [20] a set of model management operations is already defined. Hence, we use similar operations.

After the SME has applied all relevant patterns, a technology for implementing the repository manager has to be chosen. Implementation is supported by *SERUM templates*.

SERUM Templates

Partitioning a framework into a technology dependent (*templates*) and a technology independent (*patterns*) part enables design solutions without being technology dependent. Technology refers to programming languages (C++, Java, ...), ORDBMSs, strategies for architectural design (client-centric, server-centric, repository servers, caching and buffering strategies) and communication mechanisms (CORBA, OLE, RPC, ...).

A SERUM framework must provide implementations for the customized API, the repository servers and the object-relational database schema. For each technology supported by SERUM, templates must be provided with base mappings for user-defined classes. Thus, SERUM provides mappings of UML to the object models of PLs and the DBMSs. Here, object orientation is very helpful. Templates do not only consist of code fragments but also of ready-to-use components. Examples of ready-to-use components are the repository servers and the client-caches. If such components need to be customized, this is done via parameters and not by generated code.

2.4 SERUM Repository Generator

So far we have described the two parts of the SERUM frameworks. The SERUM repository generator (SRG) is the counterpart of the SME w.r.t. templates. It is used for generating the components of a repository manager using the templates of the framework and for integrating the resulting APIs, repository servers and database schemas.

Generating the components of the repository manager means generating code, supplementing the user-defined classes enabling them to cooperate with the framework classes. This process exploits the templates providing code fragments as well as the UML specification providing the generic parameters. If application logic is involved, the repository designer has to deliver an implementation. Both, the SRG and the repository designer, can take advantage of object-oriented concepts. Some components of the frameworks are ready-to-use and may be customized via parameters.

3 Versioning Principles

As already mentioned, a spectrum of versioning functionality/mechanisms is required to adequately support applications with different demands. According to [10] we think that there is a core of basic versioning facilities which can be used (extended, refined) to establish a version model fulfilling the specific needs of a given (class of) application(s). Furthermore we think that this tailoring process consists of two steps:

(1) *Adapting the basic versioning framework.* Starting from a so-called ***basic versioning framework (BVF)***, which represents the above mentioned core of basic versioning facilities (see Sect. 3.1), an ***adapted versioning framework (AVF)*** is derived, which, in turn, represents the versioning data model[5] adequately supporting the application(s) in mind.

(2) Applying the adapted versioning framework. The AVF resulting from step (1) and a product data model[6] given for a certain application are 'melted' in a way delivering a customized versioning repository manager. This step is detailed in Sect. 4.

In this section we want to emphasize the first step.

3.1 The Idea of the Basic Versioning Framework

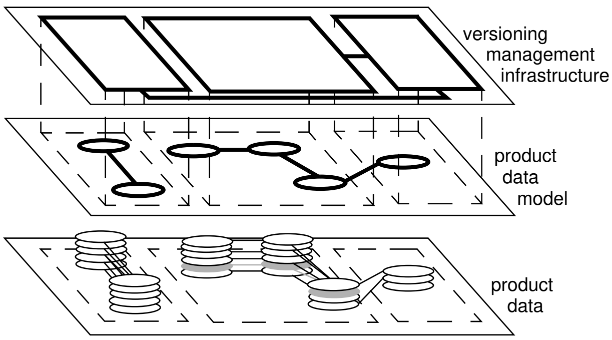

Fig 4: Basic Model of Versioning

Since we think that *handling versions is a special way of managing views to product data*, our BVF follows a bottom-up approach. Assume, there is some product data given, consisting of objects and associations (relations). In the lower layer of Fig. 4 ***product data objects*** (PDO) are shown as ovals and ***product data associations*** (PDA)[7] as lines. A view on this product data, like the grey marked excerpt in the lower layer of Fig. 4 (consisting of PDOs and PDAs) is a version. The structure of the product data is given as a ***product data model*** (PDM, see middle layer in Fig. 4). Thus, at this layer, ovals and lines do not represent instances but types (classes). In order to be able to version not only single PDOs but also semantic units, it must be possible to specify so-called ***versionable structures*** (VS), which are illustrated as rectangles at the upper layer of Fig. 4. The set of structuring elements associated with this upper layer (VSs, corresponding associations and additional versioning information) is called the ***version management infrastructure*** (VMI). The overall advantage of this bottom-up

5. Note, the notion *data model* is used in the meaning of a modeling system. Thus, it is to be understood in a similar way, the term relational data model is understood. It is not meant to be a database schema.

6. This time, the notion *data model* refers to a sort of database schema, i. e. the result of a modeling process. At this point, we stick to this notion (*model)*, since it is mostly used in the literature this way.

7. Note, the lower layer of figure 3 shows data occurrences and not types!

approach is that the PDM may be defined without coping with versioning aspects and afterwards the versioning semantics may be defined on top of the PDM.

3.2 Elements of the Basic Versioning Framework

First of all, the PDM contains *PDO* classes as well as *PDA* classes (see UML-structures in Fig. 5). The class `Version` serves as a container grouping together elements which are to be versioned as units. Note that there are no restrictions about the multiplicity of the associations connecting `PDO` and `Version` (`VPDOAssociation`), or `PDA` and `Version` (`VPDAAssociation`) respectively. For example, each PDO may be part of several versions (but does not have to belong to a version), and each version may consist of several PDOs. Versions may also have associations with each other (`VVAssociation`), e. g. to express derivation relationships or to represent overlapping. As already mentioned, PDM classes are assigned to (classes of) VSs. A version must belong to exactly one VS (`VSVAssociation`), VSs (of course) may have many versions. Like versions, VSs can be associated with each other (`VSVSAssociation`).

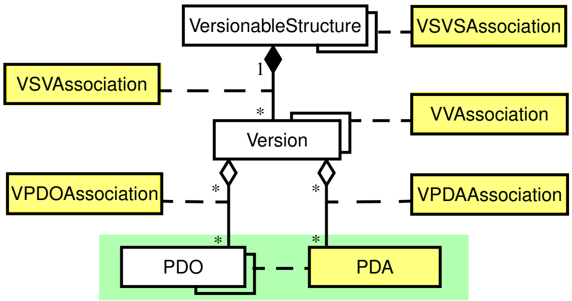

Fig 5: Basic Structure of Versioning

Note that there are the following dependencies concerning associations. A PDA may only be part of a version if at least one of the two PDOs connected by this PDA also belongs to that version. If the other PDOs do not belong to the same version, then there must be a VVAssociation between both versions (the PDOs belong to), abstracting from the PDA. Furthermore, if versions of different VSs are related to each other, then there must be a corresponding VSVSAssociation, in turn, abstracting from the associations at the version level and at the product data level.

Recall that Fig. 5 only contains the classes `PDO` and `PDA` in order to clarify the semantics of the BVF structures. Actually the process of assigning PDM classes to VSs is part of step (2), as mentioned at the beginning of this section. In contrast, defining specific semantic roles PDOs/PDAs may play or have to play within a certain AVF is part of step (1). Different semantics are expressed by corresponding subclasses of the classes `PDO` and `PDA`. Due to space restrictions, we cannot detail all these aspects, but want to give an overview, which subclasses provide dedicated versioning semantics PDOs and associations may be equipped with. A more detailed description can be found in [14].

3.2.1 Specializing the PDO Classes

The following classes (see Fig. 6) are offered by the BVF in order to provide appropriate version semantics for PDOs. The special semantics are encoded in the structural and behavioral elements of these classes.

- Instances of the class `Object_NonVersioned` cannot be associated to any versioned objects and cannot be part of a version.

- An Instance of the class `PDO_NonVersioned` can be associated to versioned and non-versioned objects but cannot be part of a version itself. To associate an instance of this class to a versioned object, a special PDA class is needed, which will be outlined in the next subsection.

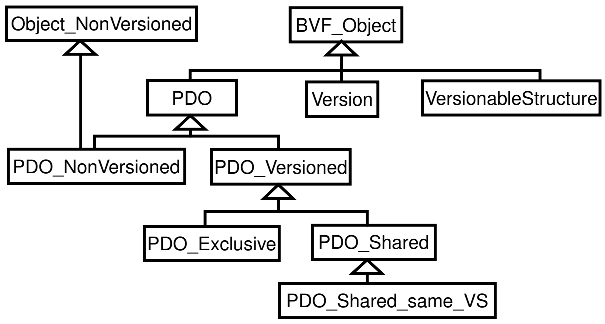

Fig 6: Inheritance Hierarchy of the Object-Classes

- Instances of the class `PDO_Exclusive` may belong to at most one version. Because a version belongs to exactly one VS, instances of `PDO_Exclusive` can be part of at most one VS, too.
- Instances of `PDO_Shared` may be associated with several VSs.
- Instances of `PDO_Shared_same_VS` may be part of several versions, but these versions must belong to the same VS.

3.2.2 Specializing the Association Classes

Similar to PDO classes pre-defined association classes are carriers of dedicated versioning semantics in the BVF. Obviously not only `PDA`, but also `VVAssociation` and `VS-VSAssociation` have to be considered.

3.2.2.1 Product Data Associations

Product data associations may occur in different constellations. Thus, we divide the set of pre-defined association classes into four groups (see illustration in Fig. 7):

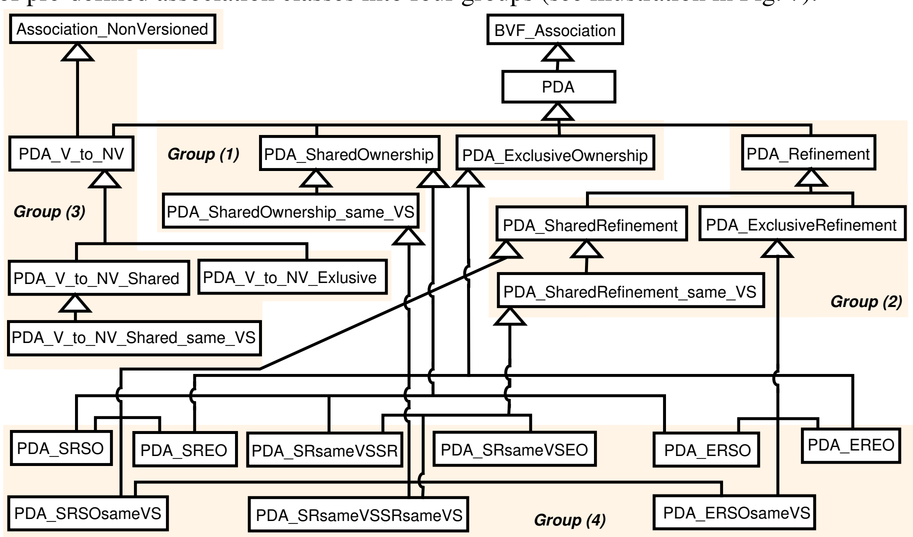

Fig 7: Inheritance Hierarchy of PDA Classes

(1) The simplest possible case is an association between PDOs of the same version. We consider these associations being part of the corresponding version. Nevertheless, we have to distinguish three different types.

- Instances of the class `PDA_ExclusiveOwnership` can be part of at most one version and, therefore, belong to at most one VS. Such PDAs may associate (direct and transitive) instances of the class `PDO_Versioned` (see above). Note that two PDOs, associated with each other via `PDA_ExclusiveOwnership`, may also belong to other versions, but each PDA among them must belong to exactly one version.
- Instances of `PDA_SharedOwnership_same_VS` may belong to different versions, but these versions must be part of the same VS. These associations can connect PDOs of the classes `PDO_Shared` and `PDO_Shared_same_VS`.
- Instances of `PDA_SharedOwnership` can be part of different versions of different VSs. Therefore, both associated PDOs have to be instances of `PDO_Shared`.

(2) PDAs can also connect PDOs of different versions. This class of PDAs is called `PDA_Refinement`. Such a PDA is considered to be "part-of" a VVAssociation. `PDA_Refinement` is refined into three subclasses.

- Instances of `PDA_ExclusiveRefinement` can be part of at most one VVAssociation and, therefore, either be part of exactly two versions (which are associated through the VVAssociation) or part of no version. This, in turn, implies that an instance of `PDA_ExclusiveRefinement` can be part of none or two VSs.
- Instances of `PDA_SharedRefinement` can be part of several VVAssociations and, therefore, be part of several versions and several VSs.
- Instances of `PDA_SharedRefinement_same_VS` can belong to several VVAssociations and, therefore, also to several versions. But all VVAssociations have to belong to the same VSVSAssociation and therefore an instance of `PDA_SharedRefinement_same_VS` can belong to at most two VSs.

(3) Special PDA classes are needed to associate not-versioned objects.

- Instances of `Association_NonVersioned` connect non-versioned objects.
- Instances of the class `PDA_V_to_NV` connect a versioned and a non-versioned PDO, respectively. According to the discussions above, an instance of `PDA_V_to_NV` can be exclusive (subclass `PDA_V_to_NV_Shared_Exclusive`), shared between versions of different VSs (subclass `PDA_V_to_NV_Shared`) or shared between versions of the same VS (subclass `PDA_V_to_NV_Shared_same_VS`).

(4) To create an AVF, it can also be helpful to have pre-defined definitions for associations which may exist within versions as well as between versions. The classes of this group (see Fig. 7) result from combinations of concepts mentioned in (1) and (2).

3.2.2.2 Associations between Versions and between VSs

Associations between versions (`VVAssociation`) can be used to manage the derivation graph (`VVA_Derivation`) or to abstract from `PDA_Refinement` instances (`VVA_Refinement`). While a `VVA_Derivation` instance can only associate versions of the same `VersionableStructure` instance, an instance of `VVA_Refinement` always associates versions of different `VersionableStructure` instances.

The class of associations between VSs (`VS-VSAssociation`) can be specialized to a class called `VSVSA_Refinement`. Instances of this class abstract from instances of `VVA_Refinement`. Fig. 8 shows the dependencies between the different refinement classes. Each `PDA_Refinement` instance can be part of several `VVA_Refinement` instances, but each `VVA_Refinement` instance has to belong to exactly one instance of `VSVSA_Refinement`.

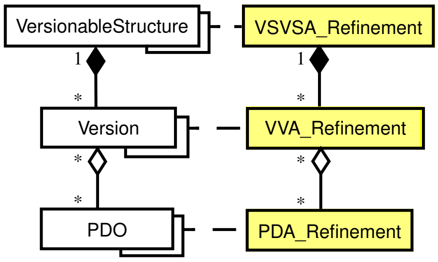

Fig 8: Refinement-Associations

3.3 Collaboration of BVF Classes

To be adequately adaptable, besides the mentioned set of classes a communication infrastructure is required. After having discussed the structural elements of the framework in the sections above, we will now have a closer look at the communication infrastructure and how it can be used to express particular versioning semantics.

Designing a framework means separating variant from invariant parts. The data objects, especially the PDOs, mainly consist of variant parts, since their attributes and methods are typically application-specific. In contrast, the relationships, which are the structural backbone of the versioning framework, do not depend on application-specific definitions. Hence, we have chosen to control the semantics of the relationships by associating an additional object, called the *signal handler*, with each data object. The signal handler has to control the operations, a user applies to the associated object, possibly by implicitly invoking further operations.

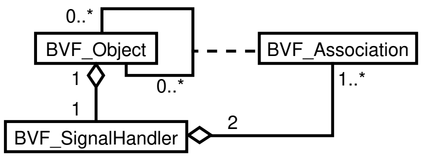

Fig 9: Communication Infrastructure

The idea of our signal handlers is similar to the observer design pattern presented in [5], where the signal handler acts as the observer (Fig. 9). An action, i. e. an operation applied to a data object, triggers two events: a before-action event and an after-action event. After an event has occurred, the event handler, implemented by the `BVF_Object` class, sends a signal to the signal-handler (`BVF_SignalHandler`). In contrast to the original design pattern, the observer is closely coupled with its data object, but only reacts to an event, if it might be necessary to take further action in order to maintain the inherent integrity of one or more of the object's associations.

Let us examine a (versioning) example: Assume, the notion of 'frozen' versions is to be specified. A frozen version ensures a durable view on a product data state. In this scenario, a PDO may be shared between different versions, some of which are frozen and others are not. Assume further, an update operation is applied to this PDO. Now, to ensure versioning semantics, the object must be copied. Then, the copy is considered to be the representative of the PDO in the frozen versions, and the original may be updated (in the context of the non-frozen) versions afterwards. Fig. 10 illustrates a corresponding sample message flow. It starts with an update message received by PDO P_1, which is shared between two versions V_1 (frozen) and V_2 (updateable). The update message initiates a before-update event (1.1). After the event handler has recognized the event, it sends a signal to the signal-handler (1.1.1). The signal handler checks whether or not corresponding versions are frozen (messages 1.1.1.1 to 1.1.1.4). Since V_2 is frozen, the signal handler has to create a copy of P_1, disconnect V_2 from P_1 and connect it to the new instance (1.1.1.5 to 1.1.1.7). Afterwards, the update operation is applied to P_1 (1.2).

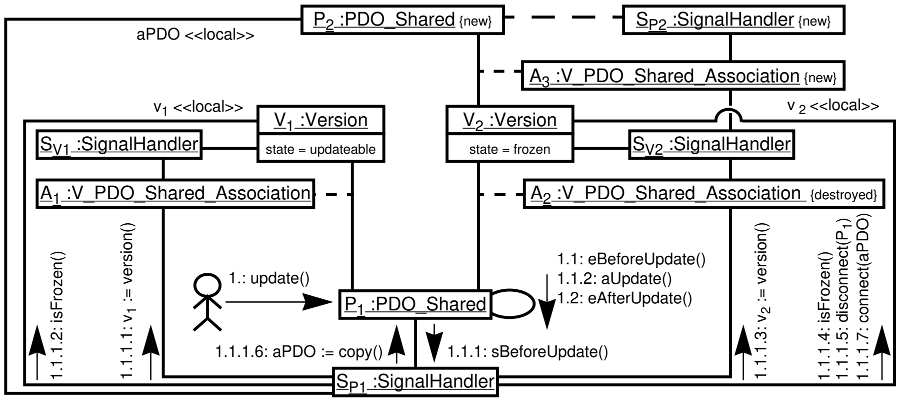

Fig 10: Message Flow

Note that no variant (application-specific) parts of objects are affected except the copy method. Besides this method, which either has to be provided by the user or must be generated from the specification, only framework classes, i. e. classes initially provided by the BVF, participate.

4 Generating a Repository-Manager in SERUM

In the previous section, we have introduced our BVF by outlining its structural elements (Sect. 3.2) as well as its communication mechanisms (Sect. 3.3), both allowing to establish an AVF representing a tailored versioning model. In this section, we want to clarify that the AVF is still a framework and to illustrate, how this framework can be used to generate a repository manager tailored to the needs of software development applications. The following subsections outline the steps which are to be carried out in SERUM in order to reach this goal.

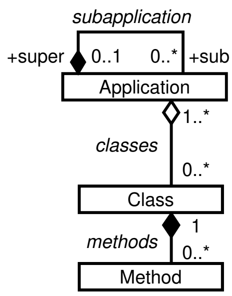

Fig 11: Sample PDM

4.1 Providing the Product Data Model

As already mentioned, the prerequisite for applying the AVF is a PDM, which, first of all, does not take any versioning aspects into account. Fig. 11 shows a software development example. It consists of applications, which may be divided into sub-applications in order to ease administration. Several classes may belong to each (sub)application and a class is not allowed to exist without a corresponding application. Furthermore, a class may contain methods, whereas methods must not exist without belonging to a class. Note, to simplify explanation, the UML model (Fig. 11) is not complete and the example is intentionally very simple.

4.2 Specifying Versioning Structures

In this step, the repository designer has to incorporate versioning aspects by synthesizing the VMI(-specifications), as introduced in Sect. 3.1, and the given PDM. A graphical illustration of the result is given in Fig. 12. Two versionable structures, named `VS_Application` and `VS_Class`, are specified. `VS_Application` allows versioning each single application and `VS_Class` allows versioning each class together with its methods as units.

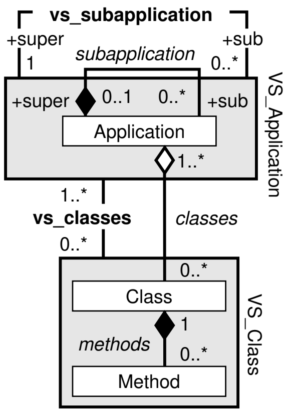

Fig 12: PDM with Versioning Structures

For specification purposes, we use a version definition language (VDL), which can be considered as a special-purpose HLL (cf. Sect. 2). The VDL statements corresponding to the graphical illustration in Fig. 12 are shown in Fig. 13. The first statement defines the VS `VS_Application`. The class which is in charge of managing versions of this VS has been named `V_Application` (see line 4). The cardinality restrictions specified in the OBJECTS-clause (see line 2) clarify that an instance of `Application` is not allowed to be part of two versions. The second statement (see line 6) defines a subclass of `VSVSA_Refinement` as well as a corresponding subclass of `VVA_Refinement` (cf. Sect. 3.2.2.2) abstracting from the `subapplication` relationship. This allows handling an application hierarchy also at version (and VS) level.

```
1.   DEFINE VERSIONABLE STRUCTURE
2.   OBJECTS(Application [0..1] EXCLUSIVE [1]) WITH
3.        VS_NAME IS VS_Application
4.        V_NAME IS V_Application
5.
6.   DEFINE LINK REFINEMENT OF
7.     subapplication(Application(super), Application(sub))
8.   BETWEEN VS_Application(super) AND VS_Application(sub)
9.     WITH
10.       VS_NAME IS vs_subapplication
11.       V_NAME IS v_subapplication
12.
13.  DEFINE VERSIONABLE STRUCTURE
14.  OBJECTS( Class [0..1] EXCLUSIVE [1],
15.          Method [*] EXCLUSIVE [1])
16.  ASSOCIATIONS(methods(Class, Method)) WITH
17.       VS_NAME IS VS_Class
18.       V_NAME IS V_Class
19.
20.  DEFINE LINK REFINEMENT OF classes(Application, Class)
21.  BETWEEN VS_Application(Application) AND VS_Class(Class)
22.    WITH
23.       VS_NAME IS vs_classes
24.       V_NAME IS v_classes
```

Fig 13: VDL Script

Looking at the third statement (line 13), we see that, again, at most one instance of `Class` is allowed to belong to a version of `VS_Class`. An instance of `Method` may only occur in a version of `VS_Class`, if it is connected to an instance of `Class` which is also contained in the same version. Therefore, the association `methods` is under control of the version and the cardinality restrictions regarding this association are checked by the framework at a 'per version' base. Last but not least (fourth statement, see line 20) the refinement properties of the association `classes` at the VS level and the version level are specified.

4.3 PDVM Evolution

Next, the PDM is enhanced by definitions (classes) contributing to managing versions in a way determined by the chosen AVF and previous definitions of the repository designer (given as a VDL script). Consequently, a design pattern gets a PDM and one or more VDL statements as input and delivers a modified PDM as output. Modifications (additional class definitions, refinements of existing class definitions, etc.) pertain specifications which, in turn, ensure the versioning semantics expressed by the VDL definition w.r.t. the underlying AVF.

Design patterns are applied by executing corresponding design pattern scripts. We want to illustrate this complex process by examining examples of its most typical steps.

In order to provide the base functionality needed for managing PDOs and PDAs, each PDM class becomes a subclass of `PDO` and each association is replaced by an association class, which is specified as a subclass of `PDA`[8].

```
begin define pattern "VersionableStructure"
begin parameters
  ClassUList          aPdoS;
  sequence<boolean>   aPdoExcS;
  Multiplicity        aPdoMultiS [];
  Multiplicity        aPdoVMultiS [];
  AssociationUList    aPdaS;
  Name                name;
  Name                vname;
end parameter
begin constraints
end constraints
begin definitions
  begin mdl
    (object Class <name>
      superclasses
        (list inheritance_relationship_list
          (object Inheritance_Relationship
             supplier "VersionableStructure")))
    (object Class <vname>
      superclasses
        (list inheritance_relationship_list
          (object Inheritance_Relationship
             supplier        "Version")))
    (object Association "version"
      connections
        (list association_end_list
          (object AssociationEnd ""
            aggregation   composite
            multiplicity  "[1]"
            type          name)
          (object AssociationEnd ""
            aggregation   none
            multiplicity  "[1..*]"
            type          vname)))
  end mdl
end definitions
begin alter model
  // all classes are subclasses of PDO
  ApplyPattern.ProductDataObject(aPdoS);
  // connect version to pdo
  for(int i = 0; i < aPdoS.length; i++)
  {
    AssociationEndUList assoEnds    =
        new AssociationEndUList();
    assoEnd[0] =
        AssociationEnd.create_association_end
          (aPdoS[i].name(), true, false, none,
             instance, aPdoMultiS[i] ) );
    assoEnd[1] =
        AssociationEnd.create_association_end
          (vname, true, false, none, instance,
             aPdoVMultiS[i]) );
    Association asso =
        Association.create_association
          (newName(), false, false, false);
    asso.add_connection(assoEnds);
  };
end define pattern
```

Fig 14: Sample Design Pattern Script

8. Note that each AVF offers a class PDO as well as a class PDA.

Each VS definition (in the VDL script) is processed by the design pattern Versionable-Structure; Fig. 14 shows the corresponding script. This script takes a VS definition as input and delivers definitions of two (associated) classes as output. The first class serves for representing versionable structures and managing their versions, the second for representing versions and managing their PDOs. Each link definition leads to three new association classes. For example, the second VDL statement in Fig. 13 adds the association classes `vs_subapplication`, `v_subapplication` and `subapplication`. The latter replaces the original reflexive association on `Application`.

```
Application
    self.version.v_subapplication->forAll
    (vsa|(vsa.sub=self.version)
    implies (1>=count(vsa.pda->select(sa|sa.sub=self))))
```

Fig 15: Sample Constraint

Since (in our sample scenario) an application may be connected to more than one super-application, each included in another version, we have to relax the cardinality restrictions at the product data level and consider the version level, when checking the cardinality restrictions. OCL constraints [17] are needed, because of the more complex consistency requirements. The sample constraint given in Fig. 15, which is derived from an UML template, ensures that an application is not connected to more than one super-application regarding a pair of associated versions.

An excerpt from the enhanced PDM delivered by the complete model evolution process of our example is shown in Fig. 16. Due to simplicity, we did not include all model elements. For example, the generalization relationships are not displayed.

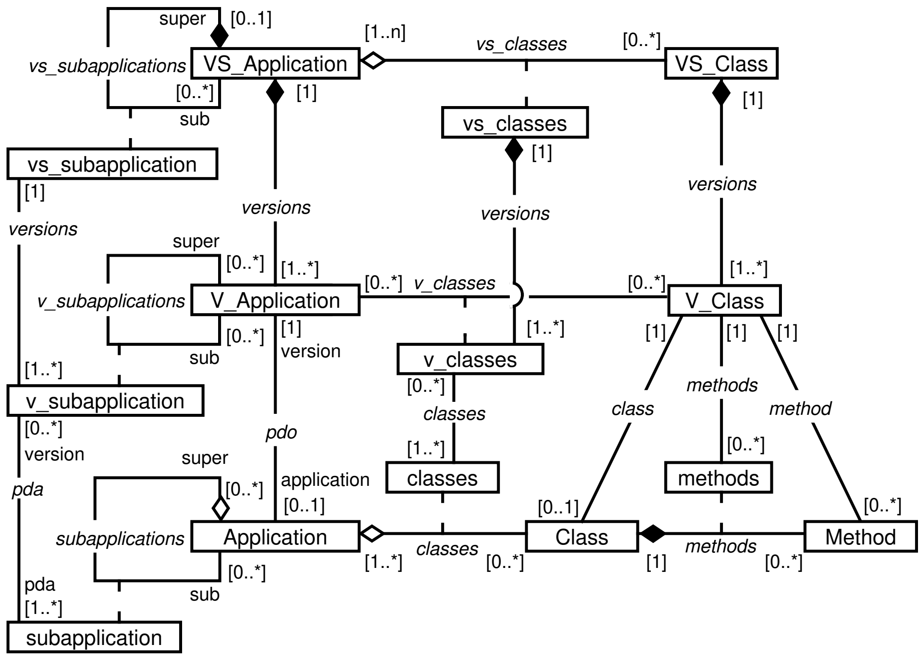

Fig 16: Enhanced PDM

4.4 Generating the Repository Manager

The next step in the SERUM approach is to generate a repository manager from the enhanced model. An ORDBMS is used by the repository manager for storing data. Therefore, we have to create an ORDBMS schema (consisting of table definitions, UDTs and UDFs). Additionally, an API for accessing the versioned data is needed. Although different programming languages may be supported, we will restrict our considerations to examples in Java. Further components of a repository manager may be repository servers. A repository server associated with the versioning framework may enforce a dedicated synchronization protocol, as for example the C^3-locking-protocol [23]. Due to space restrictions, we cannot detail this aspect; in the following subsections, we will give examples for schema design and some selected API functions.

ORDBMS Schema

Currently, evolution of ORDBMS technology moves very fast. Until now, the SQL-3 standard [8] is still in evolution and existing systems calling themselves object-relational differ to a great extend in concepts implemented and the syntax used. Our examples mainly use the language of the ORDBMS IDS/UDO[9] ([6]).

Each class in the enhanced PDM is mapped by the SRG (see Sect. 2.4) to a named row type. We exploit inheritance by subclassing framework types. The PDO classes inherit from the class `PDO`, version classes from `Version` and versionable structures from `VersionableStructure`. Association classes are mapped to named row types, too. Since IDS/UDO does not support references at the moment, we use foreign keys for implementing relationships in this example (Fig. 17).

```
CREATE ROW TYPE VS_Application
( ) UNDER VersionableStructure;
CREATE ROW TYPE V_Application
( ) UNDER Version;
CREATE ROW TYPE Application
( ) UNDER PDObject;
CREATE ROW TYPE subapplication
( super   IdType,
  sub     IdType,
) UNDER VSVSA_Refinement;
CREATE ROW TYPE v_subapplication
( super       IdType,
  sub         IdType,
  vstructure  IdType
) UNDER VVA_Refinement;
CREATE ROW TYPE vs_subapplication
( super   IdType,
  sub     IdType,
  version IdType
) UNDER PDAssociation;
```

Fig 17: Schema Definition (excerpt)

```
CREATE FUNCTION isTop(va V_Application): boolean;
CREATE FUNCTION isVersionOf(va1 V_Application, va2 Application): boolean;
CREATE FUNCTION isVersionOf(va V_Application, vs VS_Application): boolean;
CREATE FUNCTION subapplicationConnects
      (super Application, sub Application): boolean;
CREATE FUNCTION v_subapplicationConnects
      (super V_Application, sub V_Application): boolean;
CREATE FUNCTION vs_subapplicationConnects
      (super VS_Application, sub VS_Application): boolean;
CREATE FUNCTION subapplicationsCount
      (pdo Application, super V_Application, sub V_Application): integer;
```

Fig 18: UDFs (excerpt)

In order to support convenient access via the SQL interface of the ORDBMS, the automatically generated row types representing association classes are encapsulated. Thus, beside others, UDFs are created (see sample functions in Fig. 18) to check,

9. Informix Dynamic Server - Universal Data Option

- whether a version is the current (topical) one of its VS (Fig. 18: 1);
- whether two given occurrences are connected (Fig. 18: 4, 6, 8);
- whether a given PDO belongs to a given version (Fig. 18: 2);
- whether a given version belongs to a given VS (Fig. 18: 3);
- how many PDOs are connected to a given PDO w.r.t. a given version association (Fig. 18: 10).

By using generated functions, the OCL constraint, given in Sect. 4.3, can be easily mapped to SQL (see Fig. 19).

Additionally, the SRG generates UDFs for manipulating relationships, creating instances of versionable structures, creating versions, etc. from the UML specification.

```
CHECK(NOT EXISTS
SELECT   *
FROM     application a1,
         v_application va1, v_application va2
WHERE    NOT (1 >= subapplicationsCount(a1, va1, va2))
```

Fig 19: Cardinality Constraint

Application Programming Interface

Object-oriented programming languages (OOPL) have become the state of the art in system development. Hence, we intend to provide APIs for OOPLs. In order to give an idea of how to embed the versioning functionality (offered by the generated repository manager) in an OOPL, we give an example of integrating API calls into Java [1]:

```
public  class Application extends PDO {
  public  ApplicationList sub() throws AmbiguousVersionException;
  public  Subapplication subapplication() throws ... ;
  public  ApplicationList sub( V_Application aVersion ) throws ... ;
  // ...more methods and attributes }
public  class Subapplication extends PDA {
  public  V_SubapplicationList versions()
          throws AmbiguousVersionableStructureException;
  // ...more methods and attributes }
public  class V_Subapplication extends VVA_Refinement {
  public  V_Application sub() throws AmbiguousVersionableStructureException;
  // ...more methods and attributes }
```

Fig 20: Sample Java API Classes

Each class in the enhanced PDM is mapped to a Java class. Methods are generated for the traversal of relationships. Assume that we want to navigate from an application to its sub-applications. Without taking a certain version context into account the result would be ambiguous. Hence, the Java class `Application` is equipped with a set of methods enabling navigation from one version to another. If the context of the target PDOs is definite, either determined by the workspace context[10] or by a configuration context[11], the `sub()` method can be used. Calling this method without a definite version context w.r.t. the target leads to an exception. The exception object provides a list of all versions of sub-applications. This list can be used to choose a target version and

10. A workspace is a private subset of the repository data.

11. A configuration provides a consistent view to the PDO level, i. e. no ambiguous version contexts occur. Configurations are beyond the scope of this paper.

access the application belonging to this version with the `sub(V_Application)` method. Of course, the developer can also collect the eligible versions in advance. The association classes `Subapplication` and `V_Subapplication` provide methods needed for that purpose (Fig. 20: 6, 9).

5 Conclusions

In this paper, we have introduced the SERUM approach providing an infrastructure for tailoring repository managers to special application needs by exploiting generic methods. SERUM considers a design domain, as for example software engineering, consisting of several domain sections. The (sample) domain section used in this paper to illustrate SERUM's genericity has been versioning, one of the key aspects of repository technology. By means of the versioning aspects, our discussions showed:

- There is a core of basic versioning facilities (basic versioning framework) which can be tailored to dedicated versioning models (adapted versioning frameworks). Disregarding configurations, the basic versioning framework introduced in this paper fulfills the demands stated in [10].
- Since handling versions is a special way of managing views to product data, it is easy for the repository designer to synthesize the version management infrastructure with a (non-versioned) product data model. The actual work of expanding the product data model by versioning facilities is performed be applying pre-defined design patterns.
- The SERUM repository generator is able to generate the (versioning) repository manager from the enhanced product data model. The latter encompasses an OR-DBMS schema, special UDFs for version manipulations in SQL, special UDFs for checkin/checkout as well as special UDFs for embedding version manipulations into an OOPL.

Besides offering the possibility of adapting versioning facilities, the SERUM approach, in general, is beneficial for the following reasons:

- Users may choose from pre-defined UML specifications.
- Application-specific semantics may be incorporated by refining, enhancing, and adapting the chosen UML model in an intuitive manner. This customization process is supported by pre-defined technology-independent design patterns in order to relieve users.
- Repository manager functionality (code) is automatically generated from the customized UML model by applying technology-dependent design templates.

Obviously, the basic idea of reusing design artifacts has effectively been put into practice by the SERUM approach. The paper has further shown that the concepts of object orientation and object-relational database technology are crucial prerequisites for gaining the mentioned benefits. The former enables reuse and model adoption/refinement in an intuitive manner and the latter provides extensible data management, which, in turn, is essential for integrating the generated repository manager functionality with adequate data management facilities. Thus, SERUM is the first approach generating extenders of object-relational database systems, which, in turn, implement repository manager functionality.

As future work, we intend

- to detail concepts for generating repository servers, i. e. functionality the DBMS may not be extended by,
- to examine other domain sections, e. g. designflow management, more detailed, and
- to evaluate/validate the overall SERUM approach in realistic application scenarios.

6 Literature

[1] Arnold, K., Gosling, J.: The Java Programming Language, Addison-Wesley, 1996.

[2] Bernstein, P.A., Dayal, U.: An Overview of Repository Technology, Proc. 20th VLDB, Santiago, Chile, September, 1994, pp. 705-713.

[3] Bernstein, P.A.: Repository Internals, Tutorial Handouts, 21th VLDB, Zürich, Schweiz, September, 1995.

[4] Conradi, R., Westfechtel, B.: Version Models for Software Configuration Management. Technical Report AIB 96-10, RWTH Aachen, October, 1996.

[5] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Publishing Company, 1995.

[6] Getting Started with INFORMIX-Universal Server, Version 9.1, Informix Software Inc., März 1997.

[7] Harrison, D., Newton, R., Spickelmier, R., Barnes, T.: Electronic CAD Frameworks, Proc. of the IEEE, 78:2, February, 1990, pp. 393-417.

[8] ISO Final Committee Draft - Database Language SQL ´ftp://jerry.ece.umassd.edu/isowg3/dbl/BASEdocs/public/´, 1998.

[9] Johnson, R. E.: Frameworks = Components + Patterns, CACM, 40:10, October, 1997, pp. 39-42.

[10] Katz, R.: Towards a Unified Framework for Version Modeling in Engineering Databases, ACM Computing Surveys, Vol. 22, No. 4, December, 1990, pp. 375-408.

[11] Kim, W.: Object-Relational - The unification of object and relational database technology, UniSQL White Paper, 1996.

[12] Krueger, C. W.: Software Reuse, ACM Computing Surveys, 24:2, June, 1992, pp. 131-183.

[13] Loeser, H.: Exploiting Extensibility of ORDBMS for client/server-based Application Systems, Proc. 10. GI-Workshop Grundlagen von Datenbanken, Konstanz, June, 1998, pp. 77-81, in german.

[14] Mahnke, W., Ritter, N., Steiert, H.-P.: A basic versioning framework for SERUM, Technical Report, Sonderforschungsbereich 501, Dept. of Computer Science, University of Kaiserslautern, 1998, in preparation.

[15] Nink, U., Ritter, N.: Database Application Programming with Versioned Complex Objects, in Klaus R. Dittrich, Andreas Geppert (eds): Proceedings of the BTW'97, March, 1997, pp. 172-191.

[16] OMG, The Common Object Request Broker: Architecture and Specification, Version 2.2, OMG Document ad/98-07-01, August, 1998.

[17] OMG, Object Constraint Language Specification, Version 1.1, OMG Document ad/97-08-08, September, 1997.

[18] OMG, UML Notation Guide, Version 1.1, OMG Document ad/97-08-05, September, 1997.

[19] OMG, UML Semantics, Version 1.1, OMG Document ad/97-08-04, September, 1997.

[20] OMG, OA&D CORBAfacility Interface Definition, Version 1.1, OMG Document ad/97-08-09, September, 1997.

[21] Rammig, F. J., Steinmüller, B.: Frameworks and Design Environments, Informatik-Spektrum, Vol. 15, 1992, pp. 33-43, in german.

[22] Rao, B.R.: Object-Oriented Databases, Technology, Applications, and Products, Database Experts' Series, McGraw-Hill, 1994.

[23] Ritter, N.: The C^3-Locking-Protocoll - A Concurrency Control Mechanism For Design Environments, ITG-Fachbericht 137, Softwaretechnik in Automation und Kommunikation (STAK'96), Munich , March, 1996, pp. 95-110.

[24] Stonebraker, M., Brown, P., Moore, D.: Object-Relational DBMSs, Second Edition, Morgan Kaufmann Series in Data Management Systems, September 1998.

[25] Wakeman, L., Jowett, J.: PCTE - The Standard for Open Repositories, Prentice Hall, 1993.

[26] van der Wolf, P.: CAD Frameworks - Principles and Architecture, Kluwer Academic, 1994.