In: Proc. 2nd Int. Symposium on Cooperative Database Systems for Advanced Applications (CODAS'99), keynote paper, Springer-Verlag, Singapore, March 1999

Generating Repository Managers for Cooperative Design Applications[1]

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

Department of Computer Science
University of Kaiserslautern
P O Box 3049, 67653 Kaiserslautern, Germany
e-mail: {haerder/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. Some of the most important features of a repository are version management and activity management. Regarding these issues repository technology, as we think, should not only provide rigid, generic services, but should be flexible enough to reflect different application needs. For that reason, we propose providing repository managers by generic methods, i. e., generating corresponding functionality. In this paper, we consider a sample, representative cooperation model exposing the crucial role of *versioning*, which, in turn, is used as an example to illustrate the overall process of generating a corresponding repository manager in our approach. Thus, we want to show that by exploiting generic methods and reuse as well as the extensibility properties of new object-relational database technology, repository managers can be flexibly tailored to special application needs.

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

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. Our concern is to make well customized repositories available for users. To reach this goal, we propose a framework approach, allowing users to reuse predefined specifications of repository 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, we first want to mention basic services a repository manager has to provide in order to serve cooperative design applications properly.

Repositories

In the literature, two kinds of technology can be found aiming at an adequate support of design applications: frameworks [22, 8, 28] and repositories [3, 4, 27]. 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

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".

application. Repositories or, more precisely, 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.

In [3], 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 modelling, 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 [3]: *checkout/checkin, version control, configuration control, notification, context management and workflow control*.

Requirements of Cooperative Design Applications

After having given a very brief list of services usually associated with a repository manager, we now want to consider the actual needs of design applications in a little more detail in order to identify those repository services really important for that class of applications. Obviously, repository managers for design applications have to fulfil requirements w. r. t. both data management as well as activity management.

Regarding *data management*, versioning facilities are to be provided. Complex structured design data is usually versioned for different purposes. First, the process evolution can be documented by associating design process states with versions (or configurations, respectively) representing design object states. Second, versions enable designers to step back to previously established design states in order to try another way of reaching the given design goal. Third, versioning allows for configuring complex design objects from versions of composing design objects. Thus a version model is required, providing adequate data structures for versions and configurations and appropriate (generic) operations for manipulating versions and configurations.

This leads us to facilities for processing design data, i. e., *activity management* facilities. Activity management in our opinion comprises several important aspects. First, a processing concept is needed which is well suited for design applications. Thus, client-side data processing must be supported. Second, data processing must be encapsulated by (design) transactions in order to avoid inconsistencies. Transaction management facilities, esp. conflict management mechanisms [24, 25], must be tailored to adequately deal with versioned data. Besides transaction management, activity management, as we think, comprises designflow management and cooperation control. Both heavily rely on versioning. While states within design flows and, thereby, certain important design qualities are reflected by versions or configurations, cooperation control deals with controlling visibility of design object versions within a team of cooperating designers (design tasks).

A design model reflecting all the needs mentioned so far in an integrated manner is the CONCORD model [23, 25]. Therefore, we decided to consider it as a representative allowing to demonstrate our approach of precisely tailoring repository managers to the special needs of cooperative design applications. CONCORD will be further introduced in Sect. 2.

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 enabling technology for this approach, we use the extensibility features of new object-relational database systems [12, 26, 15], i. e., we generate extensions of object-relational database systems, which, in turn, implement repository manager functionality. Our approach, which is called the SERUM approach, will be further introduced in Sect. 3.

Overview of the paper

As already mentioned, we want to illustrate the benefits of our SERUM approach in the field of cooperative design applications. Therefore, we first introduce the CONCORD model as a representative design model in Sect. 2. We report on an earlier implementation of a repository (named VStore, see Sect. 2.2), which has been used as implementation platform for the CONCORD prototype. VStore has especially been designed and implemented (manually) for serving as CONCORD repository, and, thus, provides generic repository services supporting CONCORD. Starting from this generic approach, we want to emphasize the benefits of a generative approach, which is much more flexible w. r. t. tailoring services and much more convenient w. r. t. providing the corresponding system components. Correspondingly, Sect. 3 introduces the SERUM process of customizing and generating a repository manager. Sect. 4 discusses this process in more detail by using the provision of adequate versioning facilities as an example. Sect. 5 concludes the paper and gives an outlook to future work. Throughout the paper we will further see that object-relational database technology may be extremely helpful in such a generative approach.

2. Overview of CONCORD

The CONCORD (CONtrolling COopeRation in Design environments) model [23] is based on a versioning model and captures the dynamics inherent to design processes. To reflect the spectrum of requirements, such as *hierarchical refinement, goal orientation, stepwise improvement and team orientation*, three different levels of abstraction are distinguished as illustrated in Fig. 2.

2.1 Activity Management

At the highest level of abstraction (*administration/cooperation level*), the more creative and administrative part of design work is reflected. The key concept at this level is the *design activity (DA)* which is the operational unit representing a particular design task or subtask. Relationships between DAs are essential for flexible cooperation and, therefore, are explicitly modelled, thus capturing task-splitting (*delegation*), exchange of design data (*usage*), and negotiation of design goals (*negotiation*). The inherent integrity constraints and semantics of these cooperation relationships are enforced by a central conceptual component, called *cooperation manager*.

Looking inside a DA reveals the *design control level*. There, the organization of the particular actions to be performed in order to fulfil a certain (partial) design task and corresponding control/data flow is the subject of consideration (*design flow*). Usually, these actions (design operations) are design tool applications. The correctness of tool invocations is guaranteed by a conceptual component, called *design manager*. Design (object) states are captured by means of a versioning model (cf. Sect. 2.2).

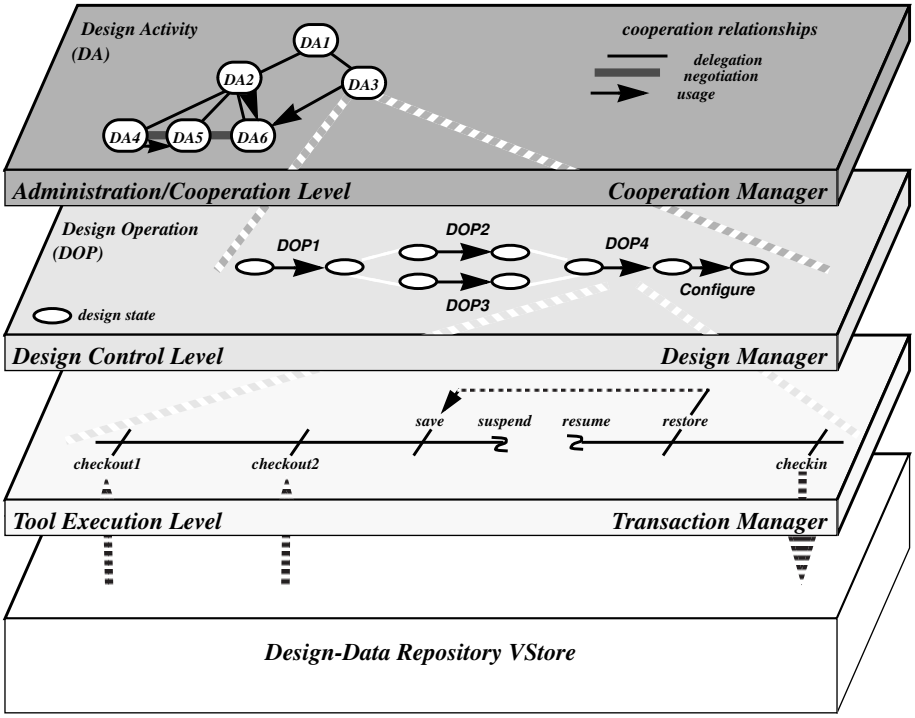

Fig 1: CONCORD Overview

From the viewpoint of the design-data repository, a design operation is a long transaction processing design object versions in a checkout-work-checkin cycle (see *tool execution level*). The derivation of schema-consistent and persistent design object versions is guaranteed, again, by a central conceptual component, called *transaction manager*.

2.2 The Generic Design-Data Repository VStore

The three levels discussed above are conceptually located on top of the design-data repository VStore. VStore completely covers the data management needs of CONCORD by implementing the versioning model described in [13]. This versioning model supports explicit and apparent versions of complex structured design data and offers a descriptive query language for retrieval, insert and update of versions and corresponding configurations. Additionally, basic services are provided by VStore, which have especially been designed and implemented for supporting the CONCORD activity managers.

Obviously, versioning is an essential requirement of design applications. Therefore, the CONCORD activity levels are (conceptually) located on top of VStore and each activity manager heavily relies on the versioning mechanisms provided by VStore. At the tool execution level adequate transaction facilities, e. g. concurrency control mechanisms, are provided to protect version manipulations. Also, the design control level depends on the versioning concepts, because design object states get certain design qualities by getting associated with certain design states. Finally, at the highest activity level versions are to be considered as the subjects of cooperation (control). Due to this central role of versioning, we decided to use the generative provisioning of versioning facilities as an example for presenting our SERUM approach (see Sect. 4).

3. The SERUM Approach

The *SFB 501*, our SERUM project is part of, aims at the development of large systems with generic methods. Obviously, in large development processes a shared repository should be used in order to support cooperation of developers and reuse of design. A suitable repository has to support all the tools applied in the different steps of the development process. This comprises both, managing product data related to a current system under development as well as managing process data, i. e., information related to the development process itself. The larger the development project is, the more multifaceted are the requirements concerning the repository. Furthermore, the variety of tools needed in the various phases of the software development process leads to sometimes contradictory demands. In our opinion, it is hard, if possible at all, to fulfil all these differing needs by providing a single 'stand-alone' repository system.

Consequently, we want to build a family of repository managers, each providing customized access to the central database for a particular set of design tools. Together, these repository managers establish the actual repository reflecting all the needs of the entire domain (domain repository). Because of similarities, we do not want to build each of them from the scratch. Instead, we want to exploit reuse techniques.

The key idea of SERUM is to generate customized repository managers from a high-level specification of its application-specific properties. Our approach defines a sequence of transformations and supports the corresponding steps towards an executable repository manager with suitable tools. The idea of reuse is put into practice in form of process reuse and reuse of building blocks. Process reuse is embodied in the SERUM process itself and the SERUM tools. Building blocks are reused throughout the process for completing the specification and for composing/generating the repository manager. Fig. 2 gives a graphical illustration of the SERUM process for customizing a repository manager. The generic methods applied in each step of the SERUM process are based on a set of pre-defined frameworks.

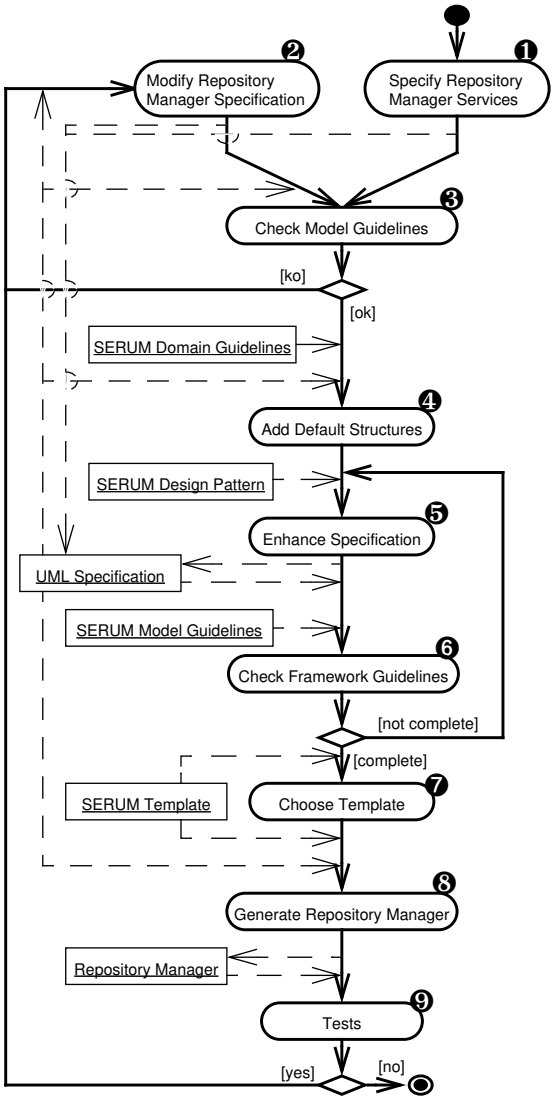

Fig 2: SERUM Process

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 three parts, the framework ***guidelines***, the technology-independent ***design patterns*** and the technology-dependent ***templates***. In the following, we examine SERUM frameworks and their components in a little more detail, especially regarding their usage within the SERUM process (see Fig. 2).

If a new repository manager needs to be build or an existing one must be modified, the ***repository designer***[2] first has to give (❶) (or modify (❷), respectively) a UML[3] specification of the (application-specific aspects of the) services to be provided by this repository manager[4]. Obviously, the transformation process cannot start from an arbitrary UML specification. Thus, ***domain guidelines*** restrict the design space. These guidelines are constraints upon the initial UML specification provided by the repository designer. They are specified in OCL (Object Constraint Language [19], part of UML). Domain guidelines depend on the (kind of) repository services the repository designer wants to specify, e. g., a versioning service. A SERUM tool, the SERUM model enhancer (SME), checks the guidelines (❸). This check is especially necessary, if the repository designer started with a completely new specification (❶) and did not select an existing one (❷) in order to adapt it to his special needs. The repository designer has to improve the initial model until all domain guidelines are fulfilled. Regarding versioning, the repository designer specifies the product data objects and their relationships. The resulting UML model is called a product data model. Since the versioning framework is not able to deal with multiple inheritance, one rule of the guidelines checks whether or not the product data model only contains single inheritance.

Next, the SME adds the ***default structures*** (❹). Default structures are, for example, abstract base classes with default attributes, relationships and behaviour. SERUM tools and reused building blocks depend on the existence of these base structures. Therefore, it is enforced by a guideline that these default structures exist in the specification, if a particular framework is used.

The next step is to enhance the UML model specified by the repository designer and to complete the specification automatically by exploiting domain knowledge (❺). This step is also supported by the SME. The UML model is altered, following modification rules. These rules are described as scripts, which are part of the SERUM design pattern definition (see example in Fig. 3).

In [6] 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 initially specified by a repository designer. A SERUM design pattern consists of two parts, an informal one helping the repository designer to select the pattern, and a formal one, the design pattern script, used to refine user specifications. Due to the second part, SERUM design patterns are much more formal than those presented in [6]. In this paper, we want to concentrate on the second part 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.

---

2. The repository designer is the person specifying application-specific semantics needed by SERUM to generate repository manager services.
3. Unified Modelling Language [21, 20].
4. Note that, as described in the following paragraphs, the application-independent aspects of the specification will be automatically incorporated by the subsequent steps of the SERUM process.

- *Preconditions*
  If a SERUM design pattern is applied in order to enhance a UML model, this model has to fulfil preconditions. The SME checks these preconditions, which we specify in our design-pattern definition script using OCL constraints.
- *Constraints*
  Usually, several design patterns are applied. Hence, it must be avoided that a subsequent design pattern script can destroy the changes made by an earlier one. Therefore, a design pattern defines some constraints, which have to be valid after the repository designer has finished his work. Constraints are expressed in OCL.
- *Model Evolution Operations*
  Applying a design pattern to a 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.

```
1.  begin define framework “VersionManagement”
2.    begin domain guidelines
3.      ClassClass.allClasses()
4.            ->forAll( c
5.                  | c.supertype().size = 1;
6.      // additional domain guidelines
7.    end domain guidelines
8.    begin default structures
9.      begin mdl
10.           (   object Class “ProductDataObject”
11.             // definitions
12.           )
13.     end mdl
14.     // additional default structures
15.   end default structures
16.   begin model guidelines
17.     // additional model guidelines
18.   end model guidelines
19.   begin define pattern “ProductDataObject”
20.     begin   parameters
21.       ClassUList          : aClasses;
22.     end parameters
23.     begin preconditions
24.       aClasses->forAll( c | exists(“Class”,c.name ));
25.     end preconditions
26.     begin constraints
27.       aClasses
28.       ->forAll( c
29.             | c.allSupertypes
30.                   ->exists( s
31.                     | s.name()=“ProductDataObject” ));
32.     end constraints
33.     begin alter model
34.       for ( int i; i < aClasses.length; i++ )
35.       {
36.         GeneralizationClass.create_generalization
37.               (aClasses[i].name(),“ProductDataObject” );
38.       }
39.     end alter model
40.   end define pattern
41.   // additional framework components
42. end define framework
```

Fig 3: Framework Definition Script

For example, all versions managed by our versioning framework have to be instances of a particular class `ProductDataObject` (see Sect. 4). This rule is specified by a design pattern which inserts an inheritance relationship between this root class and each (product data) class specified. The (product data) classes themselves are incorporated as base classes. Everything, the repository designer has to do, is to enumerate all classes he wants to be versionable. He may do this either interactively or by using a scripting language. The process of applying design patterns to a UML model can be seen as a customization process, in which the framework is adjusted to the needs the

repository designer wants to be fulfilled. It results in a UML specification of a repository service, which is stored in the SERUM UML repository.

Now, the UML specification has to be checked again (❻). This time, it has to be proven, whether or not the specification is complete in the sense that the SERUM repository generator can generate a repository manager from it, i. e., the ***model guidelines*** are checked. If model guidelines are violated, the repository designer has to further adapt the initial specification or to continue with the process of enhancing the specification.

While the design patterns provide the implementation-independent parts of a SERUM framework, SERUM templates represent the implementation-specific parts. This approach enables reuse of design solutions without being technology-dependent. The idea of a SERUM template is to bring together a configuration of components with two main properties. First, these components are able to work together in a repository manager. Hence, SRG does not need to know which configurations are useful. Second, the configuration of components will lead to a repository manager with well-known properties, regarding non-functional requirements. If the repository designer selects a template, then he also determines the properties of the resulting repository manager. Each component is allowed to be included in many templates. The components of a SERUM template can be divided into three main groups:

- *Code Generation Rules*
  As we have mentioned before, several programming languages need to be supported. We do not want to embody the mapping rules for every target language into the code of the repository generator, because this would restrict the repository generator's usability too much. Therefore, the SRG must be able to generate code following code generation rules.
- *Source Code Templates*
  The concept of source code templates is well-known from programming languages, especially C++. Unfortunately, not all programming languages provide this concept. Additionally, the expansion rules are not flexible enough, mostly limited to simple term rewriting mechanisms. Hence, we want to integrate functionality into the repository generator which allows for expanding source code templates.
- *'Ready-To-Use'-Components*
  'Ready-to-use' components can be applied without generating or modifying code. If such components need to be customized, this is done via parameters. The implementations of the abstract base classes (if they result from expanding a source code template) are examples of 'ready-to-use' components. Also, repository servers and client caches are usually used without code modifications. Each component may be part of more than one template, but a template provides the particular set of parameters, as for example the caching strategy, the repository generator needs for customizing the components. Other parameters concerning non-functional properties are set by the repository designer.

Each framework may contain several templates, where each template includes the components for generating an executable repository manager. The repository designer has to chose a template which fulfils the needs (❼) he has in mind. In our versioning example, he may want to access the repository by a Java [1] API. Hence, he would chose a template supporting Java APIs. Consequently, the template includes an implementation of the abstract class `ProductDataObject` (see above) in Java.

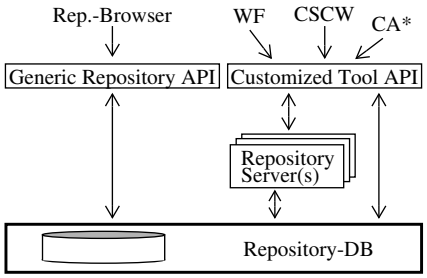

Fig 4: Repository Manager

The UML specification serves as input for the SERUM ***repository generator*** (SRG). Based on the 'half-fabricated' components provided by the SERUM templates, the SRG generates the new repository manager (❸). It produces a ***repository database schema***, a ***customized tool API*** and (several) ***repository servers*** which together establish the new repository manager (Fig. 4). In this step the application logic needs to be integrated into the repository manager. The repository designer has to provide implementations for those methods, which neither exist as source code templates nor can be generated from input specification. Additionally, the repository generator stores information about the generation process in the SERUM meta-database. This information enables generic tools, as for example browsers, to access repository data via the ***generic repository API (GRAPI)***.

Afterwards, the new repository manager is tested (❾). If changes are necessary, the whole process restarts with the modification of the specification (❷).

Fig. 5 illustrates the tools involved in the SERUM process. The whole process is based on the SERUM repository, which stores all information needed for generating the repository manager. It consists of three main components, the UML repository, the framework repository and the meta-database. UML specifications are stored in the UML repository which manages versions of UML models. In the framework repository the definitions of the frameworks, i. e. the definitions of the patterns and the components of the templates, are stored. The third component is the meta-database. It is used to store intermediate data processed in the steps of the SERUM process. If, for example, a design pattern script is executed by the SME, then identifiers of the initial version of the UML model, the design pattern and the resulting version are stored here. Additionally, the mapping information needed by the GRAPI is stored in the meta-database. An API, called the *repository definition API*, provides access to the repository for the SERUM tools, mainly the SME and the SRG.

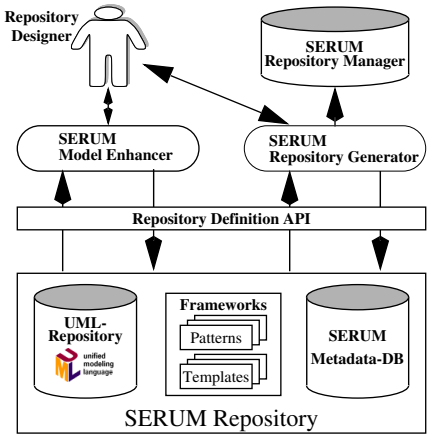

Fig 5: SERUM Tools

4. Providing Versioning Services in SERUM

As already motivated previously, 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). Thus, the overall tailoring process actually consists of two steps[5]:

(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, an ***adapted versioning framework (AVF)*** is derived, which, in turn, represents the versioning data model[6] adequately supporting the application(s) in mind. The basic versioning framework [17] consists of structural elements as

5. It will turn out that the second step refers to the process described in the previous section.

well as a communication infrastructure. Adapting the BVF means refining both the structural elements and the communication in between them, e. g., propagation of actions along relationships, in order to capture certain versioning semantics. We do not want to detail this first step in this paper and, therefore, refer to [16, 17] discussing it in more detail.

(2) *Applying the adapted versioning framework*
The AVF resulting from step (1) and a **product data model**[7] **(PDM)** given for a certain application are 'melted' in a way delivering a customized versioning repository manager. This is exactly the process which has been described at an abstract level in the previous section. In the following, we want to concentrate on this (second) step by discussing it concretely for the example of versioning.

The following steps (see Fig. 6) have to be performed in order to apply an AVF, i. e., the second step mentioned above can be refined as follows:

(1) Choose an AVF from the SERUM Frameworks (or adapt a new one from the BVF and afterwards choose this one).

(2) Define a product data model (PDM) without considering versioning aspects.

(3) Identify the units of versioning in the PDM, and, thereby, specify the ***versioning management infrastructure (VMI)***. The specification depends on the chosen AVF.

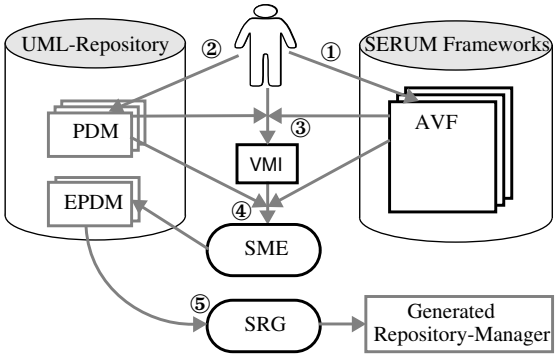

Fig 6: Process of applying an AVF

(4) Pass the PDM and the VMI to the ***SERUM model Enhancer (SME)*** [16]. From PDM and VMI, the SME generates an enhanced product data model (EPDM) which, in contrast to the PDM, also captures versioning aspects.

(5) Pass the EPDM to the SERUM repository generator (SRG) which generates corresponding repository manager functionality.

In the following we want to consider each of the mentioned steps in a little more detail.

Step (1): Choosing an AVF

The AVF should be chosen before defining the PDM, because the AVF determines the model guidelines to be fulfilled by the PDM (see ❸ in Fig. 2) before it may be melted with the AVF. Therefore, the repository designer should be aware of the guidelines when specifying the PDM.

Step (2): Defining the PDM

Now, the repository designer defines the PDM, which, first of all, does not take any versioning aspects into account. Like BVF and AVF the PDM has to be specified in UML notation, because the UML metamodel [21] is used as SERUM metamodel [16].

---

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

To clarify the process of applying an AVF, we use a simplified example from a software development environment. Starting from the sample PDM shown in Fig. 7 (UML notation), we want to establish a new repository manager. The example considers applications, which may be divided into sub-applications in order to ease administration. A (sub-)application may contain several packages, whereas each package may belong to several applications.

Step (3): Specifying the VMI

Next, the PDM is superimposed by a VMI, i. e., the units of versioning are defined. Fig. 8 shows a sample result of applying this step to the PDM illustrated in Fig. 7. Fig. 8 illustrates two sample units of versioning, `VS_Application` and `VS_Package`, also called versionable structures (VS). In this example, each VS contains exactly one PDM class (`Application` resp. `Package`).

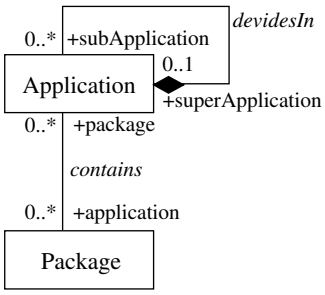

Fig 7: Sample PDM

Note that a VS generally may include several PDM classes and PDM associations. Since VSs can be considered as being units of versioning, each instance of a VS[8] class can have several versions, and each of these versions is a structure consisting of instances of the corresponding PDM classes. Like the PDM classes the VS classes can have associations, which refine the PDM associations. There are also associations at the version layer, which refine the VS associations; but these associations as well as the version classes are generated (cf. step 4 below) and do not have to be defined in the VMI. In our example, there are two VS associations, `vs_contains` and `vs_devidesIn`, which refine the PDM associations `contains` resp. `devidesIn`.

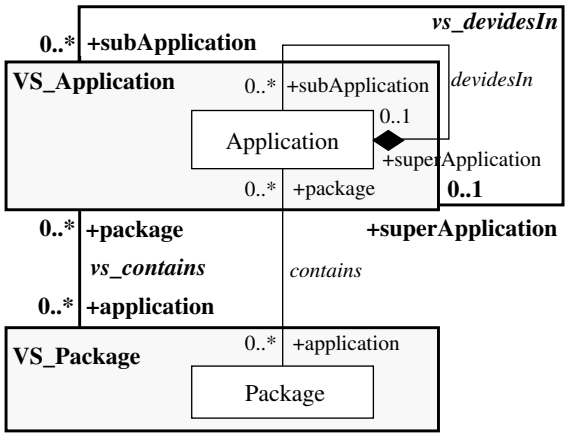

Fig 8: PDM with the VMI

In order to specify the VMI in SERUM we use a version definition language (VDL). The basic VDL is defined in the BVF and can be adapted (if necessary) in the AVF. The VDL statements corresponding to the VMI of Fig. 8 are shown in Fig. 9. The first statement (lines 1-7) defines the VS class `VS_Application`. By such a VS definition (see also lines 18-24) PDM classes (and PDM associations) are associated with a VS. Additionally, it is specified, how many instances of the corresponding PDM classes or PDM associations may belong to a single version of a VS instance (with [n..m] INSTANCES) and whether or not these instances may belong to (versions of) other VS instances. In line 4, it is specified, that an `Application` instance can only belong to a single version, no matter to which VS instance this version belongs to. The associations among the VSs are defined in line 8-17 and line 25-34. Cardinality restrictions do not have to be specified, because they can be derived from the PDM.

8. VS is a class of the AVF, and each VS class inherits the properties of VS.

Step (4): Enhancing the PDM

In this step, the enhanced PDM is generated from the original PDM, the chosen AVF, and the VMI specified in step 3. This task is performed by the SME. At first, the SME checks the PDM against the model guidelines of the AVF (see ❸ in Fig. 2). If the model guidelines are fulfilled, then the default structures of the AVF are added to the PDM. Usually, the default structures comprise abstract base classes for organizational purposes (a VS superclass, PDM superclasses and PDM 'superassociations') and for the behaviour (SignalHandler [16]). Afterwards, the PDM is enhanced by applying the VDL compiler, which, in turn, exploits the SERUM design patterns associated with the AVF to enhance the PDM. In Fig. 10 an excerpt of the resulting enhanced PDM is illustrated. Only the structural classes for versioning are shown; inheritance hierarchy and communication classes (SignalHandler, etc.) are omitted due to complexity. Furthermore, OCL constraints are part of the enhanced PDM, but are not considered in Fig. 10.

```
1.  DEFINE VERSIONABLE STRUCTURE
2.    CLASSES(
3.        Application WITH [0..1] INSTANCES
4.          AND EXCLUSIVE OWNERSHIP BY ONE VERSION)
5.    WITH
6.        VS_NAME IS VS_Application
7.        V_NAME IS V_Application
8.  DEFINE LINK REFINEMENT
9.    OF BASELINK devidesIn
10.     BETWEEN Application AS superApplication
11.           AND Application AS subApplication
12.   AS VERSIONLINK
13.     BETWEEN VS_Application AS superApplication
14.           AND VS_Application AS subApplication
15.   WITH
16.       VS_NAME IS vs_devidesIn
17.       V_NAME IS v_devidesIn
18. DEFINE VERSIONABLE STRUCTURE
19.   CLASSES(
20.       Package WITH [0..1] INSTANCES
21.         AND EXCLUSIVE OWNERSHIP BY ONE VERSION)
22.   WITH
23.       VS_NAME IS VS_Package
24.       V_NAME IS V_Package
25. DEFINE LINK REFINEMENT
26.   OF BASELINK contains
27.     BETWEEN Application AS application
28.           AND Package AS package
29.   AS VERSIONLINK
30.     BETWEEN VS_Application AS application
31.           AND VS_Package AS package
32.   WITH
33.       VS_NAME IS vs_classes
34.       V_NAME IS v_classes
```

Fig 9: VDL Skript

Note that basically the enhanced PDM resulting from this step can be further manipulated by applying more enhancement steps in order to incorporate features reflected by different SERUM frameworks. Since these enhancements are similar to the one described above, we do not want to detail this point.

Step (5): Generating the Repository Manager

In this step, the repository manager is generated by the SRG from the enhanced PDM resulting from step 4. But before, the enhanced PDM has to be checked against the framework guidelines (see ❻ in Fig. 2). If the guidelines are fulfilled, the repository designer may choose from the set

of available SERUM templates reflecting different technology demands. Depending on the chosen templates, the SRG generates the new repository manager (see ❽ in Fig. 2). Due to space restrictions, we cannot give a detailed view to the results of such an SRG application and, therefore, have to refer to [16] giving concrete examples of ORDBMS schema structures including user-defined functions for the manipulation of versioned data, functions supporting checkin/checkout and manipulation of buffered versions, and API functions for calling all the mentioned manipulation functions from programming languages like Java.

Obviously, the data management facilities needed by CONCORD applications can be provided as described in this section.

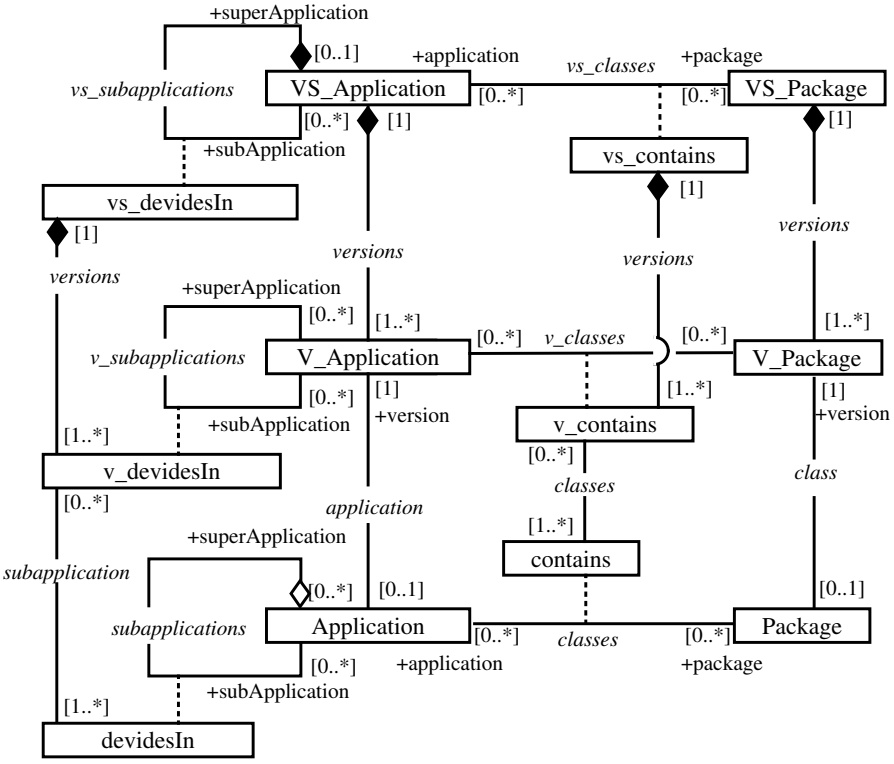

Fig 10: Enhanced PDM

5. Conclusions and Outlook

In this paper, we have discussed the potential of SERUM, a generative approach for accomplishing repository manager functionality, by considering the demands of cooperative design applications.

The SERUM approach, in general, is beneficial for the following reasons:

- The repository designer only has to model application-specific aspects of the repository manager service, he wants to realize. Corresponding specifications may be expressed in UML.
- Pre-defined, technology-independent design patterns relieve repository designers by automatically enhancing UML models by application-independent aspects.

- Repository manager functionality (code) is automatically generated/composed from the customized UML model by applying technology-dependent design templates.

As a representative design model, we have chosen to consider the CONCORD model capturing the dynamics of design applications by integrating version management as well as sophisticated activity management mechanisms. In order to enable applications to use the CONCORD functionality, a repository is needed providing basic data management as well as activity management services. We have given a detailed description on how version management facilities tailored to the needs of a distinguished application can be generated. Thus, w. r. t. versioning we could show an efficient way of generatively providing repository manager functionality, which before had to be implemented completely manually. We think that similar (generative) customization techniques may be applied to provide basic services for activity management, too. Furthermore, we have argued that UML and the extensibility property of new object-relational database technology is very helpful in this concern.

The discussion has shown that repository facilities can be decomposed in basic services, which, at their core, are application-independent and, therefore, can be provided by means of what is called SERUM frameworks. Certainly, we have to admit that we currently cannot give a complete list of basic components and a prove that all thinkable cooperation models can be provided in SERUM. Nevertheless, we think that the SERUM way of providing repository manager functionality is very promising, since it puts effectively into practice the basic idea of reusing design artifacts [14, 18]. The extensive application of reuse in SERUM considerably contributes to the goals of SERUM, which are providing a basic infrastructure for assisting and relieving repository designers in creating tailored repository manager functionality. Reuse is not only the foundation for the tailoring process, but also simplifies this process for the user (repository designer) by hiding details, reduces error probability, and helps to capture the complexity of the software generation step.

As future work, we intend to clearly identify all domain sections to which the mentioned SERUM process of customization may be applied. Especially the application-independent (core) functionality contributing to customizing and generating activity management components have to be identified and provided as SERUM components. This also requires to define reasonable points of customization as well as rules and user guidelines ensuring that the different repository managers generated within SERUM fit together and, thus, may collaborate appropriately.

Another point of interest is to completely exploit the extensibility potential of ORDBMS technology, e. g., also using features such as adapted index structures or controlled access to externally stored data via SQL within SERUM.

Finally, we plan to evaluate the overall SERUM approach in software development applications.

Literature

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

[2] Austern, M. H.: Generic Programming and the Stl : Using and Extending the C++ Standard Template Library, Addison-Wesley, Oct. 1998

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

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

[5] Conradi, R., Westfechtel, B.: Version Models for Software Configuration Management, ACM Computing Surveys 30:2, 1998, pp. 232-282.

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

[7] Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques, Morgan Kaufmann Publ., San Mateo, CA, 1993.

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

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

[10] Katz, R.: Towards a Unified Framework for Version Modeling in Engineering Databases, ACM Computing Surveys 22:4, 1990, pp. 375-408.

[11] Katz, S. Richter, C. A., The, K.: PARIS: A System for Reusing Partially Interpreted Schemas, 9th International Conf. on Software Engineering, IEEE Computer Society Press, 1987, pp. 377-385

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

[13] Käfer, W., Schöning, H.: Mapping a Version Model to a Complex Object Data Model, Proc. 8th Int. Conf. on Data Engineering, Tempe, Arizona, 1992.

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

[15] 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.

[16] Mahnke, W., Ritter, N., Steiert, H.-P.: Towards Generating Object-Relational Software Engineering Repositories, Proc. Datenbanken in Büro, Technik und Wissenschaft (BTW'99), Freiburg, Germany, March 1999.

[17] Mahnke, W., Ritter, N., Steiert, H.-P.: A basic versioning framework for SERUM, Technical Report, SFB 501, Dept. of Computer Science, Univ. of Kaiserslautern, 1998.

[18] Mili, H., Mili, F., Mili, A.: Reusing Software: Issues and Research Directions, IEEE Transactions on Software Engineering 21:6, 1995, pp. 528-562

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

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

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

[22] Rammig, F. J., Steinmüller, B.: Frameworks and Design Environments, Informatik-Spektrum 15:1, 1992, pp. 33-43, in german.

[23] Ritter, N., Mitschang, B., Härder, T., Gesmann, M., Schöning, H.: Capturing Design Dynamics - The CONCORD Approach, Proc. 10th. Int. Conf. on Data Engineering, Houston, Texas, Feb. 1994, pp. 440-451.

[24] Ritter, N.: The C3-Locking Protocol - A Concurrency Control Mechanism for Design Environments, Proc. STAK ' Rechnergestützte Teamarbeit', Munich, March 1996, pp. 95-110.

[25] Ritter, N., Mitschang, B., Härder, T.: Conflict Management in CONCORD, Proc. 6th. Int. Conf. on Data and Knowledge Engineering Systems for Manufacturing and Engineering (DKSME), Tempe, Arizona, Oct. 1996, pp. 81-100.

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

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

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