GENERATING VERSIONING FACILITIES FOR A DESIGN-DATA REPOSITORY SUPPORTING COOPERATIVE APPLICATIONS[1]

THEO HÄRDER, WOLFGANG MAHNKE, NORBERT RITTER and HANS-PETER 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

Received (to be inserted by Publisher)
Revised (to be inserted by Publisher)
Communicated by (Name of Editor)

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. Versioning comprises the specification, storage, and maintenance of versioned design objects whereas activity management is responsible for cooperation control, designflow management and management of design transactions processing versioned design objects. Regarding these issues (version and activity management) repository technology, as we think, should not only provide predefined services, but should be flexible enough to reflect different application needs. For that reason, we propose to provide repository managers by generic methods, i. e., by generating the corresponding functionality. In this paper, we consider a representative cooperation model, which is based on versioning services, in order to identify the major data manipulation and activity control needs of cooperative design applications. We will focus on the data manipulation needs by introducing our generative approach for customizing versioning facilities. Additionally, we will outline our ideas of applying a generative approach also for the provision of tailored activity control services. Thus, the paper wants 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 and, thereby, applications do not have to be forced to deal with systems only providing predefined services.

Keywords: Repositories, Cooperation, Versioning, Reuse, Generic Methods, ORDBMS.

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

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”.

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 by amending user specifications. Before introducing our approach, we first want to mention the basic services a repository manager has to provide in order to serve cooperative design applications properly.

1.1 Repositories

In the literature, two kinds of technology can be found which aim at an adequate support of design applications: frameworks [1, 2, 3] and repositories [4, 5, 6]. 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 of supporting the (special) design 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 to both, system providers and system users. In simple words: we want to provide a framework allowing the *generation* of repository managers.

In [4], 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 value-added services [4]: *checkout/checkin, version control, configuration control, notification, context management and workflow control*.

1.2 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, which are really important for that class of applications. Obviously, repository managers for design applications have to fulfil requirements w. r. t. both data management and activity management.

Data management requirements comprise versioning facilities. Complex structured design data is 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 design goal. Third, versioning allows for configuring complex design objects from versions of simpler 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, (design) data processing must be encapsulated by (design) transactions in order to avoid inconsistencies. Beside general transaction management facilities [7], design transaction management especially requires a concurrency control mechanism, which is well suited for processing versions, and integrity maintenance mechanisms allowing the specification of integrity conditions in terms of versions and configurations and to check these conditions at appropriate points in time. The challenge here is, to equip these mechanisms (for concurrency control and integrity maintenance) with adequate conflict management services [8], since blocking a transaction in the case of a data access conflict, as performed by traditional concurrency control protocols, is not acceptable in design applications due to the usually long duration and the interactive character of design transactions.

Besides transaction management, activity management, as we see it, comprises designflow management and cooperation control. These two fields mainly deal with project coordination and administration, but, definitely, are also as closely connected to data processing as transactions are. Designflow management facilities are needed to pre-plan the steps to be performed by groups of designers or single designers in order to reach their respective design goals. Thus, design flows allow reuse and, thereby, optimization of processes, which have been successful in the past, and relieve designers as well as project leaders. The challenge of design flow management is that design applications cannot be completely planned in advance, as, for example, business processes can. Thus, designflow management has to provide facilities for dealing with ad-hoc phases.

Cooperation control facilities have to aim at adjusting the visibility of usually (w. r. t. the corresponding design goal) preliminary design information. Regarding the huge amount of data created during a design process, data associated with a certain design task should only become visible (and, thereby, accessible) to cooperating tasks, if it has a (by some means) stable design quality and, thus, may help cooperating designers to fulfil their own tasks.

A design model reflecting all the needs mentioned so far in an integrated manner is the CONCORD model [9, 8]. Since CONCORD comprehensively covers all relevant needs of cooperative design applications, we decided to consider it as a representative allowing us 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.

1.3 Genericity

Usually repository manager services are as generic as database system services are. Let us consider versioning facilities as an example of generic services. Usually a repository manager implements a given versioning model offering a predefined set of data structures and generic operations. As discussed in [10] there are very many facets which versioning models may differ in. We think, many of the different concepts which lead to many different version models are very application-specific. Consequently, a generic version model cannot support *all* applications properly, but serves some more and others 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, 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.

1.4 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. This discussion is supposed to identify the services a repository manager has to provide in order to be well suited for cooperative design applications. For that purpose, we discuss the services provided by an earlier implementation of a repository (named VStore, see Sect. 2.4), which has been used as an implementation platform for the CONCORD prototype. VStore has especially been designed and implemented (manually) for serving as a CONCORD repository, and, thus, provides generic repository services supporting CONCORD.

Our goal is to show the following. First of all, it is not necessary to completely implement such a repository (like VStore), because it is possible to provide equivalent services with generic methods. Second, a generative approach (like SERUM) is able to support a whole family of systems which VStore is only one element of. Third, the SERUM process of tailoring repository manager functions can be automated to a considerable extent and can reasonably be controlled by users. 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. As we will see, our approach substantially benefits from the use of object-relational technology. Sect. 5 outlines how access to versions is controlled in CONCORD. In Sect. 6 we discuss how the basic ideas of SERUM can also be applied in the area of activity management. Sect. 8 concludes the paper and gives an outlook to future work.

2. Overview of CONCORD

The CONCORD (CONtrolling COopeRation in Design environments) model [9] 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 roughly illustrated in Fig. 1.

2.1 Administration/Cooperation Level (AC Level)

At the highest level of abstraction, the more creative and administrative part of design work is reflected. There, the focus is on the description and delegation of design tasks as well as on controlled cooperation among the design tasks. The key concept at this level is the *design activity (DA)*. A DA is the operational unit representing a particular design task or subtask. During the design process, a *DA hierarchy* can be dynamically constructed resembling a hierarchy of concurrently active tasks. All relationships between DAs essential for flexible cooperation are explicitly modelled, thus capturing task-splitting (cooperation relationship type *delegation*), exchange of design data (cooperation relationship type *usage*), and negotiation of design goals (cooperation relationship type *negotiation*). The inherent integrity constraints and semantics of these cooperation relationships are enforced by a central conceptual component, called a *cooperation manager*.

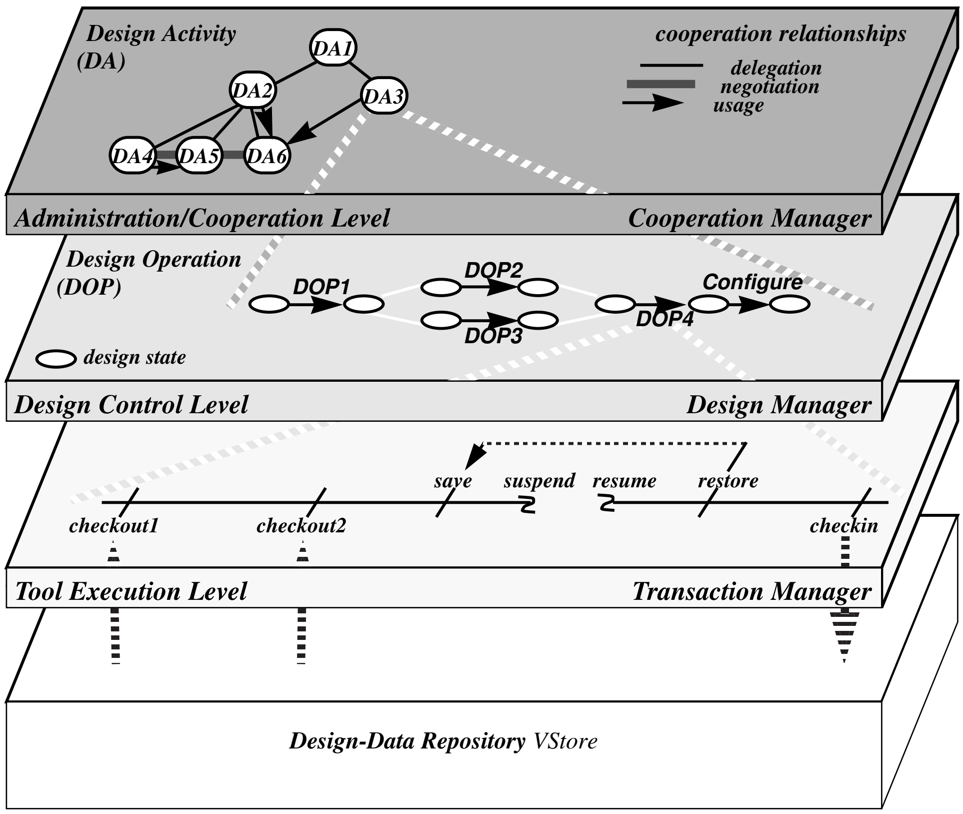


Fig 1: CONCORD-Overview

2.2 Design-Control Level (DC Level)

Looking inside a DA reveals the DC level. There, the organization of the particular actions to be performed in order to fulfil a certain (partial) design task is the subject of consideration (*design flow*). At this level, Fig. 1 shows an execution plan (state/transition graph) of a particular design activity. It models the *control/data flow* among several design actions performed within a DA. Usually, these actions are design tool applications. The operational unit for the execution of a design tool is the *design operation (DOP*). In order to control the actions within the scope of a single DA, but without restricting the designers' creativity, flexible mechanisms for specifying the design flow for a DA are provided. The correctness of tool invocations is guaranteed by a conceptual component, called a *design manager*. The design manager also provides for recoverable design-flow executions that are needed for level-specific and isolated failure handling. Design tools are applied to improve existing design states in order to finally reach a design state that completes the current (partial) design task. Design (object) states are captured by means of an object and version model (cf. Sect. 2.4).

2.3 Tool-Execution Level (TE Level)

From the viewpoint of the design-data repository, a DOP is a long transaction (see Fig. 1). A DOP has the properties of conventional transactions. Because of long duration, it is internally structured by save/restore and suspend/resume facilities to be able to rollback the design at the application level and to continue the design work after breaks. A DOP processes design object versions in three steps. First, the input versions are checked out from the integrated data repository, and cached in an object buffer at the workstation for efficiency reasons. Second, the design data is mapped to in-memory storage structures tailored to the application needs. It is processed by one or more design tools. Third, the finally derived new versions are propagated back to the data repository (checkin operation). The derivation of schema-consistent and persistent design object versions is guaranteed, again, by a central conceptual component, called a *transaction manager*. It is also responsible for the isolated execution of DOPs and for recoverable DOP executions that are, again, necessary for level-specific and isolated failure handling.

2.4 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 [11]. 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. These basic services are briefly discussed in the following.

2.4.1 Transaction Management Services

Long design transactions are supported by an object-buffer-based processing concept as described above. A special concurrency control protocol [12] has been developed and implemented, taking the version semantics of the underlying data model into account, and thereby protecting version manipulations sufficiently, without being unnecessarily restrictive. Handling data access conflicts (between concurrent transactions) is done in a way which avoids blocking of design transactions. Similar (conflict management) principles have been chosen w. r. t. integrity maintenance, since integrity violations detected during checkin do not directly lead to the abort of the corresponding (long design) transaction, but initiate an exception handling process allowing the application to save as much of the work performed by this transaction as possible.

2.4.2 Basic Designflow Management Services

Supporting the designflow manager, VStore provides an open transaction management, i. e., the designflow manager may collaborate with the transaction manager in order to always be aware of the current design state. Additionally, special-purpose integrity constraints may be defined, expressing intermediate or final goals of local design tasks. Thus the designflow manager may delegate the checking of such constraints to the VStore integrity maintenance component at appropriate design states.

2.4.3 Basic Cooperation Control Services

Cooperation control is mainly based on groupware, but also exploits basic VStore services, at least as far as data-specific cooperation is regarded. This means that each (cooperating) design task (DA) is associated with a workspace of private design data. Only if cooperation relationships and corresponding protocols allow the cooperation with other tasks by making private data visible, meaningful portions (versions) of private design data become accessible to others. For such purposes, VStore provides sophisticated access control services to be exploited by the cooperation manager. Additionally, the cooperation manager may also cooperate with the integrity maintenance component of VStore to ensure that access to design data belonging to ‘foreign’ workspaces does not invalidate the design quality already reached.

3. The SERUM Approach

The *SFB 501*, of which our SERUM project is part, 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.

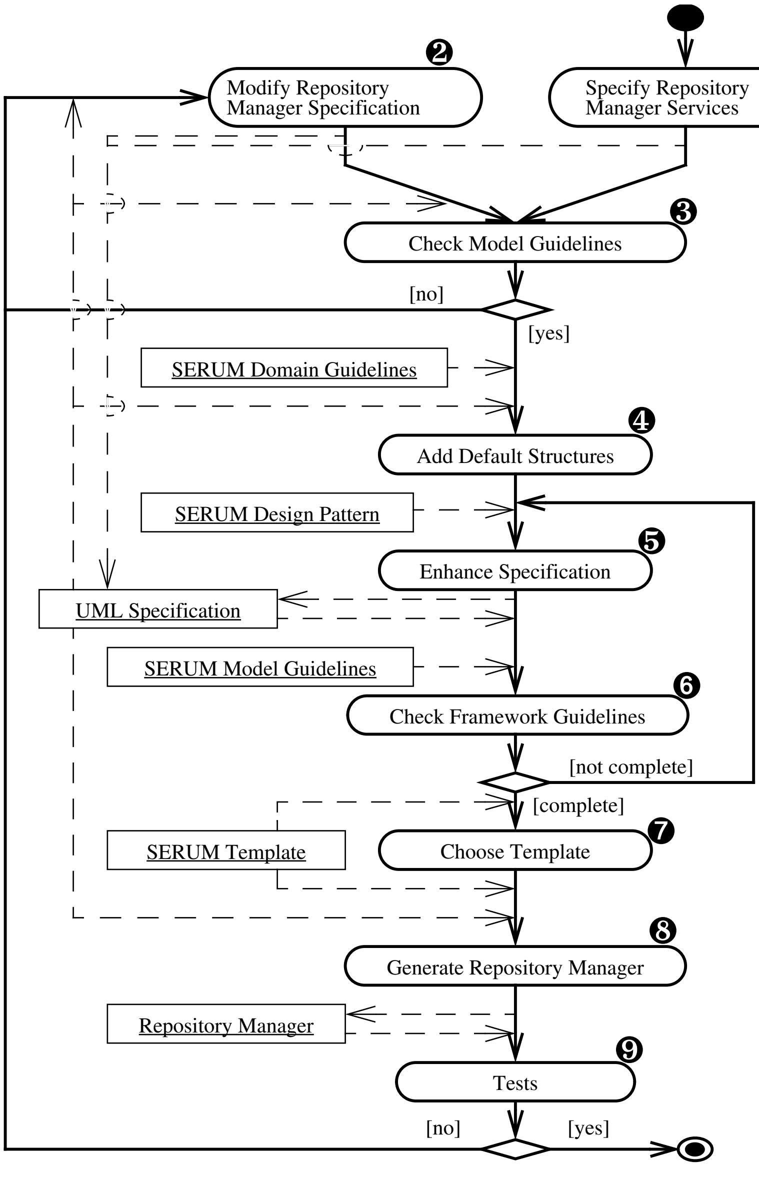


Fig 2: SERUM Process

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 they have a lot 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 the 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.

According to [13] "*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 built or an existing one 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].

This initial UML specification must observe ***domain guidelines***, which are specified in OCL[5]. Domain guidelines depend on the (kind of) repository services the repository designer wants to specify, e. g., a versioning service. For example, 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*** (PDM) only contains single inheritance. A SERUM tool, the SERUM ***model enhancer*** (SME), checks the guidelines (❸).

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.

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 [17] 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

2. The repository designer is the person specifying application-specific semantics needed by SERUM to generate repository manager services.
3. Unified Modelling Language [14, 15].
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.
5. Object Constraint Language [16], which is part of UML.

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 (see Fig. 3). Besides parameters (specifying the elements of the input model which are to be manipulated) and preconditions/constraints such a script contains *model evolution operations* performing the actual transformations, e. g., by enriching class definitions by new attributes, new relationships, new superclasses, new methods, etc. In our example (Fig. 3), we suppose that all versions managed by our versioning framework have to be instances of a particular class ProductDataObject (see Sect. 4). This rule is specified by the 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. All, 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 fulfilled. It results in a UML specification of a repository service, which is stored in the SERUM UML repository.

```
begin define pattern "ProductDataObject"
  begin parameters
    ClassUList  : aClasses;
  end parameters
  begin preconditions
    aClasses->forAll( c | exists("Class",c.name ));
  end preconditions
  begin constraints
    aClasses
    ->forAll( c
        | c.allSupertypes
             ->exists( s
               | s.name()="ProductDataObject" ));
  end constraints
  begin alter model
    for ( int i; i < aClasses.length; i++ )
    {
      GeneralizationClass.create_generalization
          (aClasses[i].name(),"ProductDataObject" );
    }
  end alter model
end define pattern
// additional framework components
end define framework
```

Fig 3: Framework Definition Script

Now, the UML specification has to be checked again (❻). It has to be proven, whether or not the specification is complete in the sense that the *SERUM* ***repository generator*** (SRG) can generate a repository manager from it, i. e., the ***model guidelines*** are checked.

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. As components a SERUM template may contain *code generation rules* giving additional flexibility w. r. t. programming languages, *source code templates* comparable to the mechanism known from the programming language C++, and *'ready-to-use' components* which 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 [18] 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.

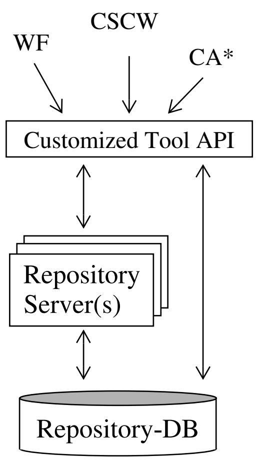


Fig 4: Repository Manager

The UML specification serves as input for the SRG. Based on the 'half-fabricated' components provided by the SERUM templates, the SRG generates the new repository manager (❽). It produces a ***repository database schema***, a ***customized tool API*** and (several) ***repository servers*** (application 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 the input specification. Additionally, the SRG stores information about the generation process in the SERUM meta-database.

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[6]:

(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[7] adequately supporting the application(s) in mind. The basic versioning framework [19] consists of structural elements as 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 [20, 19] which discuss it in more detail.

(2) Applying the adapted versioning framework

The AVF resulting from step (1) and a **product data model**[8] **(PDM)** given for a certain application are 'melted' in a way delivering a customized versioning repository manager. This second step corresponds with the process which has been described at an abstract level in the previous section. In the following, we want to concentrate on this step by discussing it concretely for the example of versioning; it can be further refined as follows:

(1) Choosing an AVF

It is useful to choose the AVF before defining the PDM, because the AVF determines the model guidelines which have 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.

(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.

To clarify the process of applying an AVF, we use a small and intentionally incomplete example from a software development environment (see sample PDM in Fig. 5). 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.

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

7. 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.

8. 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.

(3) Specifying the VMI

Next, the PDM is superimposed by a VMI, i. e., the units of versioning are defined. Fig. 6 shows a sample result of applying this step to the PDM illustrated in Fig. 5. Fig. 6 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`).

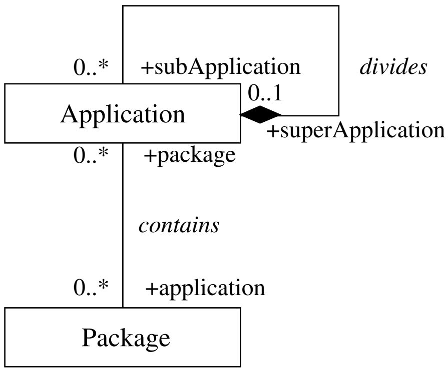


Fig 5: Sample PDM

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[9] 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_divides`, which refine the PDM associations `contains` resp. `divides`.

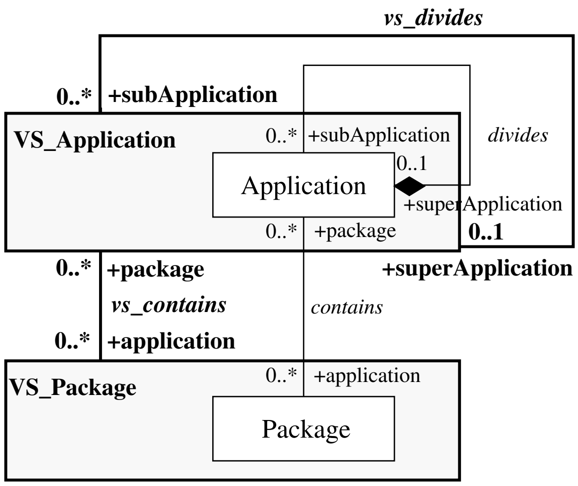


Fig 6: PDM with the VMI

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 in the AVF. The VDL statements corresponding to the VMI of Fig. 6 are shown in Fig. 7. The first statement defines the VS class `VS_Application`. By such a VS definition 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 `Ap-`

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

plication instance can only belong to a single version, no matter which VS instance this version belongs to. The associations among the VSs are defined 'DEFINE LINK REFINEMENT'-statements. Cardinality restrictions do not have to be specified, because they can be derived from the PDM.

```
DEFINE VERSIONABLE STRUCTURE
 CLASSES(
   Application WITH [0..1] INSTANCES
    AND EXCLUSIVE OWNERSHIP BY ONE VERSION)
 WITH VS_NAME IS VS_Application V_NAME IS V_Application

DEFINE LINK REFINEMENT
 OF BASELINK divides
  BETWEEN Application AS superApplication
     AND Application AS subApplication
 AS VERSIONLINK
  BETWEEN VS_Application AS superApplication
     AND VS_Application AS subApplication
 WITH VS_NAME IS vs_divides V_NAME IS v_divides

DEFINE VERSIONABLE STRUCTURE
 ... WITH VS_NAME IS VS_Package V_NAME IS V_Package

DEFINE LINK REFINEMENT
 OF BASELINK contains ...
```

Fig 7: VDL Script (Excerpt)

(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 [20]). 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. 8 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. 8.

Note that 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.

(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 [20] which gives concrete examples of ORDBMS schema structures. These include 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.

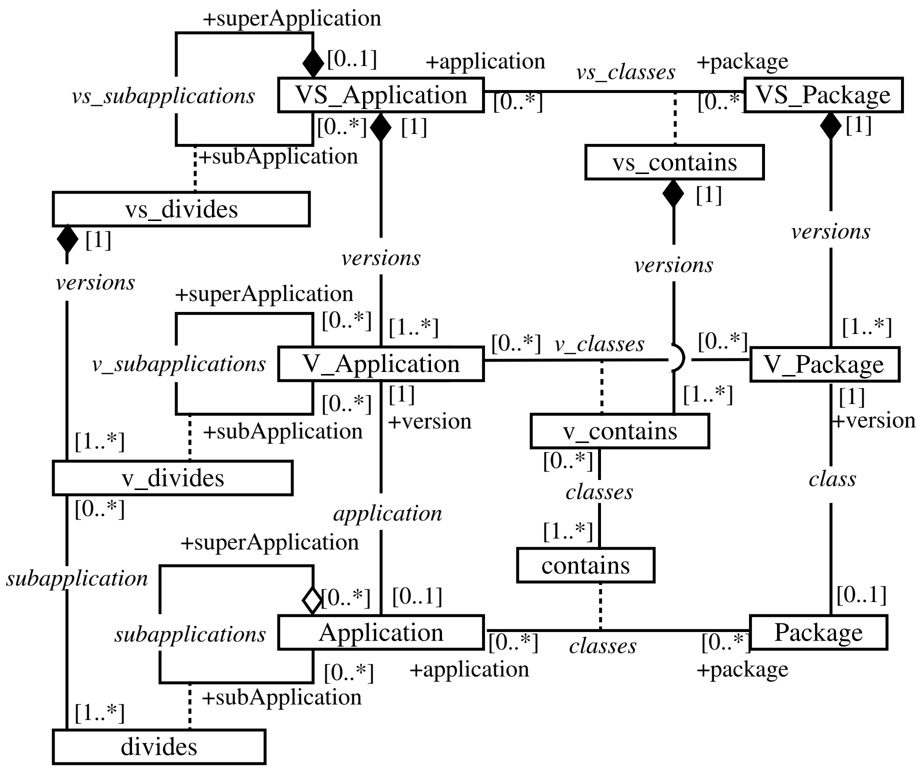


Fig 8: Enhanced PDM

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

5. Exploiting Generated Versioning Facilities in CONCORD

In this section, we describe how the version management facilities which can be generated by the mechanisms introduced in the previous section are facilitated by the CONCORD activity control mechanisms. For clarity purposes, we discuss the most important activity control aspect at each of the three CONCORD activity levels, respectively.

5.1 TE-Level: Controlling Concurrent Access to Versioned Design Data

Concurrency occurs, if parallel transactions (DOPs) want to access the same (local) data. In order to determine an adequate notion of correctness, we had to take into account, whether or not design tools do work cooperatively, and, if they do, whether or not the natural cooperative capabilities already provided by the versioning concepts are sufficient. After having examined VLSI design, mechanical engineering as well as software engineering, we came to the conclusion that tools do not directly cooperate. Furthermore, we realized that it is a natural way of processing that tools derive versions independently and that there are special tools for merging and configuring. This lead us to isolate DOPs, i. e. to exclude the well-known multiuser anomalies such as unrepeatable read [7]. To achieve the wanted degree of isolation, we developed the C^3-locking-protocol (C^3 stands for Concurrency Control in CONCORD) [12] which is an extension of the well known two-phase locking protocol (2PL) [7]. The major requirements were the following. The protocol has to provide an adequate protection for versioned data. Due to long duration, abort cannot be used as a general mechanism for solving conflicts (deadlocks). Furthermore, the protocol could take the inherent properties of design processing into account, e. g. high interactiveness.

Version Lock Mode	Internal Effect	External Effect
VS (version read)	reading the version;	reading the version, deriving a successor version;
VD (version derive)	reading the version, deriving a successor version;	reading the version, deriving a successor version;
VDX (version exclusive derive)	reading the version, deriving a successor version;	reading the version;
VX (version exclusive)	reading/updating the version, deriving a successor version;	no rights;
VXC (version exclusive convertible)	reading the version;	no (further) rights;

Table 1: Version Lock Modes

Table 1 illustrates the lock modes obtainable for versions. The column ‘Internal Effect’ lists the rights of the lock holding transaction and the column ‘External Effect’ correspondingly lists the remaining rights of others. Considering a single version and a request of a transaction to derive a successor version, we had to foresee different mechanisms. Provided, the derivation graph structure, the version is embedded in, allows the derivation of multiple successors, the system grants a VD lock, giving concurrent transactions the possibility to also check this version out for derivation. If, on the other hand, the derivation structure does not allow multiple successors, e. g. in the case of a list structure, then the VDX mode must be given.

We also have to explain the two exclusive modes. VX is self-explaining. The VXC mode reminds at the Update lock mode introduced to prevent conversion deadlocks on hot spots [7] (in the original meaning it is granted instead of a shared lock). In the C^3-protocol, however, this mode is used differently, as we will see in the following. In our checkout/checkin scenario, manipulations are carried out on a copy (stored within the object buffer) so that it is often appropriate to use the VXC mode (instead of an exclusive lock) and prevent from that point in time on further transactions from getting shared modes (VS, VD, VDX). So, there is a high probability that the initially active transactions holding shared locks will be committed until the considered transaction gets to its checkin step in which the VXC mode must be converted to VX.

Since the processing context of a design tool is usually known in advance and design flow can often be pre-planned, we found it acceptable to introduce some kind of preclaiming [GR93], which we called pre-specification of access mode, because it is not completely equivalent to preclaiming as we will see. Thus, with each query the application specifies which access mode it wants to get on the query's result set. Fig. 9 shows the pre-specification clauses and the corresponding compatibilities of modes.

lock held by a concurrent transaction
lock requested

		VS	VD	VDX	VXC	VX
SELECT VERSIONS FOR READ]	VS	+	+	+	-	-
SELECT VERSIONS FOR DERIVATION	VD	+	+	*	-	-
	VDX	+	*	-	-	-
SELECT VERSIONS FOR UPDATE	VXC	+	+	+	-	-
	VX	-	-	-	-	-

Fig 9: Lock-Mode Compatibilities

The major principles of the protocol are as follows. For each version in the query's result set, the lock manager checks whether or not the requested mode can be granted. If the requested mode is incompatible, the lock manager determines the highest possible mode (which could be granted on all elements) and offers it to the application. The application program now has the following possibilities: to accept the lower mode, to reject and get back control, or to wait synchronously or asynchronously. The drawback of this concept is that the application programmer has to deal with concurrency control aspects, but we think that this is restricted to an acceptable extent. For example, it can be appropriate to accept a derivation lock for a certain version instead of the originally requested update lock and not having to wait until other (usually long running) DOPs terminate. VXC modes are handled in a similar way. After a 'SELECT ... FOR UPDATE' request, the lock manager may offer a VXC lock, if there is a certain probability that the design tools which are currently holding incompatible, shared locks will be finished until the requesting tool gets to its checkin step.

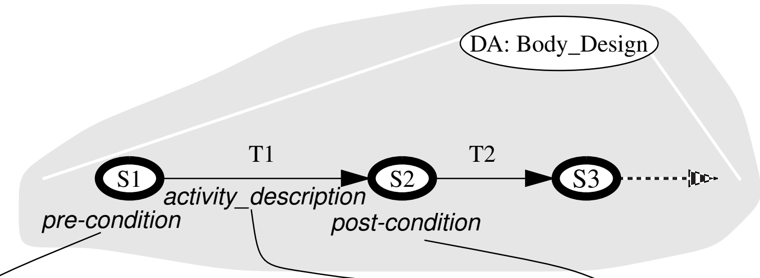
We just want to add, how deadlocks are handled and which correctness criterion is achieved, since these aspects directly contribute to the topic of conflict management in this system layer. Deadlocks may occur, since we allow DOPs to initiate several checkout steps. Nevertheless, deadlocks are expected to be very infrequent, since they may only occur, if transactions decide to wait synchronously. Additionally, due to the pre-specification, deadlocks can be detected very early, i. e. during the checkout phase, so that it is acceptable, to use abort as a resolution mechanism. Since the manipulation phase is expected to be the most time-consuming phase within DOP processing, a reiteration of checkout is not too expensive.

Using the above mentioned principles within a two-phase locking protocol ensures at least a 2.9 degree of isolation (cf. [7], degree 3 is equivalent to serializability). This means that dirty reads, lost updates and unrepeatable reads cannot occur. Additional phantom protection would require extension of the lock granule. Since phantom protection is hardly required in the design environment, C^3 does not provide phantom protection by default, but the application may switch to full serializability.

5.2 DC-Level: Managing the DA-internal Designflow

At the DC level, we consider the internal structure of a single DA. Here only those versions are visible, which are relevant for data flow aspects. The set of operators is equivalent to the set of available design tools. The designflow (description) to be observed is application-specific and, therefore, specified by the user who created the DA. Fig. 10 shows an excerpt of a sample design-flow specification arranging the design steps of a particular DA (which is responsible for designing the body of a new car).

Note that the user does not have to specify workflows in the language of the example. The CONCORD system provides a graphical specification tool guiding the user in creating design-flow specifications. The graphical specifications are internally mapped to the specification language. Nevertheless, we see in the example that the basic concept of design-flow specifications is the transition. A transition transfers design states into each other. A design state can be imagined as the set of currently accessible design versions. A transition specification contains a pre- and a post-condition as well as an activity description. The conditions have the usual semantics. The activity description is more interesting. Here, we provide the possibilities of specifying design-tool invocations, sub-design-flow specifications or abstract activity descriptions. Nesting of design-flow specifications can also be used to control cooperative phases as we will see in the next subsection. An abstract activity description means just leaving it to the designer on how to fulfil this design step, i. e. letting him dynamically decide on which tools to execute in order to fulfil the post-condition. Besides all that, well-known concepts for control flow (e. g. the sequence used in the example of Fig. 10, see CONTROL-FLOW clause) and data flow are supported, which we do not discuss in this paper.




```
DESIGNFLOW_DESCRIPTION Body_Design
    INPUT_PARAMETERS (in: inf_data)
    OUTPUT_PARAMETERS (out: body_data)
    OUTPUT_BINDINGS (out:=S3.body)
    TRANSITIONS (
      (T1     (TRUE)
              (Determine_Frame_Data
              IN (in := START.in) OUT (Frame := out))
              (Frame.state = verified))
      (T2     ...)
      ...
    CONTROL_FLOW (S1->(T1, S2)->(T2, S3)-> ... );
```

Fig 10: Excerpt of a Sample Designflow Specification

5.3 AC-Level: Controlling Cooperation

At the highest activity layer, we consider the AC-System manipulating versions which are subjected to cooperation. This includes preliminary data which is exchanged between cooperatively working DAs (usage) as well as final data w.r.t. a certain DA which has fulfilled its task and passes the result back to the superordinated DA (delegation). Thus, the activity units are the (sub-)DAs incorporated into a design process and the operators are the corresponding design-flow types or the incorporated tool invocations, respectively. The corresponding design flows may be interleaved or may access cooperation data according to application-specific correctness criteria. Constraints may be divided into cooperation-processing constraints and cooperation-data constraints, together restricting the set of allowed histories of tool applications.

Let us first consider the different types of cooperation supported in CONCORD. Although, cooperation specifications logically belong to the AC-level, they are embedded into design-flow specifications. Due to simplicity of specifications we used the same language. This language supports the specification of explicit as well as implicit cooperation. *Explicit cooperation* relies on operations which can be compared with operations on access rights as

```
EXPLICIT_COOPERATION_DESCRIPTION Phase_1
 INVOLVED ARE (DA1, DA2)
 OUTPUT_PARAMS (out: transmission_data)
 OUTPUT_BINDINGS (out:=S3.transmission)
 TRANSITIONS (
 (T1   (TRUE) DA1: PERMIT (DA1, DA2, K2.engine, R) (TRUE))
 (T2   (TRUE)
       DA2:         (Transmission_Design,
       INPUTS (     in1:=DA2.START.in,
                    in2:=DA1.K2.engine)
       OUTPUTS ( transmission:=out))
       (transmission.state=verified))
 (T3   (TRUE) DA1: REVOKE (DA1, DA2, K2.engine, r)) (TRUE)))
 CONTROL_FLOW (S1->(T1, S2)->(T2, S3)->(T3, S4));
```

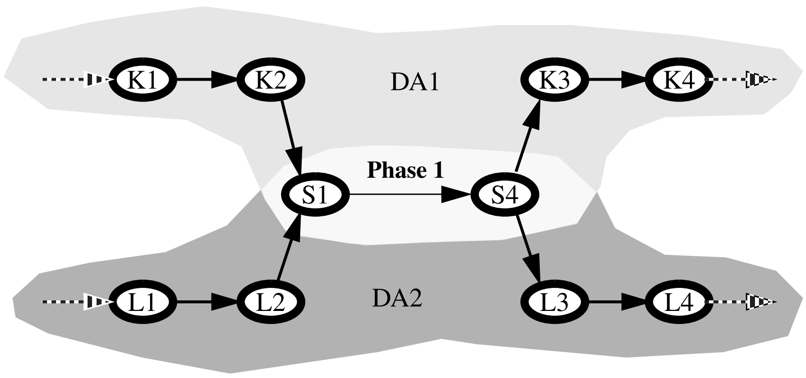


Fig 11: Sample Specification for Explicit, Pre-planned Cooperation

known from SQL2 [21]. Note that all data which is created or derived by design-tool applications is, first of all, inserted into the local scope of the corresponding DA and can only be made accessible for other DAs due to cooperation. Now, the operations *permit* or *transmit* can be used to cooperatively exchange preliminary design data. Permit grants a certain access right (read, derive or update) on a certain design-object state to a cooperating DA. Rights given by the permit operation can be withdrawn again by issuing the *revoke* operation. The operation *transmit* transfers ownership from one DA to another one; this can only be undone by issuing the transmit operation again (into the opposite direction). The usage of these operations can be allowed between certain DAs in certain design phases or can explicitly be preplanned. Fig. 11 gives, at its upper part, a sample specification of a pre-planned, explicit cooperation phase; the illustration at the lower part shows how this specification can be embedded into the design-flow specifications of the involved DAs.

The example shows how cooperation between DAs can be pre-planned as a sub-design-flow specification appearing in each of the design-flow specifications of the two DAs involved. After having reached design state S1 (which is associated with state K2), DA1 permits access on the designed engine to DA2. Then DA2 uses this information to design the transmission. Afterwards the access rights are revoked. Besides this pre-planned invocation of cooperation operations, there is the possibility of allowing the dynamic usage between certain DAs in certain design phases. Suppose, the two DAs of the example both have a sub-design-flow with abstract (open) activities. Within the specifications of these sub-design-flows it can be specified that the corresponding designers are allowed to issue cooperation operations dynamically.

```
IMPLICIT_COOPERATION_DESCRIPTION Phase_2
        INVOLVED ARE (DA1, DA2, DA3)
        INPUT_PARAMS (in1: engine_frame_data,
                in2: transmission_frame_data, in3: body_frame_data)
        OUTPUT_PARAMS (out1: engnine_frame_data,
                out2: transmission_frame_data, out3: body_frame_data)
        AUTOMATA (A1);

FINITE_STATE_MACHINE A1
        STATES (S1, S2)
        TRANSITIONS (
                (S1, S2, ANY, Verify_Frame_Data, ACCEPT)
                (S2, S1, ANY, NOT Verify_Frame_Data, ACCEPT))
        START_STATE S1
        END_STATES (S2);
```

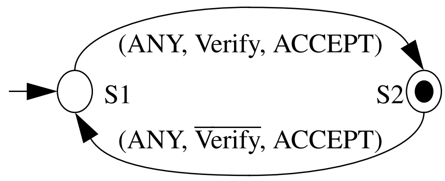


Fig 12: Sample Specification for Implicit Cooperation

The basic idea of *implicit cooperation* is that there are no local data scopes of the cooperating DAs, but there are object pools the DAs are concurrently working on by applying design tools. Correctness of processing can only be enforced by specifying, which sequences of design-tool applications are correct and which are not. For that purpose, we exploit the mechanism of finite, deterministic automata, known from the 'Cooperative Transaction Hierarchies Approach' [22]. A sample specification is illustrated in Fig. 12. We see the specification of an implicit cooperation phase, which can be imagined as a sub-design-flow occurring in the design flows of the involved DAs. The object pool, being the set of data the cooperative phase is working on is initialized in the INPUT_PARAMS clause and the correctness criteria is given by the automaton A1. Due to simplicity we use just one automaton in the example. The mechanism still holds for a set of automata. The basic idea is the following. Suppose we are in the initial state S1. Now, the involved DAs may issue design-tool applications on the data in the object-pool. The final state of the automaton (S2) can only be reached by verifying the frame_data (upper transition). The design phase can only be left when the automaton is in the final state. But if, after having verified the frame_data, a DA again modifies the frame_data, and, thereby, makes S1 the current state (lower transition), again the verify tool must be invoked, before the (cooperative) design phase can be finished. Thus, the automaton enforces the last step of the phase to be the verification step. In a similar way, more sophisticated patterns of cooperation can be specified.

This finishes our outline of the possibilities in specifying user-defined correctness criteria related to special usage relationships. Delegation and Negotiation relationships follow predefined protocols which do not have to be adapted to application-specific needs. Since these protocols are intuitive, we do not detail them.

6. Providing Cooperation Control Services in SERUM

After having given a detailed description of the SERUM process for providing tailored repository manager functionality for data management, especially version management, in Sect. 4, as well as an outline on how cooperative access to versions is controlled at the different CONCORD activity layers we now want to deal with the problem of generatively provisioning repository manager functions for activity management, which is the focus of our current work. Consequently, and, additionally, due to space restrictions, we cannot give a description, which is as detailed as the one given in Sect. 4. Thus, this section is more intended to describe our position and goals of ongoing work than to present final solutions. In order to outline the basic ideas on how to support activity management by means of SERUM concepts, we decided to elaborate on cooperation control mechanisms residing at the topmost activity layer of CONCORD (see Fig. 1 and Sect. 5.3), the administration/cooperation level.

First we want to recall the basic cooperation control services a repository has to provide in order to properly serve as a CONCORD repository (cf. Sect. 2.4). Regarding the basic concepts CONCORD offers at the administration/cooperation level, we can identify three major areas of activity control. First, users (roles, groups) are to be managed and these users must be associated with DAs, which are the central operational units at this level representing design tasks. Second, private workspaces of design data are to be associated with DAs and, additionally, private data can selectively be made visible (accessible) for cooperating DAs. Visibility can be controlled dynamically by users or can be pre-planned by enhancing design flow descriptions by operations adjusting visibility of meaningful design object states. Third, data manipulation actions performed within the scope (sphere of control) of a DA, e. g. design tools, must implicitly be caused to only access data, which is currently visible to that DA (w. r. t. the corresponding private workspace and current cooperation relationships).

In order to provide activity management components which may be customized towards fulfilling the mentioned tasks, SERUM offers three basic concepts (for cooperation control):

- *Dependency Management.* In cooperative design environments, a lot of dependencies between different tasks exist w. r. t. control flow and data flow. Hence, dependency management is a helpful support for the CONCORD design manager.
- *Access Control.* Access control is not only related to security issues. If multiple developers work on the same shared database, access control is crucial in order to avoid inconsist-

encies. Therefore, flexible repository mechanisms are necessary which allow for customized access control facilities.

- *Filter.* Besides controlling object access, it is necessary to limit the scope of operations. In CONCORD design operations of a DA are limited to the data owned by this DA or having been made visible by cooperating DAs. In order to provide such functionality, the SERUM approach includes the concept of filters. Filters control the visibility of objects to subjects in a given context and, hence, restrict the scope of operations.

In SERUM each of these aspects is related to a base component. In the following, we will take a closer look at these components.

6.1 Dependency Manager

Petri nets have been proven to be a useful concept in the field of workflow management and designflow management [9]. Their mature formal foundation allows for tool supported analysis and simulation. In the SERUM dependency manager (DM), we exploit the features of petri nets for managing dependencies between different tasks. It includes a petri net machine (PNM) which provides functionality for the specification and animation of petri nets. Our PNM is based on a variant of predicate/transition nets (P/T nets). A special feature of our implementation is that it has almost completely been integrated into the ORDBMS by using the extensibility features of object-relational database technology.

While we use petri nets for managing dependencies between different tasks, external actors have to take care of the tasks themselves. One of our design goals was to decouple these actors from the PNM. Therefore, we have introduced a special kind of places, called magic places. A signal is raised, if a token is received by one of these places. The signal contains the token and the identifier of the magic place. External actors are free to catch the signal and to handle the job. Afterwards, they send a notification to the magic place. This notification changes the internal state of the token making it visible for depending transitions. Hence, there is no direct linkage between magic places and actors. Both are decoupled, which allows for a broader field of application.

In P/T nets a firing condition is associated with each transition. If the firing condition is not fulfilled, the transition must not fire. The integration of the PNM into the database server allows for linking transitions to the state of product data stored in the same database. To do so, we take advantage of the activity features of the ORDBMS, i. e., database triggers. If the state of the product data related to a transition changes, then an associated database trigger asks the transition to check its firing condition. This allows the relation of dependencies to product data without a tight integration into the product data management facilities.

6.2 Access Manager

In cooperative design environments, access control is crucial. Hence, SERUM provides a component for integrating customized access control into the repository manager. This component is called the access manager (AM). If a subject wants to apply an operation to an object, it first has to send a request to the AM. The AM checks whether or not this subject is allowed to apply the operation. Typical responses are to deny access to the object or to delay access until the subject is allowed to apply the operation. Additionally, the subject is allowed to decide itself how long it wants to wait, if access can not be granted immediately. The response of the AM is represented by a so-called response object which connects subjects, operations, and objects (Fig. 13). If the AM receives a request, then it reroutes it to the response object responsible for the particular triple of subject, object, and operation.

A flat view to subjects, objects, and operations is not suitable for most applications. Hence, the AM provides several customization points for more flexible modelling of the relationships between the objects in each group. For example, the 'supplier' relationship enables the repository designer to customize the AM towards implementing a hierarchical group concept for subjects, where each group is a subject itself. In our example (see Fig. 13) subject S_1 is a supplier of subject S_2, which means that all properties of S_1 are supplied to S_2 if necessary. A higher level tool may consider S_1 as being a group, but the AM does not depend on this interpretation.

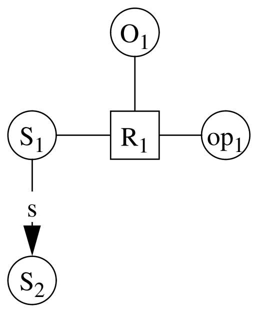


Fig 13: Access Control Example

Let subject S_1 ask for access to object O_1 with operation op_1. If the AM receives the request, then it looks for an appropriate object R_1, which connects S_1, O_1 and op_2, and forwards the request to this object. For a similar request by S_2 the AM will not be able to find a response object connecting S_2, O_1 and op_1. Hence, it asks all suppliers to provide a suitable response object. In our example, S_1 will deliver R_1. Similar concepts help to customize the facilities for managing objects and operations. Objects and operations may also be structured with relationships and the search for an appropriate response object has to take into account these relationships, too. By choosing the relationships between subjects, the repository designer can adapt the AM to his needs.

Additionally, the response objects are customization points, because these objects respond to the actual request. Hence, by extending the set of response objects available in the framework, new kinds of responses can be integrated into the AM, if necessary.

In our opinion, the generic functionality of the AM is suitable to support a broad range of applications in the field of access control in our repository manager family.

6.3 Filter Manager

In cooperative design environments, it is necessary to limit the scope of operations to product data, which is related to the work of a subject in a given context. We use the concept of filters to hide objects outside the scope of an operation. The concept of filters is embodied by the filter manager (FM) which manages the visibility of objects.

Let us examine, for example, a database query. Based on the features of an ORDBMS, queries can access the FM through a simple UDF which evaluates to ‘true’ if the object is visible. This UDF can be used in database queries to limit the scope of the query to objects, which are currently visible. Considering a CONCORD design activity (DA) as a scope relevant for the FM, it would (logically) work as follows. Designers, or design tools, respectively, may issue queries without taking their (data) scope into account explicitly. The FM knows about the scope corresponding to the query issued and after the query has been evaluated against the entire database it cuts down the query’s result set by removing all objects (versions), which are not visible for the issuing DA w. r. t. cooperation control.

Both, the FM and the AM are based upon similar subject and object management facilities, but the FM uses a context management infrastructure instead of the operation management facilities used by the AM. Here, again, the external interpretation of the relationships is not crucial to the FM.

6.4 Architectural Overview

In Fig. 14 an architectural overview of the basic services provided by SERUM in order to support customized cooperation control is given. Note that this illustration only relates those components discussed previously in this section with each other, and, therefore, is not necessarily complete. For example configuration management and context management services are not taken into account.

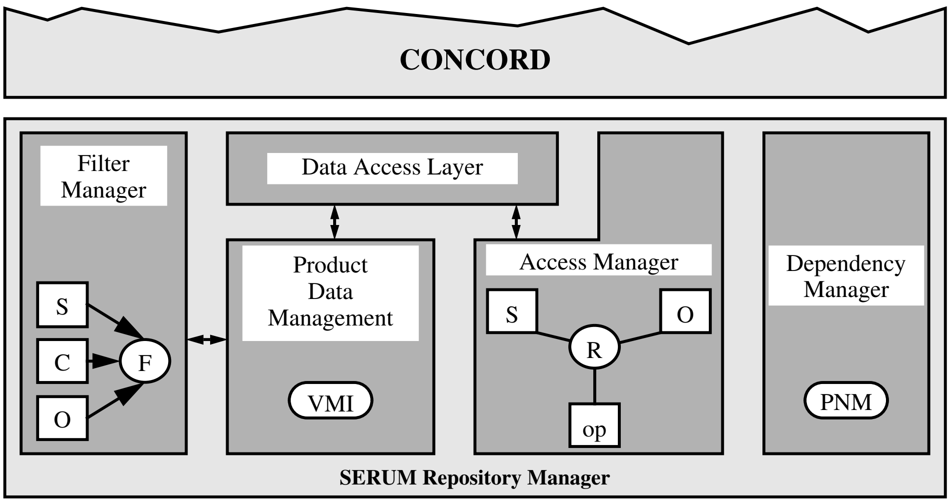


Fig 14: Architectural Overview

Every access to objects managed by the product data management (PDM) has to be passed to the data access layer. This component asks the AM for the right to apply a certain operation to these objects. This decision is taken by the AM by taking internal information into account which has been specified by high level tools through the public interface. The semantics of the internal structures as well as the interfaces provided to external tools are customized during the SERUM development process. This enables CONCORD to exploit the AM facilities to ensure synchronized access and wanted security policies.

The scope of operations provided by the PDM depends on the visibility of objects. Hence, the PDM has to ask the FM, whether or not the object to be manipulated is visible. An object's visibility depends on the subject and the corresponding context. External tools, as for example the CONCORD cooperation manager, can change the internal information of the FM through its public interface, which is generated by the SERUM application generator. Additionally, a generic interface allows other repository components to access the FM. For example, a response object managed by the AM can change the visibility of a data object, after access has been granted to a subject, in order to avoid this object being visible to other subjects any longer.

The DM is almost completely independent of other components and is used as a black box. CONCORD may use it to control the execution of design flows. The cooperation manager acts as an external component and reacts to the signals raised by the DM. If necessary, the DM can be coupled to the product data at the ORDMBS level by exploiting database triggers to monitor manipulations of product data.

Although not taken into account in Fig. 14, it has been mentioned several times that ORDBMS technology, again, is the foundation for implementing the activity management services discussed in this section. The DM has (almost) completely been realized as a DBMS extension consisting of UDTs modelling PNMs, and UDFs allowing the creation and deletion of PNM instances and firing corresponding transitions. These UDFs can be used within SQL statements serving for communication between actors and the DBMS. Furthermore, database triggers may be used to automatically call the function responsible for firing transitions. FM as well as AM functionality is needed to be encapsulated at each layer (client, repository server, ORDBS) of a repository manager (cf. Fig. 4) in order to control requests. At the lowest layer which is the ORDBS layer, again, corresponding functionality can be realized by means of UDTs and UDFs.

7. Related Work

In this paper, we brought together the worlds of cooperation models, versioning and application generators. We have chosen the CONCORD model as a representative cooperation model [9] which relies on a repository supporting versioning [10] of design artifacts. Versioning has been emphasized because of its crucial role w. r. t. data management for design applications and its manifoldness w. r. t. available versioning models. The latter underlines the need to tailor versioning facilities in an application-specific manner. This, in turn, is what we are aiming at in our SERUM approach.

What SERUM has in common with other approaches is the basic idea of reuse [23, 24]. Considering the classification of reuse techniques given in [23], SERUM exploits the principles of *software components, software schemas, application generators, and software archi-*

tectures. As many application builders, e. g., user interface builders or simulator generators (e. g., MOOSE [25]), the SERUM repository generator is highly domain-specific. However, in contrast to other approaches, our domain is given as data management facilities (repository managers) for software development applications. Furthermore, we aim at generating software components fitting into a generic software architecture, which, as we think, serves software development applications best. This architecture (as discussed in Section 3) consists of API functions, application server modules and an object-relational DB schema. The latter also comprises object-relational extensions, e. g., user-defined functions effectively supporting the application servers on top. A similar idea w. r. t. the software architecture is pursued in [26], but, to the best of our knowledge, our approach is the first one generating DB applications containing object-relational extensions. As we think, the extensibility property of object-relational DBMS [27] provides a well suited infrastructure for generatively creating DB applications.

SERUM is not a single-step but a transformational approach. In contrast to other transformational approaches, e. g. GenVoca [28], refinements are not applied to executable components but to the (UML) specification which serves as input for the SERUM repository generator. The transformations performed by design pattern applications represent the actual process of customization in SERUM. Other approaches do not support automatic adoptions of input specifications for generators. The transformation performed in these other approaches are more aiming at fulfilling non-functional requirements, which in SERUM are taken into account by the concept of design templates.

Concluding the comparison to related work, we can say that SERUM mainly differs from other approaches in its domain (generating data management facilities for software development applications), the resulting software architecture (applications servers on top of object-relational DBMS) and the transformational approach represented by the SERUM design patterns.

8. Conclusions

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. First, 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. Second, pre-defined, technology-independent design patterns relieve repository designers by automatically enhancing UML models by application-independent aspects. Third, 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. Additionally, we have outlined that similar (generative) customization processes may be applied in SERUM w. r. t. activity management, especially cooperation control. Thus, we have shown an efficient way of generatively providing repository manager functionality, which before had to be implemented completely manually. Furthermore, we have argued that UML and the extensibility property of object-relational database technology is very helpful in this concern.

The discussion has shown that the data management facilities as well as the activity management 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 effectively puts into practice the basic idea of reusing design artifacts. 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 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 and define the interfaces of the corresponding system components. Furthermore, we want 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, and to evaluate/validate the overall SERUM approach in real application scenarios.

References

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

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

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

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

[5] Bernstein, P.A., Bergstraesser, T., Carlson, J., Pal, S., Sanders, P., Shutt, D.: Microsoft Repository Version 2 and the Open Information Model, Information Systems 24:2, 1999, pp.71-98.

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

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

[8] 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.

[9] 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.

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

[11] Käfer, W., Schöning, H.: Mapping a Version Model to a Complex Object Data Model, Proc. 8th Int. Conf. on Data Engineering, Tempe, Arizona, 1992, pp. 348-357.

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

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

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

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

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

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

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

[19] 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.

[20] 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, pp. 251-270.

[21] Date, C., Darwen, H.: A Guide to the SQL Standard - A User's Guide to the Standard Relational Language SQL (3rd Edition), Addison-Wesley Publishing Company, 1993.

[22] Nodine, M.H., Zdonik, B.: Cooperative Transaction Hierarchies: Transaction Support for Design Applications, VLDB Journal 1, 1992, pp. 41-80.

[23] Krueger, C.W.: Software Reuse, ACM Computing Surveys 24:2, 1992, pp. 131-385.

[24] Mili, A., Mili, R., Mittermeir, R.T.: A Survey of Software Reuse Libraries, *Annals of Software Engineering* 5 (1998), pp. 349-141.

[25] Altmeyer, J., Riegel, J.P., Schürmann, B., Schütze, M., Zimmermann, G.: Application of a Generator-Based Software Development Method Supporting Model Reuse, Proc. 9th Conference on Advanced Information Systems Engineering (CAiSE*97), Barcelona (ES), June 1997, pp. 159-172.

[26] What is DSSA? Online Document, http://www.lfs-owego.com/dssa/what-is-dssa.html, Aug. 1995.

[27] Stonebraker, M., Brown, P., Moore, D.: Object-Relational DBMSs, Second Edition, Morgan Kaufmann Publ., San Mateo, CA, 1998.

[28] Batory, D., Geraci, B.J.: Composition Validation and Subjectivity in GenVoca Generators, IEEE Trans. on Software Engineering 23:2 (special issue on Software Reuse), 1997, pp. 67-82.