Specifying Customized Versioning Facilities of Software Engineering Repositories by Using UML-based Design Templates

Jernej Kovse
Department of Computer Science
University of Kaiserslautern
P.O. Box 3049, D-67653 Kaiserslautern
e-mail: kovse@informatik.uni-kl.de

Abstract

Software engineering repositories are often deployed in different environments, which may pose environment-specific demands related to repository services. Since most repository products do not support any significant modification of their services, the possibilities of customizing the repository to the requirements of an environment are limited. To overcome this problem, our approach assists the software engineers in delivering a UML-based specification of the repository product with its services customized to the requirements of the environment. On the basis of this specification, a customized repository to be used in the environment is generated. This paper primarily deals with the specification phase of the process for the domain of customizing the repository's version and configuration control services.

Keywords: Software Engineering Repositories, Versioning, UML Model Enhancement

1 Introduction

Bernstein and Dayal [1] define a *repository* as a shared database of information about engineering artifacts. A computer-based cooperative engineering environment may use a repository to support the storage and sharing of artifact information manipulated using the tools integrated by the environment. A *repository manager* [1] is a database application that provides services for modeling, retrieving, and managing objects in a repository. It incorporates standard amenities of a database management system (DBMS) (a data model, queries, views, integrity control, access control and transactions) as well as additional *value-added services*: checkout/checkin, version control, configuration control, notification, context management, and workflow control [1].

Software engineering (SE) *repositories* are used to store and share information related to software artifacts. SE environments where such repositories are deployed are to a large extent *dissimilar* due to diverse methodologies and tools used in the environments. Ideally, it should be possible to customize the services of a repository to these requirements to improve its performance in the environment. However, the form of the services is usually predefined by the repository vendor, which makes their customization to the environment impossible in any significant way. Instead, software engineers are forced to adapt their application of methodologies and tools present in the environment in order to be able to use the form of repository services as delivered by the vendor. This drawback may lead to a decrease in the productivity in the environment and thereby to a degradation of the SE process.

Our SERUM (Software Engineering Repositories using UML) project [2] is part of a long-term research effort[1] dealing with the development of large systems using generic methods. In this project, we investigate the process of *generating* SE repositories with customized services. The process consists of two consecutive phases:

1. *Specification phase*: We assist the software engineers in delivering a *UML-based specification* describing the semantics of the customized services the engineers want to use in their environment.
2. *Generation phase*: On the basis of the delivered specification, a repository product with customized services is *generated*.

1. Sonderforschungsbereich (SFB) 501, funded by the Deutsche Forschungsgemeinschaft (DFG).

This paper focuses primarily on the specification phase of the process for the domain of customizing version and configuration control services of SE repositories. Our research question consists of two interrelated parts. First, we want to identify the customizable aspects of version and configuration control services (Section 2). Second, we are interested in a mechanism that (a) assists the software engineers in customizing the services and (b) is capable of enhancing the UML-based specification of the repository product so that the descriptions of the customized services become a part of this specification. In Section 3, *SERUM design templates* are introduced as an implementation of such a mechanism. Section 4 briefly discusses the generation phase of the SERUM approach. In Section 5, we make a conclusion where we summarize the advantages of our solution and present some ideas for the future work.

2 Customizing Versioning Facilities of SE Repositories

Katz [3] defines a *version* as a semantically meaningful snapshot of a design object at a point in time. *Configuration* is defined as a binding between a version of a composite object and a version of each of its components. This section describes the customizable aspects of version and configuration control services of SE repositories.

2.1 Simultaneous Versioning of Related Objects

In the course of describing a complex software product, multiple versions of a repository object may be created in order to reflect the modifications committed to the object. Objects may be versioned individually meaning that each created version refers to the modifications of a single object. However, this approach does not always prove convenient, since software engineers may recognize a *group* of objects as semantically interrelated for the context of versioning. In such cases, we would like to enable the creation of a version that refers to the modifications of multiple related objects. For example, suppose the engineers want to describe a software product by storing objects of types *application*, *class*, *attribute*, *method* and *parameter* in the repository. A class may contain multiple attributes and methods (which may receive multiple parameters). An application consists of one or more classes.

Suppose the engineers would like to version a class by simultaneously versioning a class object with its related parameter, method and attribute objects. Later, versions of classes may be attached to versions of applications to form application configurations. Fig. 1 illustrates a sample application configuration. For example, the version *Class c1, version 1* is formed by simultaneously versioning the class object *c1*, the method object *m1* and the parameter objects *p1* and *p2* associated with *m1*. Offering the software engineers the possibility to define which object types are to be versioned in groups and how the created versions may be combined to form configurations is one of the key aspects of customizing versioning facilities of SE repositories.

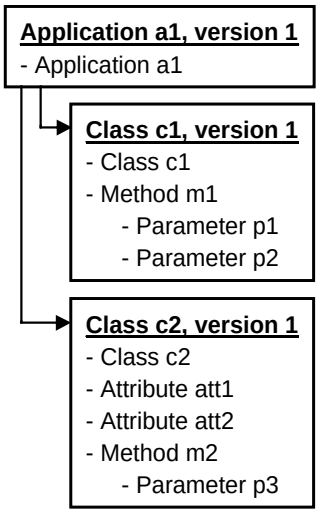

Fig. 1: Sample application configuration.

2.2 Defining Constraints on Versioning Operations

There is a basic common set of operations defined by the versioning facilities. The *create* successor operation creates a successor to a given version in the version history. The *merge* operation unites two or more versions. The operations *get successors*, *get alternatives* and *get ancestor* enable the traversal of the version history. The *freeze* operation prevents further modifications to a version. The *attach* and *detach* operations are used to form configurations by attaching and detaching component versions.

Customization of repository's versioning facilities may be achieved through specifying *constraints* related to the execution of these operations. Software engineers define the constraints in form of *pre-* and *postconditions* for an operation and *invariants*. As an

example for the usage of preconditions, suppose only frozen versions of component objects are allowed to be attached to the versions of composite objects. This requirement can be expressed as a precondition to the *attach* operation.

Invariants defined for a version have to be satisfied at any time in the version's lifecycle. For example, an invariant may be used to express a constraint that versions of a given object may have at most three successors.

2.3 Defining Possible Version States

Repository's versioning facilities usually recognize two possible version states: *unfrozen* and *frozen*. An important aspect of customizing versioning facilities is allowing the software engineers to define additional version states. For example, suppose the engineers require each created version to undergo consecutive testing phases of unit-testing and system testing after being frozen. To support this kind of versioning functionality, the engineers define the possible substates *untested*, *unit-tested* and *system-tested* of the frozen state and the allowed transitions between these states.

3 Specifying a Repository with Customized Services

This section discusses the specification phase of the SERUM approach. Initially, the UML-based specification consists of a special *product data model* expressed as a UML class diagram. In the product data model, the engineers define types of objects that will be stored and managed by the customized repository. Fig. 2 illustrates a sample product data model. In the remainder of the specification phase, the specification is gradually enhanced, so that in its final form it incorporates the engineers' requirements related to the repository product. SERUM supports this specification enhancement process by a set of *design templates*. A design template is a *generic modeling component* that refers to a customizable part of the repository product, for example, to its versioning facilities. The engineers choose the appropriate design template from the set of available templates according to the repository part they want to customize. Once a design template is chosen, it is applied in two steps, *design template configuration* and *specification enhancement*.

3.1 Design Template Configuration

A chosen design template initially comes in a generic state. Prior to its application, the template is configured by the engineers by means of setting the template's configuration parameters. A configured template involves the customized semantics of the repository services and is capable of integrating this semantics in the UML-based specification. The *Version* design template supports the customization of versioning facilities. By setting the values of its parameters, software engineers customize the version

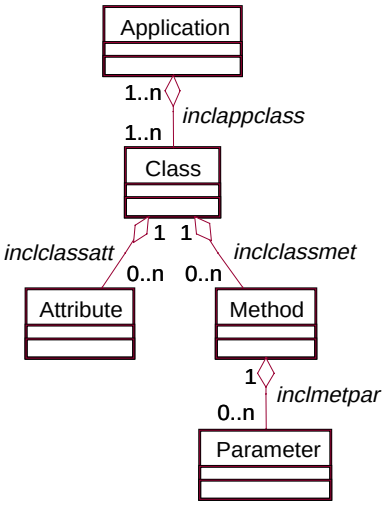

Fig. 2: A sample product data model.

```
Version Design Template, Name = V_Class
    ...
    Domain = {Class, Attribute, Method, Parameter};
    MaxSuccessors = 3;
    FrozenSubstates = {Untested, Unit-tested, System-tested};
    ...
```

Fig. 3a: Design template configuration for class versioning.

```
Version Design Template, Name = V_Application;
    ...
    Domain = {Application};
    VersionAttachments = {[ Name = V_Class,
                            Precondition = frozen]};
    ...
```

Fig. 3b: Design template configuration for application versioning.

and configuration control services. Suppose that in the repository storing the objects of types illustrated in the product data model in Fig. 2, we want to customize the versioning facilities as follows:

- Class objects with related attribute, method and parameter objects are to be versioned simultaneously (*Requirement 1*). Each class version may not have more than three successor versions (*Requirement 2*). After being frozen, a version of a class may find itself in one of the following testing phases: *untested*, *unit-tested* and *system-tested* (*Requirement 3*). For this purpose, the engineers configure an instance of the Version design template with the parameter values as illustrated in Fig. 3a.
- Application versions version application objects only (*Requirement 4*). Only frozen class versions may be attached to versions of application objects to form configurations (*Requirement 5*). For this purpose, the engineers configure another instance of the Version design template with the parameter values as illustrated in Fig. 3b.

3.2 Specification Enhancement

In this step, a configured template is used to enhance the UML-based specification of the repository product. Thereby, the customized semantics expressed through the parameter values of the configured templates becomes a part of the specification. As an example of specification enhancement, we use the two instances of design templates configured as illustrated in Fig. 3a and 3b to enhance the initial specification, consisting of the product data model illustrated in Fig. 2. As a result of both enhancements, the resulting specification contains:

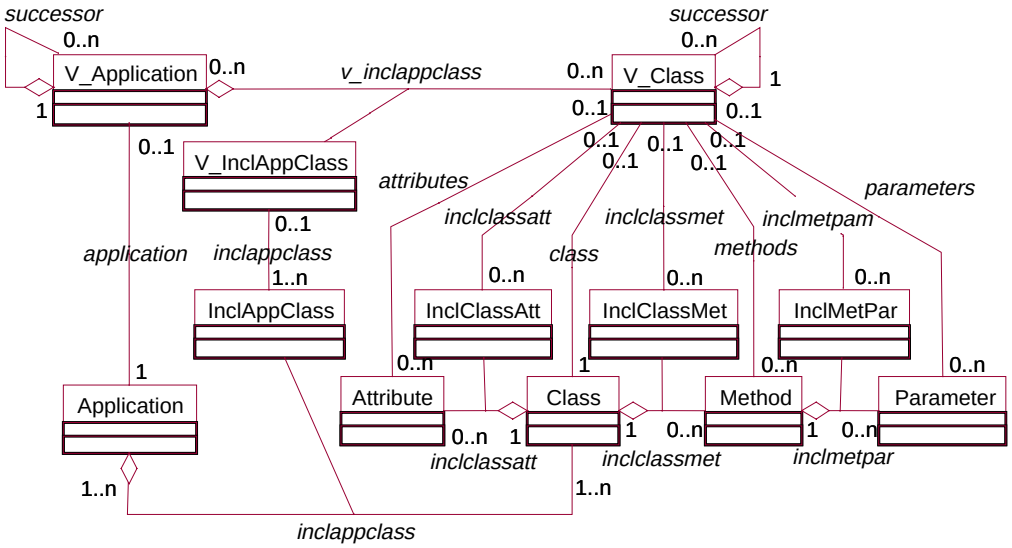

Fig. 4: Enhanced product data model.

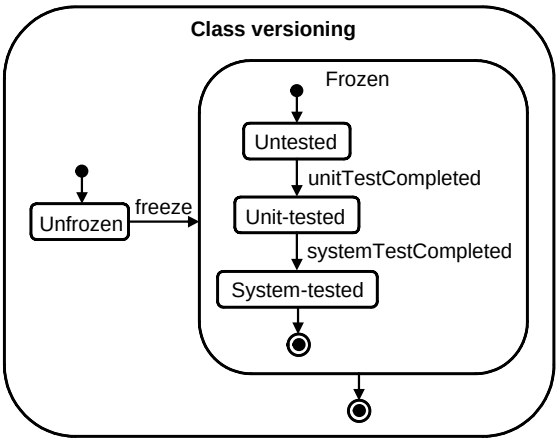

Fig. 5: A statechart diagram for class versioning.

```
context V_Class
    inv: self.successor->size < 4
    ...
```

Fig. 6: Restricting the number of successors to a class version.

```
context V_Application::attach(c : V_Class)
    pre: c.oclInState(Frozen)
    ...
```

Fig. 7: Restricting the attach operation of application versions.

- an enhanced product data model (Fig. 4) enriched with the classes *V_Class* and *V_Application* that support the required creation of versions and configurations (meeting the *Requirements 1 and 4*);
- a statechart diagram (Fig. 5) defining possible states of class versions that correspond to different testing phases (meeting the *Requirement 3*);
- two Object Constraint Language (OCL [4]) constraints (Fig. 6 and 7) related to the version and configuration control operations (meeting the *Requirements 2 and 5*).

4 Generating the Repository

This section briefly discusses the generation phase of the SERUM approach. In this phase, the repository product is generated on the basis of the UML-based specification delivered in the preceding specification phase. As a result of the generation phase, the following components are produced:

- A repository database schema contains definitions of database tables used to store repository objects. It is generated on the basis of the definition of repository object types and associations between them as present in the enhanced specification.
- A repository manager exposes customized repository services to repository clients. The basic Create-Read-Update-Delete (CRUD) operations enabling the manipulation of repository objects can be automatically generated. In the case of customized versioning facilities, the basic versioning operations described in Section 2.2 are generated and the constraints related to these operations, such as those illustrated in Fig. 6 and 7 are taken into account in this process. On the basis of the definitions of possible version states in the specification, the operations allowing the transitions between the states are generated. In case the software engineers require additional functionality to be included in the generated repository, they have to supply the corresponding code segments in the course of the generation phase.

5 Conclusion

SERUM design templates used in our approach and discussed in this paper are an implementation of a generic modeling mechanism enabling the semi-automatic enhancement of a UML-based repository specification. The discussion has illustrated that design templates may be used by the engineers to specify the customized form of repository services and integrate the resulting customization aspects in the repository specification by *specification enhancement*. In this way, SERUM design templates successfully support customization of SE repositories on the level of UML-based specification.

Repositories customized according to the requirements of a specific SE environment offer better performance in an environment and thereby play a significant role in the improvement of the SE process. For this reason, we intend to continue with our research on the proposed approach to pre-implementation customization of SE repositories. In our future work, we intend to:

- develop and evaluate a set of interactive tools enabling efficient application of design templates;
- further explore the possibilities of using the UML for the specification of customized repositories;
- compare the performance and usage of customized repositories generated using diverse implementation technologies, such as different DBMS used to store repository objects and different software component models used to expose repository services.

References

[1] Bernstein, P.A., Dayal, U.: An overview of Repository Technology, in: Proc. of the 20th Int. Conf. on Very Large Data Bases (VLDB'94), Bocca, J.B. et al. (Eds.), Santiago de Chile, Sept. 1994, Morgan Kaufmann, pp. 705-713.

[2] Härder, T., Mahnke, W., Ritter, N., Steiert, H.-P.: Generating Versioning Facilities for a Design Data Repository Supporting Cooperative Applications, in: Int. Journal of Intelligent & Cooperative Information Systems 9:1-2, 2000, pp. 117-146.

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

[4] OMG, Unified Modeling Language Specification, Version 1.3, OMG Document ad/00-03-01, March 2000.