VS-Gen: A Case Study of a Product Line for Versioning Systems

Jernej Kovse[1] and Christian Gebauer[2]

Department of Computer Science
Kaiserslautern University of Technology
P.O. Box 3049, D-67653 Kaiserslautern, Germany
[1]kovse@informatik.uni-kl.de, [2]gebauer@gmx.com

Abstract. This paper describes our experience with developing a product line for middleware-based versioning systems. We perform a detailed domain analysis and define a DSL for configuring individual systems. Afterwards, we present a template-based approach for generating versioning systems from UML models. The presented approach is evaluated from two perspectives. We first use diverse measures to determine the properties of code templates used by the generator. Afterwards, we compare the performance of a generated versioning system to a system that has been developed by means of a framework and thus has to rely on a set of generic implementation components.

1 Introduction

Versioning systems are generally used to manage data objects that change frequently over time. Their main function is to represent intermediate stages in the object's evolution path as *versions* and allow users to revert to these stages afterwards. Versioning is useful in a variety of different applications: In a software project, developers will want to version their specifications, data and process models, database schemas, program code, build scripts, binaries, server configurations, and test cases. In an editorial department of a magazine, authors will want to version the text of their articles, images, tables, teasers, and headlines before submitting the final version of their contribution.

This paper summarizes the practical experience gathered with our *VS-Gen (Versioning Systems Generator)* project. The goal of the project is to support a product line of versioning systems. A specific system from the product line is specified by describing its data model in UML and afterwards selecting the features desired for the system on the basis of this model. Feature selection is supported by a UML profile. The resulting UML model is analyzed by a template-driven generator that delivers a complete middleware-based implementation of the versioning system.

The rest of the paper is organized as follows. The features of versioning systems will be presented in Sect. 2 which provides a detailed domain analysis for the product line. Sect. 3 will first examine the implementation components of versioning systems

to illustrate what has to be generated automatically and then outline the generation approach. We evaluate our product line from two perspectives: First, in Sect. 4, we use diverse measures to examine both the properties of generated code for an exemplary system from the domain as well as the properties of code templates used by the generator. Afterwards, in Sect. 5, we examine the performance of a sample versioning system generated with VS-Gen in comparison to a system developed using a framework that relies on a number of generic implementation components. This framework can be considered as an alternative implementation of the product line. Finally, Sect. 6 gives an overview of related work while Sect. 7 summarizes the presented results and gives some ideas for the future work related to VS-Gen.

2 Domain Analysis

A versioning system stores objects and relationships between these objects. Each object and each relationship is an instance of some object or relationship type. Object and relationship types are defined by the versioning system's *information model* [3]. For example, the OMG's MOF Model can be used as an information model in case the users want to store and version MOF-based metamodels, or the UML Metamodel can be used as an information model for storing and versioning UML models. Fig. 1 illustrates a very simple information model from the domain of content management, which will be used as an example for VS-Gen throughout this paper.

Obviously, it would be possible to come up with a generic implementation of a versioning system capable of dealing with any information model without needing any change in the implementation. In such an extreme case, even attribute values could be stored in a single relational table as name-value pairs. Even though such generic solutions represent performance drawbacks, certain variants prove useful due to their simplicity (we compare the performance of one selected variant to a system generated by VS-Gen in Sect. 5).

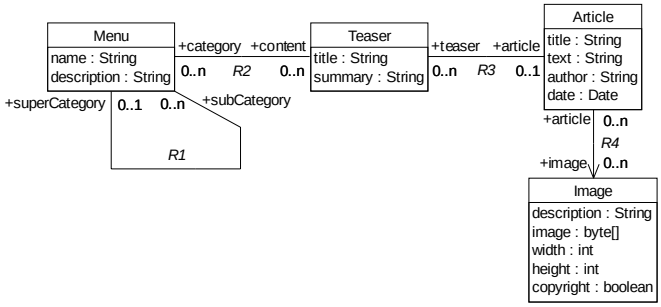

Fig. 1. A simple information model

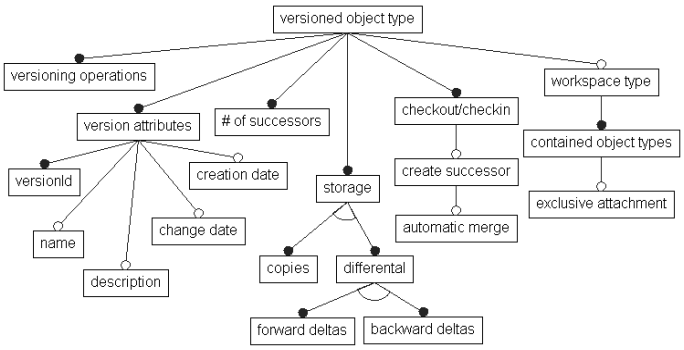

Fig. 2. Feature diagram for versioned object types

In VS-Gen, the information model represents a basis for configuring the desired versioning system. This is done by *selecting features* related to the model elements, as described in the following sections.

2.1 Object Types

An *object type* may or may not support versioning. For every object type, the versioning system provides only the basic operations *create*, *copy*, *delete* and the so-called *finders*. In case an object type supports versioning, each object (instance) of this type is also a version that belongs to some version graph. A corresponding feature diagram for versioned object types is illustrated in Fig. 2. A versioned type provides an identifying attribute *versionId* for referencing a version within the graph. Each graph is equipped with an *objectId*. A version can also be referenced directly using a *globalId* that comprises both identifiers. For a type that supports versioning, the user may require additional attributes for storing *version name*, *description*, last *change date* and *creation date*. Using the feature *# of successors*, it is possible to limit the number of direct successors for a version in the version graph. The storage of versions may proceed in copies that duplicate even the unmodified attribute values for fast access or using the so-called *forward* or *backward deltas* (storing only the differences from the predecessor to the successor version or vice versa). The *checkout/checkin* operations are used to lock/unlock the version for the exclusive use mode. It may be required that the system always creates a separate successor version upon checkout and leaves the original version unadorned (*create successor*). In this case, it may also be useful to *automatically merge* this version with the original upon checkin.

Each versioned object type provides the following operations (not depicted in Fig. 2) for managing versions and traversing the version graph: *createSuccessor*, *deleteVersion*, *merge*, *getAncestor*, *getSuccessors*, *getAlternatives*, and *getRoot*. Some of these carry further subfeatures, e.g., *prevent deletion of ancestors* is an optional subfeature of *deleteVersion* that we also use in the UML profile in Sect. 2.4. It marks whether a version can be deleted in case it already has some successors in the version graph.

2.2 Workspace Types

Workspace types (also see Fig. 2) are versioned object types with special properties. Workspaces (instances of workspace types) contain many objects (of diverse types). However, only one version of a particular object may be present in a workspace at a time – in this way, a workspace acts as a version-free view to the contents of a versioning system, allowing the user to navigate among the contained objects without explicitly referring to versions. A version arrives in a workspace using the *attach* operation. A workspace can possess an exclusive ownership for the attached versions, meaning that they cannot be attached to another workspace at the same time.

2.3 Relationship Types

The majority of interesting behavior in a versioning system is captured by properties of *relationship types*. Each type consists of two relationship ends that connect to object types. The features for an end are illustrated by the feature diagram in Fig. 3. An end is primarily characterized by the defined *role name*, whether it is *navigable*, and its *multiplicity*. More interestingly, in case an end connects to a versioned object type, the user can define the end as *floating*. A floating end contains a subset of versions that can be reached when navigating to this object. This subset is called a *candidate version collection (CVC)* and can be either *system*- or *user-managed*. In a system-managed CVC, the CVC is initialized by a version specified by the user; afterwards, the system adds every successor of this version (obtained either by invoking *createSuccessor* or by merging) to the CVC. In a user-managed CVC, the user explicitly adds versions from the version graph to the CVC.

There are two possible ways of using a CVC when navigating between objects. In *unfiltered navigation*, the user requests all versions from the CVC when navigating from some origin object (note that if the multiplicity of the end is *many*, there will be many CVCs, one for each connected object). In *filtered navigation*, the system automat-

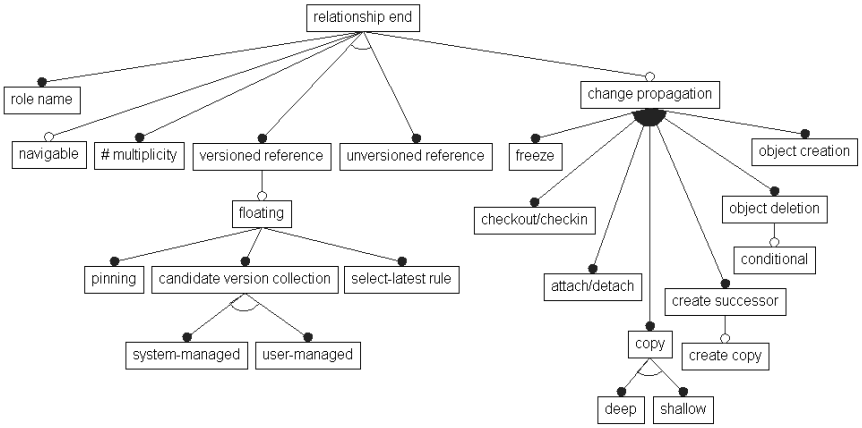

Fig. 3. Feature diagram for relationship ends

ically selects a version for the user. In case filtered navigation is carried out within a workspace, a version from the CVC that is attached to the workspace will be returned. Outside a workspace, the system first checks whether there is a *pinned version* in the CVC (this is the version the user has explicitly marked as default using the *pin* operation). Otherwise, the system selects the *latest* (*newest*) *version* from the CVC. Floating relationship ends prove useful for managing configurations of versions, but also represent a large performance overhead due to special properties. For this reason, we consider them optional, so the user can define the relationship end that connects to a versioned object type as non-floating and use it to connect to versions as if they were regular objects.

To illustrate the concepts of floating relationship ends, consider the following example based on Fig. 1. Suppose that the relationship end *content* which belongs to the relationship type *R2* and connects to the object type *Teaser* is floating. The multiplicity of the end is many. This means that a given version of a menu *m* connects to many CVCs for the teasers. Each of these teaser-CVCs is bound to a version graph for some teaser and contains a subset of versions from this version graph. Within each teaser-CVC, there is a latest version and there can be a pinned version. The following situations can arise where navigating from *m* across the relationship R2 towards *Teaser*.

- *Filtered navigation outside a workspace*. For every teaser-CVC related to *m*, a pinned version is returned, in case it exists. Otherwise, the latest version is returned.
- *Filtered navigation within a workspace w*. For every teaser-CVC related to *m*, we check whether there is a teaser version attached to *w* and return this version. Note that a CVC that connects to *m* does not have to contain a version attached to *w*. For such a CVC, no version is returned.
- *Unfiltered navigation*. All versions from all teaser-CVCs related to *m* are returned. It is up to the client application to decide which versions to use.

As illustrated in Fig. 3, relationship ends can also be used to propagate operations from the object the operation got invoked on towards its connected objects. When propagating *createSuccessor*, the user can choose to invoke *copy* on a connected object, thus initiating a new version graph. When propagating *delete*, the user can define that the connected object is to be deleted only if it does not connect to any other objects.

2.4 A Configuration DSL for Versioning Systems

In VS-Gen, a domain-specific language (DSL) is used for configuring a desired versioning system. The DSL takes form of a UML profile, illustrated in Fig. 4, and currently supports a subset of features presented in Sect. 2.1-2.3. In the configuration process, object types and relationship types are first described in a UML model. Afterwards, model elements are branded with stereotypes and tagged values are selected for tag definitions. An example of applying the profile to our sample information model from Fig. 1 is illustrated in Fig. 5. We also added a workspace type *EditorialDepartment* that can attach objects of type *Menu*, *Teaser*, and *Article*.

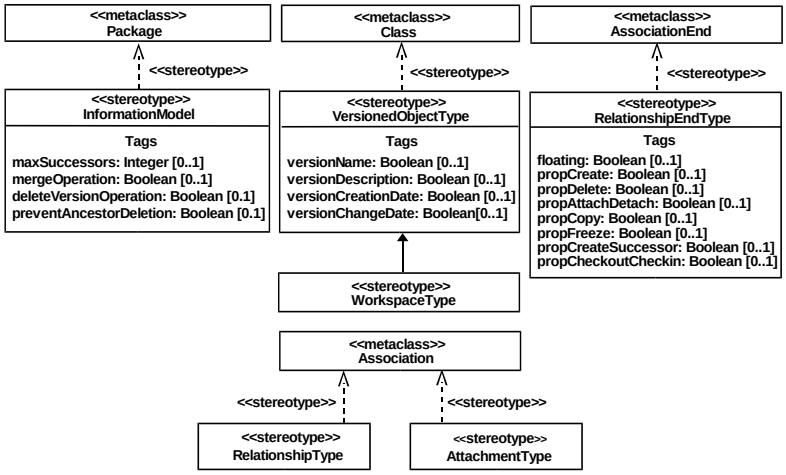

Fig. 4. A UML Profile for information models in VS-Gen

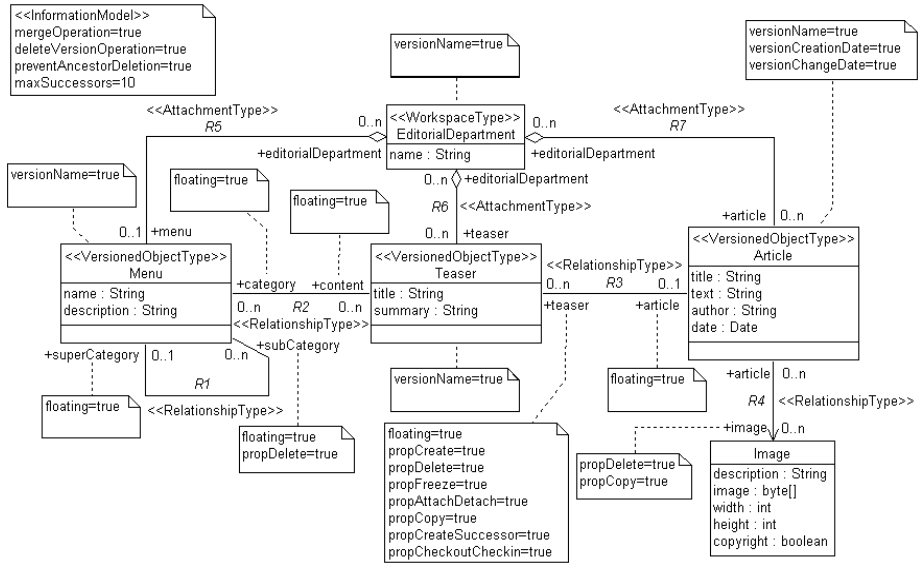

Fig. 5. A sample information model branded with stereotypes and tagged values

3 Generated Components

This section outlines the implementation components of a versioning system and the template-based approach used to generate these components. The target platform for generated systems is J2EE [20] with a RDBMS as a persistent storage for objects and relationships.

3.1 Persistence and Entity Layers

The *persistence layer* is composed of relational tables used for storing versioned objects. Every object type and workspace type gets mapped to a separate *object type table* using the *globalId* as the primary key. Versioning information (attributes *frozen*, *versionName*, *versionCreationDate*, *versionChangeDate*, and the one-to-many association for linking a predecessor version to its direct successors) gets integrated directly into the table. If possible, we avoid a separate *relationship type table* to minimize the number of joins required when navigating between objects. This can be done for non-floating relationship ends with multiplicity *one*. For floating relationship ends, references to the pinned version and the latest version are materialized as additional foreign key values to assure the fastest access possible. In this way, we avoid running through all target version keys in a CVC seeking for a flag that would mark the pinned version or looking for the latest version. In case the multiplicity of a floating relationship end is *one*, the two values are stored as columns in the object type table for the source of the navigation. Materializing the latest version setting represents an overhead for operations that change the state of the CVC, i.e., operations for creating or deleting the relationships. However, the materialization speeds up the operations for filtered navigation outside workspaces, which are invoked more frequently in typical usage scenarios.

The *entity layer* mirrors tuples from database tables to the so-called entity objects (implemented as CMP entity beans [19]) to present the access layer with a navigational access to data stored by the persistence layer. The navigational access is supported through the use of CMR [19]. The generated code for the entity layer relies on container services of the application server that assure persistence, entity caching, instance pooling, and context-based prefetch for navigational access.

3.2 Access Layer

The access layer is comprised of stateful session components (stateful session beans [19]) that carry out versioning operations by accessing and modifying the state of entity objects. Technically, the session components can be accessed by a native EJB client or, since we provide an additional layer for converting SOAP calls to operation invocations on session instances, by a Web Services client. This additional layer is implemented using the Axis framework [2].

For every workspace and object type, a separate session component will be provided in the access layer. The core object management operations provided by every component include *createObject*, *copyObject*, *deleteObject*, *findAll* and *findByObjectId* and the *get/set* operations for the defined attributes. Session components for versioned object types provide operations *createSuccessor*, *deleteVersion*, *merge*, *getRoot*, *getAncestors*, *getSuccessors*, and *getAlternatives*. Since an instance of a session component is never bound to a particular object, operations that perform work on an existing object require the object's *globalId* as parameter, e.g., given a session component instance *menuAccess* for the object type *Menu* from Fig. 5, *menuAccess.createSuccessor(393218)* will create a successor to the menu with the *globalId 393218* and return the *globalId* of the successor version to the client. In this way, a single session component

instance can answer diverse operation calls for different objects of the same type thus saving server-side resources. Operations also come in variants that provide value-based return, e.g., *createSuccessor_Value*, returning a serializable representation of an object to the client.

Session components in the access layer also provide operations for managing and traversing relationships. For example, invoking operations *addCategory* or *removeCategory* on a session component instance *teaserAccess* for the object type *Teaser* creates or deletes a relationship among a teaser and a menu (note that *Menu* is referred to from the *Teaser* using the role name *category*). Both operations maintain CVCs on the corresponding relationship ends. The operation *getCategory* performs a filtered navigation from the teaser to the menu, returning either the pinned version from each connected CVC on the side of the menu or the latest version, in case the pinned version does not exist. For a session component instance a current workspace can be set (this is the reason for session components being stateful) in case an attachment relationship type has been defined among a corresponding object type and the workspace type. For example, the operation *setCurrentEditorialDepartment* can be invoked on *teaserAccess*. With a current workspace set, the operation *getCategory* will return only the connected versions of the menu that are attached to this workspace. The operations *getCategoryPinned*, *pinCategory*, and *unpinCategory* are used for explicitly navigating to the pinned menu versions and manipulating the pin settings. Unfiltered navigation is supported using the operation *getCategoryUnfiltered*.

Taking advantage of the specified operation propagation properties, the generator hardwires the application logic for propagation directly in the implementation of session components in the access layer. For the example in Fig. 5, creating a copy of an article automatically creates copies of associated teasers and images and connects these copies with relationships.

3.3 Content Browser

In addition to the programmatic access described in the previous section, the versioning system can also be accessed using a content browser. This access is convenient for manually invoking the operations of a versioning system. The browser is a Web application based on the JSP Model 2 Architecture [17]. It consists of generated JSPs, which contain no Java scriptlet code and serve only as views to the requested information. In addition, for each object type, we generate a servlet that takes the role of a Web controller. These servlets invoke operations on the access layer and dispatch the returned information to the JSPs.

3.4 Example

To illustrate the concepts described by Sect. 3.1-3.3, we take a look at an example scenario for the system generated from the information model illustrated in Fig. 5. Suppose that in the content browser, the user gets an overview of a specific article version. From this overview, the *createSuccessor* operation is invoked on this version. The invocation

```
1:  public ArticleLocal createSuccessor_Local(ArticleLocal object, HashMap recursionMap) throws Exception {
2:      if (!object.getFrozen()) { // ... Exception throwing code }
3:      if (getSuccessors_Local(object).size() >= 10) { // ... Exception throwing code }
4:      ArticleLocal newCopy = null;
5:      newCopy =  mHomeInterface.create(object.getObjectId());
6:      object.getSuccessors().add(newCopy);
7:      newCopy.setTitle(object.getTitle());
8:      newCopy.setText(object.getText());
9:      newCopy.setAuthor(object.getAuthor());
10:     newCopy.setDate(object.getDate());
11:     recursionMap.put(object.getGlobalId(), newCopy);
12:     TeaserAccessLocal teaserAccessBean = getTeaserAccessBean();
13:     for (Iterator it=getTeaser_Local(object).iterator(); it.hasNext();) {
14:         TeaserLocal linked = (TeaserLocal)it.next();
15:         if (linked.getFrozen()) {
16:             TeaserLocal newLinked = null;
17:             if (!recursionMap.containsKey(linked.getGlobalId()))
18:                 newLinked = teaserAccessBean.createSuccessor_Local(linked, recursionMap);
19:             else newLinked = (TeaserLocal)recursionMap.get(linked.getGlobalId());
20:             addTeaser_Local(newCopy, newLinked);
21:         }
22:     }
23:     return newCopy;
24: }
```

Fig. 6. *createSuccessor_Local* operation for the *Article* object type

is accepted by the Web controller servlet for the *Article* object type. The servlet calls the method *createSuccessor_Value* on the stateful session bean object in the access layer that is responsible for the *Article* object type. As mentioned in Sect. 3.2, this method returns a serializable representation of the newly created version. The method relies on the method *createSuccessor_Local*, which actually creates the successor version. This method is illustrated by Fig. 6. *newCopy* (line 5) is a reference to an entity bean for the newly created version. Lines 7–10 copy the attribute values to this version. Lines 11–22 implement the propagation of the *createSuccessor* operation from an article version towards connected teaser versions. A hash map *recursionMap* is used to keep track of already visited versions to prevent cycles in operation propagation.

3.5 Generation Process

The generation process in VS-Gen is based on a set of code templates. Nearly all existing template-based code generation approaches work in the same way. First, a template that consists of static parts, placeholders for user-defined values, and control flow statements that drive the evaluation of the output of the template is prepared. Afterwards, in a process called *merging*, a context of values that will replace placeholders is prepared and the control flow of the template is executed by a template engine. To separate it from the *static code* parts in the template, we refer to context references and control flow as *metacode*. The template engine used in VS-Gen is Velocity [1]. No generated code is produced by means other than templates. Table 1 gives an overview of the 25 code templates used in VS-Gen and the implementation parts they generate.

Code generation from UML models often requires flexible access to model information that is dispersed across many model elements due to fine-grained nature of the UML Metamodel. Following the idea described by Sturm et al. [18], this problem is solved by implementing the so-called *prepared classes* that aggregate model information from diverse model parts. VS-Gen uses XMI to import an information model from

Table 1: Velocity templates used by VS-Gen

Temp-Id	Template name	Used to generate...
1	VSObjAccBean	a class implementation for components in the access layer
2	VSObjAccLocal	local component and home interfaces for components in the access layer
3	VSObjAccLocalHome	
4	VSObjAccRemote	remote component and home interfaces for components in the access layer
5	VSObjAccRemoteHome	
6	VSObjBean	a class implementation for components in the entity layer
7	VSObjLocal	local component and home interfaces for components in the entity layer
8	VSObjLocalHome	
9	VSObjRemote	remote component and home interfaces for components in the entity layer
10	VSObjRemoteHome	
11	VSObjValue	serializable representations of object types
12	ControllerServlet	request dispatcher servlets (Web controllers) for the content browser
13	ShowInstance	a JSP page that displays information on a stored object
14	ShowTypeInfo	a JSP page that displays type information (metadata, i.e., attributes and relationship types) for an object type
15	ShowList	a JSP page that displays the matching objects as a result of a finder method or a navigation operation
16	ShowIndex	a JSP page that displays the navigation bar in the content browser
17	Web-xml	a Web deployment descriptor
18	EJB-jar-xml	an EJB deployment descriptor
19	Application-xml	a deployment descriptor for the entire enterprise application
20	AS-application-xml	an application server specific deployment descriptor for the application
21	Deploy-wsdd	a Web services deployment descriptor
22	Undeploy-wsdd	a file to undeploy previously deployed Web services
23	VSObjService	proxy classes for redirecting Web services calls to the access layer
24	TestClient	a Web services based test client
25	Build-xml	a script to compile and deploy the generated implementation

a UML tool to the NSUML [16] in-memory UML repository. Before invoking the template engine, the model is analyzed in this repository to instantiate the prepared classes and fill the instances with model information. Afterwards, the instances are put in the Velocity context.

As an example, Fig. 7 illustrates a part of the template *VSObjAccBean* that has been used to generate the implementation of the *createSuccessor_Local* method from Fig. 6. Velocity statements begin with the *#* character. The *$* character is used to retrieve a value from the context. The reference *class* (see, for example, lines 1 and 8) represents an instance of a prepared class that aggregates information from the UML Metamodel classes *Class*, *Generalization*, *Stereotype*, *TagDefinition*, and *TaggedValue*. Since the multiplicity of the relationship end *teaser* that belongs to the relationship *R3* in Fig. 5 is *many*, the statements in lines 23–34 are used in the generated example from Fig. 6.

4 Evaluation

It is difficult to come up with a realistic estimate of how large individual features of versioning systems are just by examining the feature models and the UML profile from Sect. 2. For this purpose, in Sect. 4.1, we first examine the properties of the example versioning system generated for the information model from Fig. 5. Note that this examination does not apply to how the system is generated, although we used the generator to gradually add new features and examine the differences in the generated code. The results give an orientation of how labor-intensive a manual implementation of the

```
1:  public ${class.name}Local createSuccessor_Local(${class.name}Local object, HashMap recursionMap)
2:  throws Exception  {
3:      if (!object.getFrozen()) { // ... Exception throwing code }
4:  #if ( $package.getIntegerTaggedValue("maxSuccessors", -1) > 0 )
5:      if (getSuccessors_Local(object).size() >=
6:          ${package.getIntTaggedVal("maxSuccessors", -1)}) { // ... Exception throwing code }
7:  #end
8:      ${class.name}Local newCopy = null;
9:      newCopy =  mHomeInterface.create(object.getObjectId());
10:     object.getSuccessors().add(newCopy);
11: #foreach( $attribute in $class.attributes )
12:     newCopy.set${attribute.nameUpperCase}(object.get${attribute.nameUpperCase}());
13: #end
14:     recursionMap.put(object.getGlobalId(), newCopy);
15: #foreach( $associationEnd in $class.associationEnds )
16: #if ( $associationEnd.association.hasStereotype("RelationshipType") &&
17:         ($associationEnd.oppositeEnd.getTaggedVal("propCreateSuccessor") == "true") &&
18:         ($associationEnd.oppositeEnd.participant.hasStereotype("VersionedObjectType")  ||
19:             $associationEnd.oppositeEnd.participant.hasStereotype("WorkspaceType") ) )
20: ${associationEnd.oppositeEnd.participant.name}AccessLocal ${associationEnd.oppositeEnd.name}AccessBean =
21:     get${associationEnd.oppositeEnd.participant.name}AccessBean();
22: #if ( $associationEnd.oppositeEnd.isMultiValued() )
23:     for (Iterator it=get${associationEnd.oppositeEnd.nameUpperCase}_Local(object).iterator(); it.hasNext();) {
24:         ${associationEnd.oppositeEnd.participant.name}Local linked =
25:             (${associationEnd.oppositeEnd.participant.name}Local)it.next();
26:         if (linked.getFrozen()) {
27:             ${associationEnd.oppositeEnd.participant.name}Local newLinked = null;
28:             if (!recursionMap.containsKey(linked.getGlobalId())) newLinked =
29:                     ${associationEnd.oppositeEnd.name}AccessBean.createSuccessor_Local(linked, recursionMap);
30:             else newLinked =
31:                     (${associationEnd.oppositeEnd.participant.name}Local)recursionMap.get(linked.getGlobalId());
32:             add${associationEnd.oppositeEnd.nameUpperCase}_Local(newCopy, newLinked);
33:         }
34:     }
35: #else
36:     ${associationEnd.oppositeEnd.participant.name}Local
37:         linked${associationEnd.oppositeEnd.participant.nameUpperCase} =
38:             get${associationEnd.oppositeEnd.nameUpperCase}_Local(object);
39:     if (linked${associationEnd.oppositeEnd.participant.nameUpperCase}.getFrozen()) {
40:         ${associationEnd.oppositeEnd.participant.name}Local
41:             new${associationEnd.oppositeEnd.participant.nameUpperCase} = null;
42:         if (!recursionMap.containsKey(
43:             linked${associationEnd.oppositeEnd.participant.nameUpperCase}.getGlobalId()))
44:             new${associationEnd.oppositeEnd.participant.nameUpperCase} =
45:                 ${associationEnd.oppositeEnd.name}AccessBean.createSuccessor_Local(
46:                 linked${associationEnd.oppositeEnd.participant.nameUpperCase}, recursionMap);
47:         else
48:             new${associationEnd.oppositeEnd.participant.nameUpperCase} =
49:                 (${associationEnd.oppositeEnd.participant.name}Local)recursionMap.get(
50:                 linked${associationEnd.oppositeEnd.participant.nameUpperCase}.getGlobalId());
51:         add${associationEnd.oppositeEnd.nameUpperCase}_Local(newCopy,
52:             new${associationEnd.oppositeEnd.participant.nameUpperCase});
53:     }
54: #end ## End of #if ( $associationEnd.oppositeEnd.isMultiValued() )
55: #end ## End of #if ($associationEnd.association.hasStereotype("RelationshipType") && ...
56: #end ## End of #foreach( $associationEnd in $class.associationEnds )
57:     return newCopy;
58: }
```

Fig. 7. Part of the template *VSObjAccBean* used for generating *createSuccessor_Local* method

features would be. To evaluate the generation approach itself, Sect. 4.2 examines the properties of the templates presented by Table 1.

4.1 Properties of the Example Versioning System

We analyzed the code generated with VS-Gen for the sample information model from Fig. 5 by trying to trace different code parts back to the corresponding configuration concepts from the UML profile (see Table 2). In terms of the lines-of-code (LOC) mea-

Table 2: Parts of the generated implementation

Implementation part	%LOC
Object and workspace types	**62.47%**
- Article	14.35%
- EditorialDepartment	13.30%
- Image	8.88%
- Menu	13.04%
- Teaser	12.90%
Relationship types	**21.93%**
- R1 (Menu-Menu)	6.44%
- R2 (Menu-Teaser)	7.11%
- R3 (Teaser-Article)	6.61%
- R4 (Article-Image)	1.77%

Implementation part	%LOC
Attachment rel. types	**11.40%**
- R5 (EditorialDep-Menu)	3.68%
- R6 (EditorialDep-Teaser)	3.78%
- R7 (EditorialDep-Article)	3.93%
Remaining parts	**4.21%**

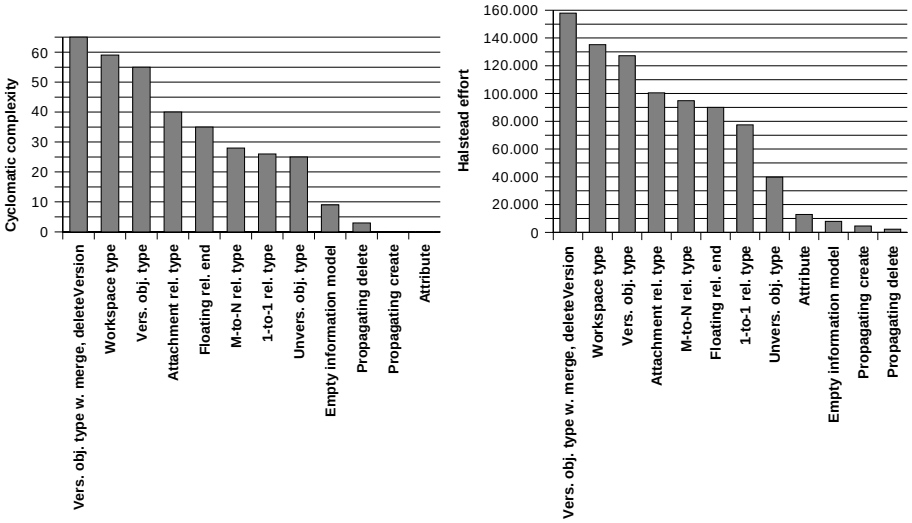

Fig. 8. Cyclomatic complexity and Halstead effort for individual concepts / features

sure, the largest part of the generated code (62.47%) belongs to the object and workspace types. As evident from Table 2, an unversioned object type (*Image*) requires substantially less code than versioned types and workspaces. A unidirectional relationship type *R4* (between *Article* and *Image*) with no floating ends is far easier to support than other relationship types. Also note that the three attachment relationship types that connect the workspace type *EditorialDepartment* to object types *Menu*, *Teaser*, and *Article* are less demanding than regular relationships types. The remaining percentage of code (4.21%) represents the counters for managing *objectIds* and *versionIds* and can not be directly ascribed to any configuration concept.

We were also interested in the values for *McCabe's cyclomatic complexity* [13] and *Halstead effort* [8] that can be attributed to individual concepts and features. The results of this analysis are illustrated by Fig. 8. The two values (complexity and effort) for an empty information model were obtained by generating a versioning system from a model with no object, relationship, and workspace types. Afterwards, we added an unver-

sioned object type, versioned object type (separately with *merge* and *deleteVersion* operations), and a workspace type and observed the differences in comparison with the values obtained for an empty model. Starting from an object type and a workspace type, we added an attribute, an attachment relationship type, a regular one-to-one relationship type, and a regular many-to-many relationship type to the model and observed the differences. Afterwards, we made both ends of the many-to-many relationship type floating and halved the obtained complexity and effort differences to obtain the values that can be ascribed to a single floating relationship end. Finally, for an existing relationship, we additionally chose delete propagation and create propagation. As evident from Fig. 8, the largest complexity and effort values are obtained for versioned object types and workspace types (in contrast to this, an unversioned object type is less demanding than a one-to-one relationship type). A floating relationship end alone adds more complexity and a bit less effort than originally required for a many-to-many relationship type. The values for operation propagation are extremely low.

4.2 Properties of the Templates

We investigated the templates from Table 1 using the following measures: *references* to prepared classes in the context, *if-statements*, *for-loops*, *statements*, *McCabe's cyclomatic complexity*, *Halstead effort*, and *LOC*. Since we were interested in the properties of the metacode contained in the templates, we counted every static output within a template as a single "atomic" metacode statement. In this way, for example, only if-statements of the metacode are counted, but not the if-statements in the Java code wrapped by the metacode. The statement count was obtained as the sum of references, if-statements, for-loops, and atomic outputs of static code. Altogether, 288 kB of template files contain 3013 references to prepared classes, 542 if-statements, 128 for-loops, and 6670 statements. The chart in Fig. 9 illustrates the distribution of the cyclomatic complexity within the templates. Not surprisingly, the template used to generate the session components in the access layer (which contain most application code and are very prone to feature selections) proves as the most complex one.

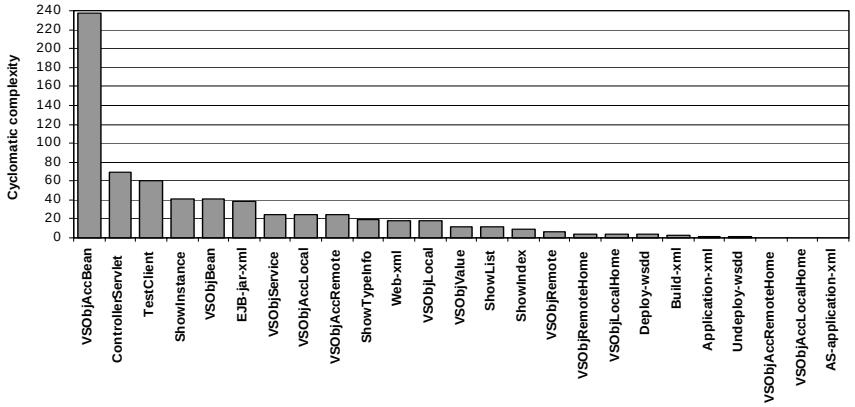

Fig. 9. Cyclomatic complexity of the templates

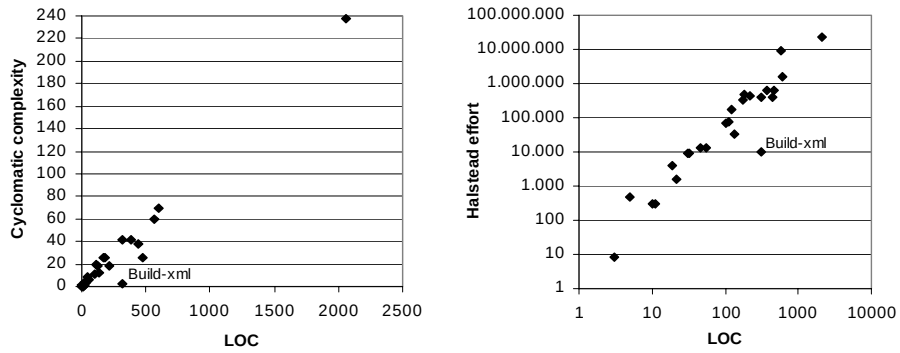

Fig. 10. Correlation between the cyclomatic complexity / Halstead effort and LOC

We were also interested whether there is a correlation between the LOC-value and the cyclomatic complexity and the Halstead effort of the metacode. As illustrated by Fig. 10, the cyclomatic complexity grows in a linear proportion to the LOC-value of a template. A steady growth trend can be observed for the Halstead effort if we use logarithmic scales for both axes. The specially marked outlier is the template *Build-xml*. This template (for generating the compile and build script for a versioning system) contains only a minor part of metacode in proportion to its static parts.

5 How Fast are Generated Versioning Systems?

There are two ways of integrating a versioning system with clients. Development environments usually contain a set of existing clients (tools), each with a private storage manager with a logical data model not necessary corresponding to that of the versioning system. In this case, *translators* [4] (also called *data exchange switches*) that convert the data between the versioning system and the storage managers need to be developed. In the second case, the versioning system is used interactively by the tools. In this case, users will develop new tools using the generated API or wrap this API so that it can be used with the existing tools. Long-lasting accesses to the versioning system in both cases, where a large number of stored objects is traversed, are disturbing for developers. A fast versioning system assures short modification, build, and test cycles in any kind of development process that uses versioned data.

Do generated versioning systems provide better performance that generic ones? To answer this question, we developed an alternative implementation of the product line in form of a *framework*. In the framework approach, the user implements a desired versioning system by extending the framework superclasses with classes specific to the information model and providing call-back methods that will be invoked by the framework through reflection. What are the differences between the systems obtained in two different ways, i.e., generated by VS-Gen or obtained by instantiating the framework? As described in Sect. 3, template metacode assures a wide range of optimizations, e.g.,

materializing the pinned version in a CVC or hardwiring operation propagation settings in the implementation of the access layer. Unfortunately, these optimizations raise the complexity of the database schema and the generated code. This does not prove a problem as long as the system is accessed by the client only through the generated API and as long as no adaptation of the generated code is required. However, if a user wanted to query the database of the versioning system directly using SQL, this would require a complete understanding of optimizations performed by the template metacode (also see our conclusion in Sect. 7 for an alternative way of dealing with this problem). In addition, due to hardwiring propagation settings in the implementation of the access layer, the settings are difficult to trace down and modify. For this reason, when implementing the framework, we tried to factor out a great deal of versioning functionality to *generic parts* and implement them so that they can be used with any information model, increasing the simplicity of the obtained system (especially the database schema) without loosing too much on performance. The differences between systems generated by VS-Gen and systems developed by extending the framework are summarized in Table 3.

We took advantage of the fact that all API calls supported by a generated system are also supported by a framework-based system (in fact, a framework-based system will usually provide a somewhat more elaborate API for an information model because, for example, all object types are versioned). This means that clients that run against a generated system run with no modifications against a framework-based system. In our performance comparison, we used the information model from Fig. 5 for a generated and

Table 3: Differences between generated and framework-based versioning systems

Concept / feature	**VS-Gen**	**Framework**
Versioning information (predecessor to a given version, information whether a version is frozen)	*The information is directly integrated in the entity components that represent object types.*	*Stored by a special table used by all object types. The versioning information is represented as an additional entity component.*
Merging information (what versions have been merged with other versions)	*Implemented as a separate reflexive many-to-many relationship on an entity component for a particular object type.*	*Implemented as a reflexive relationship on the entity component that represents versions.*
Unversioned object types	*Supported.*	*Not supported - all object types are versioned.*
Non-floating relationship ends	*Supported.*	*Not supported - all ends are floating.*
Relationship types	*Implemented separately for every relationship type. Represented as relationships among the entities that represent object types, allowing context-based prefetch.*	*All relationships are stored by a special table. A relationship is represented as an additional entity component.*
Latest version of a CVC	*Materialized as a separate relationship between entities and thus immediately available in filtered navigation. If the multiplicity of the floating end is one, only one join will be required.*	*The candidate versions, determined within the relationship table, have to be scanned to determine the latest version.*
Pinned versions	*Implemented separately for every floating relationship end. Represented as relationships between the entities.*	*All pin settings are stored by a special table. A pin setting is represented as an additional entity component.*
Attachment relationship types	*Implemented separately for every attachment relationship type defined in the information model.*	*Stored by a special table. Attachments are represented as additional entity component.*
Checkout locks	*Available as a one-to-many relationship between the workspace type and the object type.*	*Stored by a special table. Checkout locks are represented as an additional entity component.*
Operation propagation settings	*Hardwired directly in the session components in the access layer.*	*Stored in a special table that can be modified for a running versioning system. The table needs to be queried for every invocation of an operation that might propagate.*

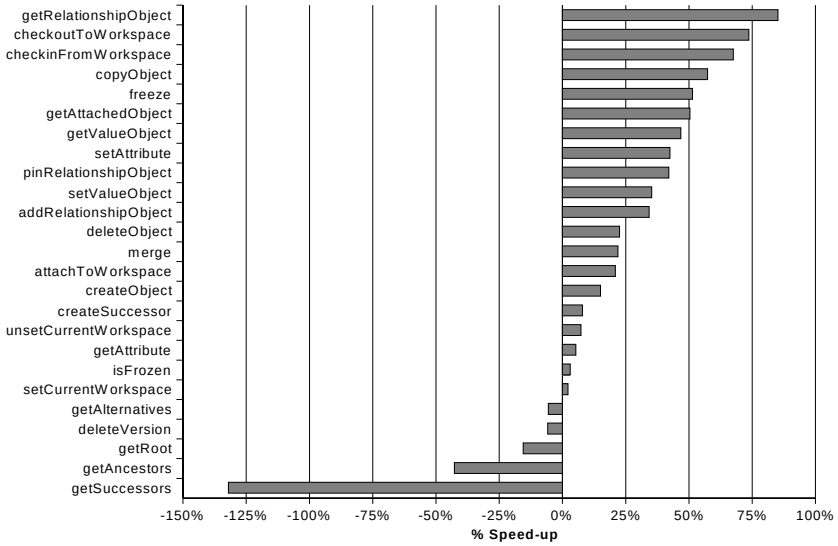

Fig. 11. Speed-up of a generated towards a framework-based versioning system

framework-based versioning system. A benchmark client that was used to simulate a typical content-management scenario for this information model carried out 192,148 operations. A detailed comparison of execution times (we subtracted the communication times between the client and server) for diverse operations is given by Fig. 11.

The speed-ups illustrated in Fig. 11 were calculated as the ratio $(t_{fw}\text{-}t_{gen})/t_{fw}$, where t_{fw} represents the time required by the framework-based implementation and t_{gen} the time required by the generated implementation. The entire run of the benchmark took 505.30 seconds for the framework-based implementation and 370.86 seconds for the generated implementation, yielding a 26.61% overall speed-up of the generated implementation. Note that execution times in Fig. 11 have been categorized, e.g., *pinRelationshipObject* represents all pinning operations carried out for objects and relationships of diverse types defined by the information model in Fig. 5. The times used for t_{fw} and t_{gen} were then obtained as median execution times within a category (median times are used instead of average times to reduce the effect of outliers.) As evident from Fig. 11, version graph navigation operations (*getAlternatives*, *getRoot*, *getAncestors*, and *getDifferences*) and the *deleteVersion* operation prove more efficient for a framework-based system due to differences in representing versioning information as described in Table 3.

Is improved performance the only reason for implementing a generator? Our experience shows that instantiating the framework by writing derived classes is a very labor-intensive task. It requires the developer to get familiar with the framework and is also very error-prone, e.g., due to naming conventions the call-back methods need to follow. For the information model from Fig. 5, 45 derived classes with a total 3822 LOC and a Halstead effort of 12,490 were required. So even for the framework-based approach used in a commercial setting, we claim that a simple generator for framework instanti-

ation would be necessary. This increases the actual cost of developing the framework when considering the fixed costs associated with both generators. These costs are independent of the complexity of generated code. For example, the facility for analyzing information models needs to be developed in both cases.

6 Related Work

A good analysis of diverse existing systems with versioning support is a prerequisite for implementing a generator for such systems. As a result, our goal was to come up with the following results.

- A common terminology (a domain dictionary) for our domain model.
- Case studies of how different features are implemented in existing generic systems.
- A strategy to making different features compatible in a single domain implementation, e.g., in our case, using a template-based generation approach.

Most of the existing work deals with the first two points. For example, an extensive overview of version and configuration management is provided by Katz [11], who proposes a *unified framework for version modeling* in engineering databases. The author analyzes a variety of existing terminologies and mechanisms for representing versioned data. A lot of recent applications to versioning are used for versioning hypermedia content. Whitehead [22] analyzes this domain and gives an overview of existing systems.

Some examples of versioning systems include CVS/RCS [7], Microsoft Meta Data Services [15], Unisys UREP [21], IBM Rational ClearCase [10], and IBM XML Registry [9]. While early systems like CVS/RCS use a file system for storage, most recent systems, like VS-Gen, rely on a database to allow the structuring of information models either as relational or OODBMS schemas, queries, and transactions. Conradi and Westfechtel [6] examine a variety of version models in both commercial and research systems and discuss fundamental concepts of the domain, such as *versions*, *revisions*, *variants*, *configurations*, and *changes*. Whitehead and Gordon [23] use *containment data models* (a specialized form of the ER model) to analyze 11 configuration management systems (including CVS and ClearCase). A comparison of these surveys to VS-Gen reveals that, at the moment, our domain model covers merely a portion of existing version models. Following the results from this work, we plan to gradually improve the VS-Gen's UML profile and domain implementation to support new features.

Versioning of metadata (relational and XML schemas, interface definitions for operations, etc.) has become significant for data and application integration approaches. In this cases, metadata (which is subject to change over time and thus needs to be versioned) is needed for analyzing and dealing with syntactic and semantic heterogeneity of diverse data sources that need to be integrated (see [12]). In case a versioning system supports a fine-grained navigational access among stored objects for the clients, much like the systems generated by VS-Gen, another common term for such a system is *repository* (see Bernstein [4] for the description of additional functionality that is usually provided by repositories, e.g., workflow management for tracing the lifecycle of stored objects). Although the majority of existing approaches support the generation of implementation parts that are specific for the user-defined information model, they restrain

from treating versioning functionality in terms of features that could be selected by the user. This usually leads to framework-based solutions similar to the one we presented in Sect. 5, with a decreased performance and only minor possibilities to customize the versioning semantics.

In our current implementation, the merge operation merely connects two branches in the version tree, relying on the user to resolve the semantic conflicts between the versions. However, the topic of automatic merging has already been considered by some authors. For example, Melnik et al. [14] discuss the definition of a *reintegrate operator* that can be used for model merging. The operator relies on automatic discovery of structural and semantic model correspondences. We plan to explore the relevance of these results to VS-Gen in our future work.

7 Conclusion and Future Work

This paper presented VS-Gen which is our approach to template-based generation of versioning systems from information models branded with system features selected by the user. In comparison to the framework-based solution that relies on a series of generic implementation components, a generated versioning system demonstrated an improved performance of 27% measured by our benchmark. The time spent on the VS-Gen project can be summarized as follows. A development team of two with a background in version management spent four months analyzing and comparing different versioning systems to come up with the domain model. The development of the UML profile, model analysis facility, prepared classes and the templates was carried out in an iterative fashion by adding a specific feature to these parts, adjusting the implementation of existing features, and carrying out the tests. These iterations took six months, the development of the templates taking the effective portion of approx. 75% of this time. The effective time for getting familiar with different technologies we use in VS-Gen (NSUML, Velocity, and Axis) is estimated to one month. Both developers had a strong background in J2EE, UML profiles, and XMI. Due to our existing experience with the implementation of versioning systems we generate and simpler optimization requirements, the development of the framework (developed after the generator part was completed) took less than three months.

The following are the lessons we learned from VS-Gen.

- Current IDEs lack sufficient support for developing templates. The minimal requirement would include syntax highlighting to easily separate code from metacode. Velocity templates 13–16 from Table 1 were especially difficult to develop since they generate JSP pages, which themselves act as templates for HTML output. Further requirements would include a quick examination of generated code by filling in the context values directly in the IDE with no need to start the entire generation process. Adding new features to our domain often required a reimplementation of existing templates. A good IDE for template-based development should support a clear overview of what parts of a template are related to a specific feature.
- Many templates were developed by first considering examples of what needs to be generated for a selection of features and afterwards generalizing these examples to

a template. The most complex case for this were different examples for the combination of floating/non-floating relationship ends with their multiplicities. These combinations affect different parts of access components that deal with relationship traversal, creation, and deletion of a relationship. We assume that this kind of example-based development could be automated to some extent by tools that compare different examples and identify varying pieces of code.

- Templates for generating code with a great deal of application logic are especially difficult to develop and test. Using the cyclomatic complexity scale proposed by [5], the templates for generating access components, the controller servlet, and the test client would fall into the category of programs with very high risk that are practically untestable. A solution would be to break down the template into many parts that can be used in a superordinated template. In case a part is used more than once, this allows reuse of code and metacode. Velocity supports the described reuse by *include*, *parse*, and *macro* directives [1]. However, in VS-Gen, no repeating parts of metacode occur in the first place, making metacode refactoring impossible. Thus the only solution is to refactor the code that needs to be generated to many parts (classes) and implement many small templates for these parts.

Our benchmark results are to be treated with care. The overall speed-up of a generated versioning system towards a framework-based system depends on the concrete selection of features and proportions among the categories of operations carried out by the benchmark. This means that the *categories* illustrated in Fig. 11 are more relevant for discussing the benefits of generated systems than the overall speed-up of 27%. To our knowledge, no well-accepted benchmarks for versioning systems exist, probably due to a large variety of existing versioning models. As mentioned by [4], in the most simple case, OODB benchmarks can be used for measuring the performance of navigational access operations in a versioning system. However, this excludes versioning operations and the effects of operation propagation that are also interesting to us.

Our work fails to answer the most important question: *How many* versioning systems implemented manually justify the effort required for developing the templates for the generative approach? In answering this question, it proves unavoidable to come up with a measure that reasonably combines the properties of the metacode and static code in a template. Having not solved this problem, we simply state that even for the case of a small information model, such as the one in Fig. 5, the LOC-value for the generated versioning system is 2.7-times greater than the LOC-value of the templates, although this result is to be treated with utmost care. Our future work will focus on a detailed examination of template metacode to determine a set of appropriate measures that can be used for evaluating the metacode's properties.

Presenting developers with a generated middleware API for accessing the data in the versioning system is not the only possible way of tackling the problem. For this reason, in a related project, we are developing a domain-specific SQL-like language for dealing with versioned data. Data definition statements in this language, which describe the information model and the selection of features are translated to SQL-DDL statements. Query and update statements are translated to SQL-DML statements. The translation takes place in a special database driver that wraps the native driver.

Company mergers often require not only integration of enterprise data but also integration of engineering data for the products, which is usually versioned. Understanding variability in different versioning models is a key prerequisite for such integration. In our future work in this area, we want to extend VS-Gen with a support for generative development of wrappers to utilize the integration of legacy versioning systems.

References

1. The Apache Jakarta Project: Velocity, available as: http://jakarta.apache.org/velocity/
2. The Apache Web Services Project - Axis, available as: http://ws.apache.org/axis/
3. Bernstein, P.A.: Repositories and Object-Oriented Databases, in: SIGMOD Record 27:11 (1998), pp. 34-46
4. Bernstein, P.A., Dayal, U.: An Overview of Repository Technology, in: Proc. VLDB 1994, Santiago, Sept. 1994, pp. 707-713
5. Carnegie Mellon University, Software Engineering Institute: Software Technology Roadmap - Cyclomatic Complexity, available as: http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html
6. Conradi, R., Westfechtel, B.: Version Models of Software Configuration Management, in: ACM Computing Surveys, 30:2 (1998), pp. 232-282
7. CVS - Concurrent Versions Systems - The Open Standard for Version Control, available as: http://www.cvshome.org/
8. Halstead, M.H.: Elements of Software Science, Elsevier, 1977
9. IBM Corp., IBM alphaWorks: XML Registry, available as: http://www.alphaworks.ibm.com/tech/xrr
10. IBM Corp.: IBM Rational ClearCase, available as: http://www.ibm.com/software/awdtools/clearcase/
11. Katz, R.H.: Toward a Unified Framework for Version Modeling in Engineering Databases, in: ACM Computing Surveys 22:4 (1990), pp 375-409
12. Madhavan, J., Bernstein, P.A., Rahm, E.: Generic Schema Matching with Cupid, in: Proc. VLDB 2001, Rome, Sept. 2001, pp. 49-58
13. McCabe, T.J.: A Complexity Measure, in: IEEE Transactions on Software Engineering 2:4 (1976), pp. 308-320
14. Melnik, S., Rahm, E., Bernstein P.A.: Developing Metadata-Intensive Applications with Rondo, in: Journal of Web Semantics, 1:1 (2003)
15. Microsoft Corp., Microsoft Developer Network: Meta Data Services Architecture, available as: http://msdn.microsoft.com/
16. Novosoft Inc.: NSUML – Novosoft Metadata Framework and UML Library, available from: http://nsuml.sourceforge.net/
17. Seshadri, G.: Understanding JavaServer Pages Model 2 Architecture: Exploring the MVC Design Pattern, JavaWorld, Dec. 1999, available from: http://www.javaworld.com/
18. Sturm, T., von Voss, J., Boger, M.: Generating Code from UML with Velocity Templates, in: Proc. UML 2002, Dresden, Oct. 2002, 150-161
19. Sun Microsystems, Inc.: Enterprise JavaBeans Specification, v2.1, Nov. 2003
20. Sun Microsystems, Inc.: Java 2 Platform Enterprise Edition Specification, v1.4, Apr. 2003
21. Unisys Universal Repository: Tool Builder's Guide, Unisys Corp., 1999.
22. Whitehead, J.E.: An Analysis of the Hypertext Versioning Domain, Ph.D. dissertation, Univ. of California, Irvine, Sept. 2000
23. Whitehead, J.E., Gordon, D.: Uniform Comparison of Configuration Data Models, in: Proc. SCM-11, Portland, May 2003, pp. 70-85