Model-Driven Development of Versioning Systems: An Evaluation of Different Approaches

JERNEJ KOVSE

THEO HÄRDER

Dept. of Computer Science, University of Kaiserslautern, P.O. Box 3049, D-67653 Kaiserslautern, Germany

E-mail: kovse@relacija.com, haerder@informatik.uni-kl.de
Telephone: +386-2-803-6140
Fax: +386-2-803-6141

Abstract. This paper analyzes the domain of versioning systems and compares three approaches to generating such systems from models. In the first approach, we define a domain-specific modeling language as a lightweight extension of UML and use templates to generate a middleware-based versioning system. In the second approach, we define a domain-specific data definition and manipulation language that can be embedded in versioning system applications and map this language to SQL. In the third approach, we assemble the specification of a versioning system in Executable UML using model transformations. The presented approaches are evaluated from the perspective of developer productivity and performance of generated systems.

Keywords: *Model-driven Development, Versioning Systems, Code Generation, Model Transformations*

1 Motivation

Model-driven software development (*MDSD*) is a special kind of *generative software development* [9] with the following characteristics.

- System specifications that serve as generation input are referred to as models and can be preferably visualized by CASE tools.
- There is at least some organization-wide agreement on (i) metamodels for the models or (ii) meta-metamodels used to define custom metamodels. Agreement on metamodels allows an existing set of tools, e.g., CASE tools, metrics tools, and generators, to be used in application engineering. Agreement on meta-metamodels allows an automated generation of such tools, which is usually referred to as method engineering [7].
- Optionally, not only the generation phase, but also the specification phase is automated through model transformations.

MDSD recently gained a lot of research and industry attention, as evident from the following work.

- Specifications and standardization proposals, e.g., *MDA* [23], *QVT* [28], or *SysML* [36].
- New generation of tools for modeling, metamodeling, and method engineering [7,32].
- Work on model transformations, e.g., *GReAT* [1] or *BOTL* [19].
- Work on generating code from models, e.g., *Jamda* [5] or code generation by *VTL* [35].
- Processes for efficient use of models in development, e.g., *agile modeling* [2].

In this work, we adopt the classification proposed by Frankel [12] who identifies four approaches to model-driven development.

- *Approach 1—a standardized metamodel.* In this approach, a standardized and well accepted metamodel such as the UML Metamodel is used when developing models. A drawback of this approach is that standardized metamodels are usually general-purpose rather than domain-specific. This requires a great number of model elements to express the same idea and thus implies long modeling cycles, unless the process is supported by model transformations. The resulting models are large and difficult to communicate between the developers.
- *Approach 2—custom metamodel.* In this approach, a developer uses a predefined meta-metamodel to define a custom metamodel. The benefit of this approach is that the custom metamodel is usually domain-specific and thus contains a small set of powerful elements that can be used to describe the systems from the observed domain in a compact way.
- *Approach 3—heavyweight metamodel extensions.* In this approach, a custom metamodel is obtained as an extension of an existing metamodel. Custom modeling elements are added to the existing metamodel and connected to its elements by generalizations and associations.
- *Approach 4—lightweight metamodel extensions.* In this approach, the metamodel is extended virtually, i.e., within the model itself and without physically changing the metamodel. This is possible only when the extension capability is predicted by the metamodel. *UML profiles,* supported by the UML Metamodel's package *Extension Mechanisms,* are a well-known example of this approach.

1.1 The Topic of This Work

In this work, we treat versioning systems as a software product line and show that this particular type of systems can be generated from models. Versioning systems are an interesting domain for MDSD and generative development in general because of the following factors.

- There is no default support for versioned data in many DBMSs, thus the generated code represents an important added value. At the same time, the generated versioning code (defined as operations on an object class which supports versioning) can easily be further refined with class-specific operations.
- A manual implementation of versioning functionality on top of a DBMS is a long-lasting and tedious task with lots of redundant code.
- There is a range of different versioning semantics supported in many commercial versioning systems and research prototypes. A large domain covering a variety of different semantics would allow an engineer to describe his own semantics (as a model) and afterwards generate the matching versioning system.

As we show in this paper, the generation process also allows an easy integration of certain performance optimizations in a generated versioning system. These optimizations are typically not available with versioning systems implemented in a generic way.

The paper will outline and compare three different MDSD approaches to generating versioning systems. These approaches differ (i) in the specification style, i.e., the language and the process used to configure a versioning system, (ii) in the generation style, i.e., the mapping of the specification on the implementation, and (iii) in the target platform for the generated code.

1.2 Overview

Sect. 2 will describe the particular type of versioning systems that we deal with in this paper. The feature diagrams for the domain are presented by Sect. 3. Sect. 4 describes the first MDSD approach, which generates a middleware-based versioning system out of a specification based on a UML profile (this is an example of *Approach 4*—the use of lightweight metamodel extensions—described above). Sect. 5 describes the second approach, which allows the mapping of a custom domain-specific data definition and manipulation language to SQL. In our concrete example, the custom language is used to define the database schema for a versioning system in a domain-specific way (data definition) and afterwards query versioned data (data manipulation). Because the definition of the language is facilitated through a custom metamodel, this is an example of *Approach 2* described above. The third approach, described in Sect. 6, allows a fast assembly of models based on a general-purpose metamodel by guiding the user through a series of configuration steps and applying model transformations according to the user's configuration choices. It is an example of *Approach 1* described above. An overview of related work is given by Sect. 7. Finally, the conclusion (Sect. 8) gives a comparison of the three approaches and outlines a set of benefits of MDSD based on our experience.

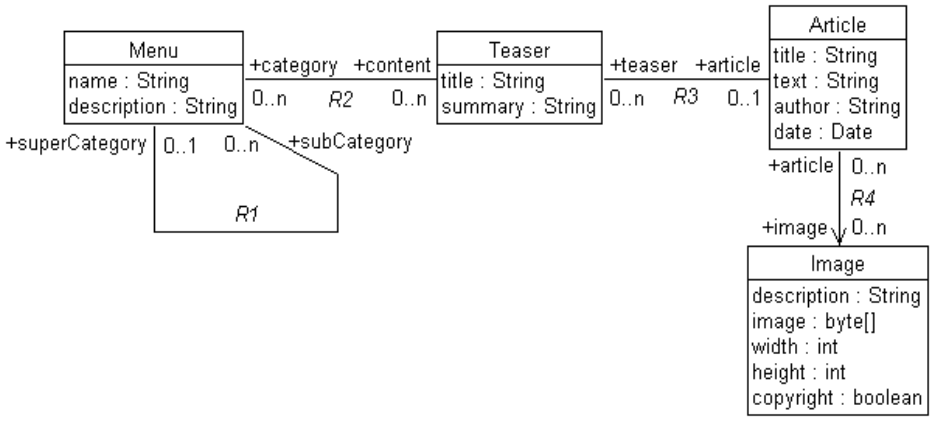

Fig. 1. A sample information model

2 Versioning Systems

Versioning is essential for any kind of domain that requires a representation of past states of data and the possibility to revert to these states. Furthermore, we often want to combine versions into *configurations*, which represent semantically correct compositions of versioned data. In this paper, we deal with object-oriented versioning systems, i.e., systems which represent versioned data as objects and relationships. The properties of objects and relationships are defined by object and relationship types. The definition of object and relationship types is called an *information model*. An example information model from the domain of content management that we also use in Sect. 4 is given by Fig. 1.

A programming model is based on a programming language and sometimes also a component model that is used to access the versioning system by client applications. Sects. 4 and 5 will illustrate two different examples of programming models, a J2EE-based programming model and an programming model based on domain-specific statements embedded in application code. An interface of a programming model can depend on the information model (in this case, it is called a *generated interface*) or be independent of the information model (in this case, it is called a *generic interface*). The following sections describe a selection of features important for understanding the code that we generate in the approaches presented in this paper.

2.1 Objects and Versions

As soon as versioning is supported, there is no difference between the terms *version* and *object* (or *instance*) from the programmer's point of view. A *versioned-object type* (a type which supports versioning) will support both versioning operations, e.g., *createSuccessor*, *freeze*, *getSuccessors*, and *getAncestor*, as well as type-specific operations for object behavior. Three kinds

of integer-based identifiers are used to distinguish between versions in a versioning system. The *object identifier* (*objId*) is a system-wide identifier of a design object. All versions that belong to the same object graph carry the same *objId*. The *version identifier* (*verId*) identifies a particular version and is unique within a version graph. The *global identifier* (*globId*) is used to refer to versions directly and is a combination of *objId* and *verId*.

2.2 Versioned Relationships

Versioning complicates the traversal of relationships among objects. Starting from some origin version, the user navigates towards the target design object. Note that the number of expected objects depends on the multiplicity assigned to the target relationship end. However, because the target objects can be versioned, the following two options should be considered.

- *Unfiltered navigation.* All target versions (they can belong to distinct design objects) that are connected to the origin version are returned to the user as result of navigation.
- *Filtered navigation.* For every subset of target versions that are connected to the origin version and belong to the same design object, exactly one target version is returned. Because all connected target versions represent potential candidates for the selections, the subsets are called *candidate version collections* (*CVCs*). An example of filtered navigation is illustrated by Fig. 2.

In the illustrated example, the *CVC 1* is used when navigating from *v2* of article *a* towards teaser *t*. Note that *v2* of teaser *t* is not included in the *CVC 1*. The presence of versions in CVCs is reflexive—in case v_i of *a* is included in some CVC of v_j of *b*, v_j of *b* is present in some CVC of v_i of *a*. In filtered navigation, only one version is to be returned from every connected CVC. One of the versions in a CVC can be marked as the *pinned version* [4]. In case a pinned version exists, the system will return this version for the CVC. In case a pinned version does not exist, the version to be returned is selected by a rule specified by the user. A very common rule is to return the latest version from the CVC. Suppose that in the example illustrated in Fig. 2, v_3 of *t*

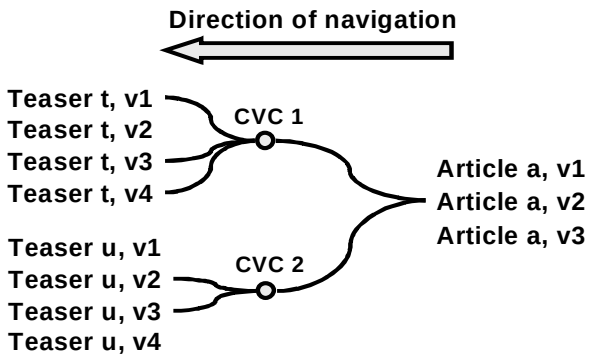

Fig. 2. Example of filtered navigation

is the pinned version for *CVC 1*. There is no pinned version for *CVC 2* and the rule that selects the latest version is used. Filtered navigation from v_2 of *a* towards teasers will thus return v_3 of *t* and v_3 of *u*.

CVCs offer great flexibility for dealing with configurations of versions but slow down relationship traversal. For this purpose, the developer can choose whether a relationship end in an information model supports CVCs. Such a relationship end is called *floating*. A relationship with a non-floating end can be traversed using only unfiltered navigation in the direction of this end.

2.3 Workspaces

A *workspace* (also called a *configuration*) is a special case of object that attaches versions of other objects. Attaching versions to a workspace usually reflects their semantic compatibility. However, at most one version of a given object can be attached to a workspace at a time. This restriction allows a workspace to act as a version-free view to objects stored in a versioning system. As a user selects a workspace, the pin setting and the specified rule for selecting versions from a CVC are ignored in filtered navigation and the version attached to the selected workspace is returned for every CVC.

Workspaces are also used as scopes for long-lived (design) transactions. The *checkout* operation puts a persistent lock on a version and the *checkin* operation releases this lock. The lock needs to be associated with one of the workspaces the version is attached to. When the lock is set, the version can be modified only when the workspace was selected by the client.

2.4 Operation Propagation

As noted by Rumbaugh [30], relationships are useful for propagating operations between objects. We use this idea in a versioning system to propagate the operations *create*, *delete*, *attach*, *detach*, *copy*, *freeze*, *createSuccessor*, *checkout*, and *checkin* across a relationship. An operation will be propagated whenever its propagation setting is defined on a relationship end in the information model. Some of the propagation settings also have an impact on the creation and deletion of relationships, as described below.

- *Propagations of create, copy, and createSuccessor.* In these propagations, relationships between the origin version and the target version will be created.
- *Propagations of attach and detach.* Attachment relationships between the target version and the workspace will be created.

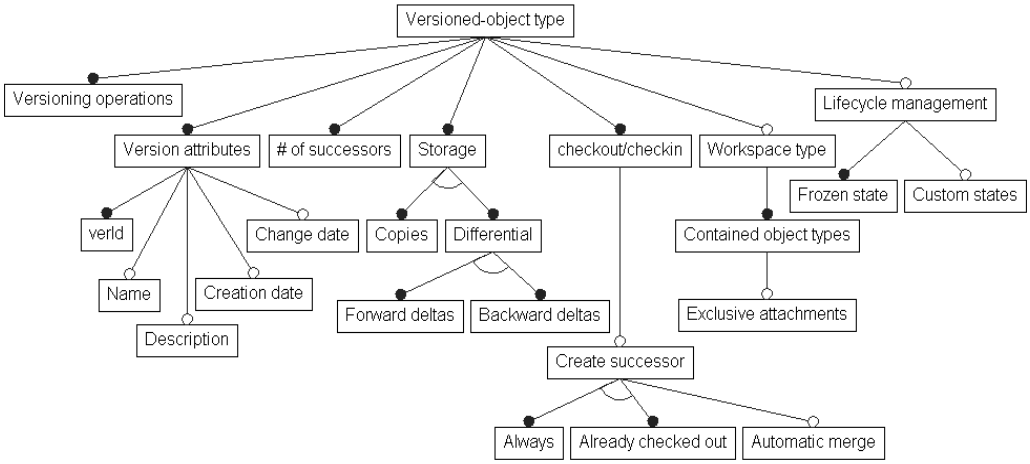

Fig. 3. Feature diagram for a versioned-object type

3 Feature Diagrams

Based on our description of the functionality of versioning systems presented in Sect. 2, this section gives an excerpt of the feature diagrams used to hierarchically organize the features during domain analysis (for an introduction to domain analysis and feature diagrams see [9] and [16]). The diagrams served as a basis for defining model-based configuration languages in the generation approaches presented in Sects. 4 and 5. The most important diagrams describe versioned-object types, versioning operations, and relationship ends and are explained in this section.

The feature diagram for a versioned-object type is illustrated by Fig. 3. Every versioned-object type includes versioning operations (represented in the subdiagram in Fig. 4). A version must include the *verId*, but may include version name, description, creation date, and last change date. To limit the breadth of the version graph, the maximum allowed number of successors to any version is defined. The storage of versions can proceed either in full copies or in differences, using forward or backward deltas. If desired, the *checkout* operation can automatically create a successor for parallel development. *Already checked out* denotes that the successor will be created only if the version has already been checked out at the time *checkout* is invoked. *Automatic merge* denotes that the branches will be reunited upon checkin. A workspace type is a special kind of a versioned-object type. We may want to disallow the simultaneous attachment of a version to multiple workspaces (*Exclusive attachment*). In its lifecycle, an object moves through different states. The frozen state is always available, but there may be other custom states that are defined by the user.

Versioning operations, illustrated in Fig. 4, include the general version graph traversal operations, as well as version manipulation operations *createSuccessor*, *freeze*, *deleteVersion*,

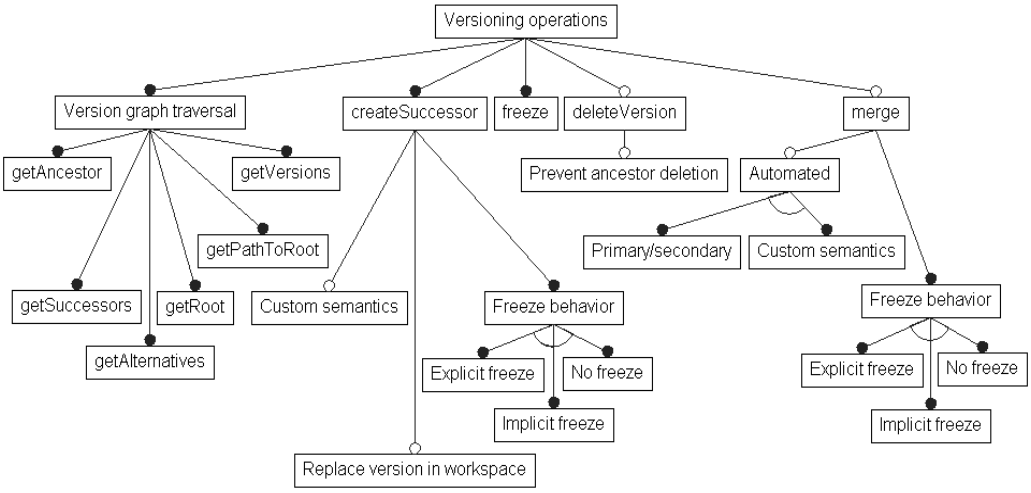

Fig. 4. Subdiagram for versioning operations

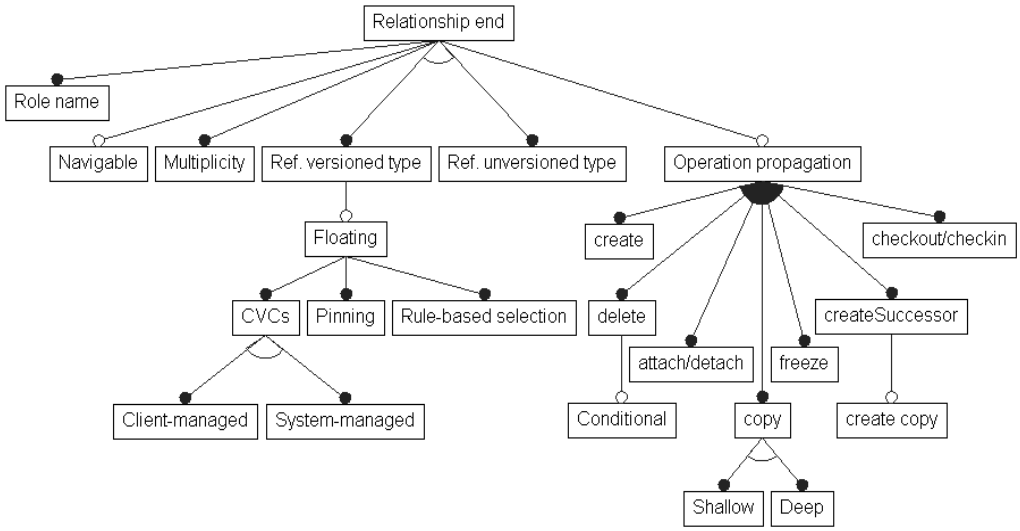

Fig. 5. Feature diagram for relationship end

and *merge*. By default, the values of user-defined attributes are copied from the ancestor to the successor, unless custom semantics for *createSuccessor* is defined. Optionally, when invoked within a selected workspace, *createSuccessor* can replace a version in the workspace. On *createSuccessor*, the system can request that the ancestor is to be frozen by the user (*Explicit freeze*), freeze the ancestor itself (*Implicit freeze*), or require no freeze action. Optionally, invoking *deleteVersion* can be prevented for versions that already have successors (*Prevent ancestor deletion*). The *merge* operation can be automated using a simple *Primary/secondary algorithm*. This algorithm is described by [22] and requires the user to mark one of the versions as primary and the other as secondary to decide on the priorities of their attribute values. A custom merge semantics can also be defined.

Most of the interesting semantics is implied by the settings associated with relationship ends (see the feature diagram in Fig. 5). Every relationship end has a role name. An end may

be navigable, i.e., it is possible to traverse the relationship in the direction of the end. An end has a multiplicity value that can be either *one* or *many*. It can reference either a versioned-object type or an unversioned-object type. In a client-managed CVC the client application chooses the candidate versions to be added to the CVC. In a system-managed CVC the application chooses only the first version to add. Each subsequent version is automatically added to the CVC. Finally, operation propagation is defined in terms of settings for a relationship end.

4 A Generator for Middleware-Based Versioning Systems

In this section we describe our experience with developing a generator for middleware-based versioning systems. First, to give an insight of what needs to be generated, Sect. 4.1 describes the architecture of such systems. We use the UML profile described in Sect. 4.2 to configure a versioning system. The implementation of the generator is discussed in Sect. 4.3. Apart from the productivity benefits achieved through the generation process, we claim that there is a range of performance features that are difficult to support in a generic implementation, i.e., by a framework for versioning systems. We implemented such a framework and compared the performance of a generated versioning system towards a generic system. Our results are presented in Sect. 4.5. Finally, the lessons we learned from this approach are summarized by Sect. 4.6.

4.1 Middleware-Based Versioning Systems

Every middleware-based versioning system that we generate consists of a persistence layer, an entity layer, an access layer, and a services layer, as described in the following paragraphs.

Persistence Layer. This layer consists of relational tables which store versioned data. Every object type and workspace type defined in the information model is mapped to a separate table, called *object-type table*. An object-type table consists of a *globId*, an *objId*, and additional user-defined attributes. In case the object type is versioned, the versioning information is integrated directly in the object-type table. Depending on whether a relationship end is floating and the multiplicity of the end, the generator tries to append the information about the relationship participation directly in the object-type table to minimize the number of joins required for traversals. In case this is not possible, a separate *relationship-type table* is created.

Entity Layer. This layer gives an object-oriented view to data stored in the persistence layer. Every object type is represented by a separate entity component that requires the generation of a component's class, local home and component interfaces, remote home and component interfaces, a value-object class, and a set of definitions for the deployment descriptor. The value-

object class is a serializable representation of an object type used to exchange object data with a remote client. The definitions for the deployment descriptor describe system-defined (*globId*, *objId*, etc.) and user-defined attributes of the type as well as its relations to other entity components.

Access Layer. This layer is made of stateful session components and is the first layer visible to the client. As in the entity layer, a separate session component for every object type exists in the session layer. The layer acts as a facade for diverse attribute updates, unfiltered or filtered relationship traversal, and operation propagation. The methods offered by the access layer are both *integer-based* and *value-based*. Integer-based methods return a single integer value or a set of values to the client, depending on the expected cardinality of the result set. Value-based methods return a single value object or a set of value objects to the client.

Services Layer. The services layer consists of proxy classes that allow the invocation of methods in the access layer using SOAP messages.

Content Browser. Instead of developing clients that use the access and services layer to modify data in the system, a user can explore the data interactively and manually invoke versioning operations. This is supported by a special Web application called *content browser*.

Test Client. Finally, we also generate a test client for the versioning system, which uses the services layer to systematically invoke a predefined set of operations for every object type and compare the returned values with the expected values. This client was important to test the code produced by the generator during its development. However, even in the productive use of the generator, it can serve as an example for developing clients.

4.2 Configuration

To support model-based configuration of versioning systems, a special UML profile, illustrated by Fig. 6, has been developed. The profile supports a subset of features described in Sect. 3. The configuration is obtained by representing the information model as a class diagram and afterwards applying the elements of the profile. A sample configuration that we will use in the evaluations presented in this section is illustrated by Fig. 7.

4.3 Generator

We adopted a template-based generation approach in this project. Nearly all such approaches work in a very similar way. First, we develop a set of templates that consist of *static parts*, *placeholders* for user-defined values, and *control flow statements* that guide the evaluation of the template. The placeholders and the control flow statements are also referred to as *meta-*

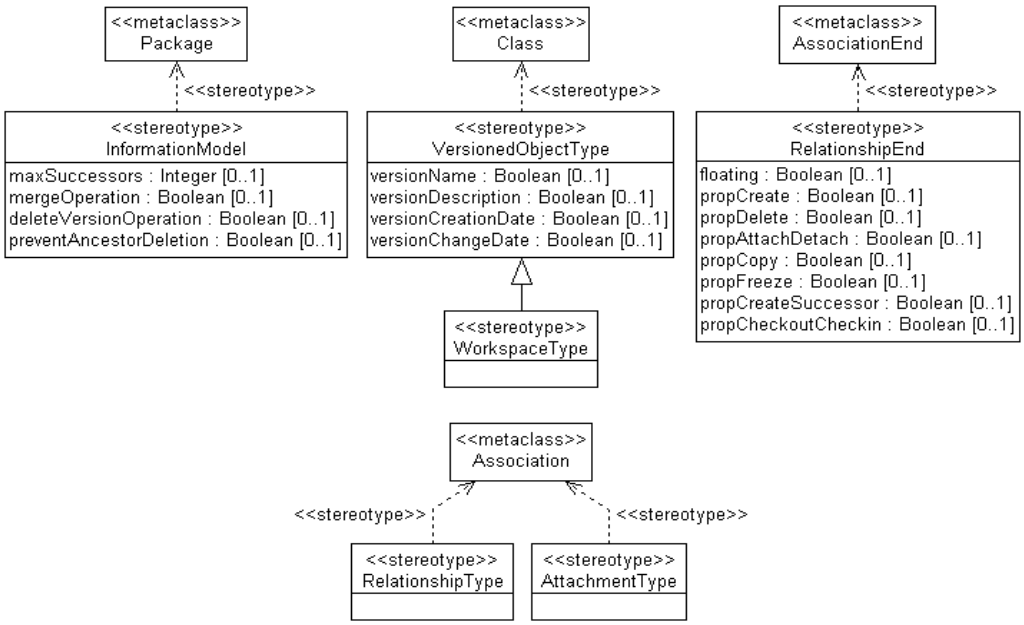

Fig. 6. UML profile used for configuration

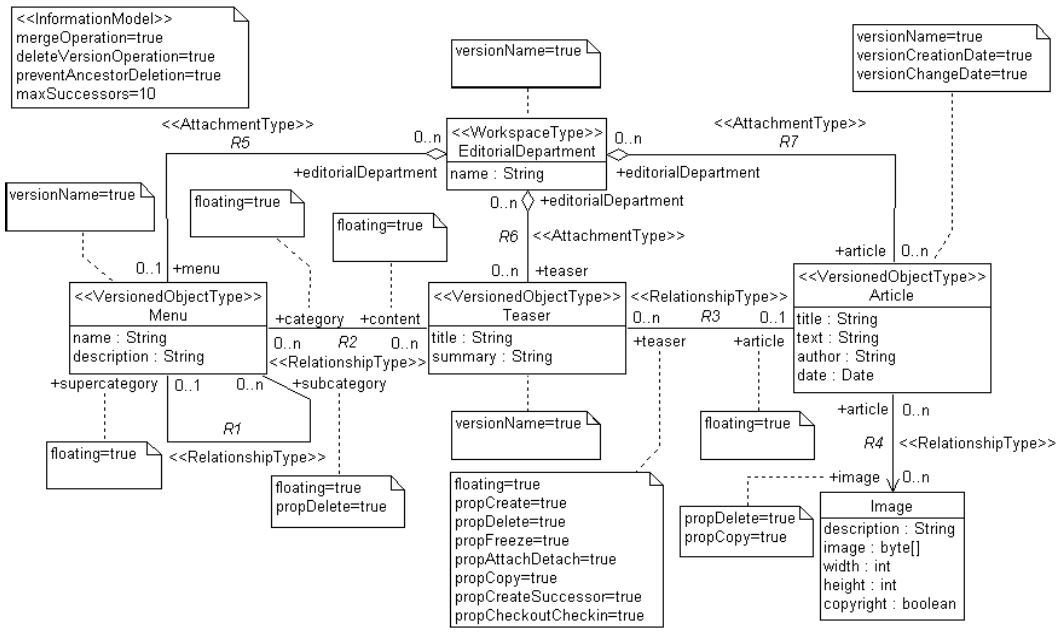

Fig. 7. An example configuration of a middleware-based versioning system

code, to separate them from the actual code segments in the template's static parts. Afterwards, we develop a generation program that prepares a context and merges the context with a template to obtain the target file. A context is a set of values, implemented as a hash table, that will replace the placeholders and be used in evaluating control flow expressions. The 25 templates used by our generator are listed by Table 1.

Before merging the templates with the context, the generator first reads the UML model into an in-memory UML repository and creates a directory structure for the generated files. Merging is guided by a set of generation targets. Every target tells the generator for which

Number	Template name	Used to generate...
1	*VSObjBean*	a class implementation for components in the entity layer.
2	*VSObjLocal*	local component and home interfaces for components in the entity layer.
3	*VSObjLocalHome*	
4	*VSObjRemote*	remote component and home interfaces for components in the entity layer.
5	*VSObjRemoteHome*	
6	*VSObjValue*	value-object class.
7	*VSObjAccBean*	a class implementation for components in the access layer.
8	*VSObjAccLocal*	local component and home interfaces for components in the access layer.
9	*VSObjAccLocalHome*	
10	*VSObjAccRemote*	remote component and home interfaces for components in the access layer.
11	*VSObjAccRemoteHome*	
12	*ControllerServlet*	a controller servlet for the content browser.
13	*ShowInstance*	a JSP page that displays information on an 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 object as a result of a finder method or a navigation operation.
16	*ShowIndex*	an HTML page that displays the navigation bar in the content browser.
17	*Web-xml*	a deployment descriptor for Web components.
18	*EJB-jar-xml*	a deployment descriptor for business components.
19	*Application-xml*	a deployment descriptor for the entire multi-tiered application.
20	*AS-Application-xml*	an application server specific deployment descriptor.
21	*Deploy-wsdd*	a Web services deployment descriptor.
22	*Undeploy-wsdd*	a file to undeploy previously deployed Web services.
23	*VSObjService*	proxy classes for the services layer.
24	*TestClient*	a test client.
25	*Build-xml*	a build script to compile and deploy the entire application.

Table 1. Templates used in the generation process

Prepared elements	Aggregated elements from the UML Metamodel
PreparedAssociationData	Association, Stereotype, TagDefinition, TaggedValue
PreparedAssociationEndData	AssociationEnd, Multiplicity, MultiplicityRange, Association, Stereotype, TagDefinition, TaggedValue
PreparedAttributeData	Attribute, Classifier, Namespace, Stereotype, TagDefinition, TaggedValue
PreparedClassData	Class, Generalization, Stereotype, TagDefinition, TaggedValue
PreparedMethodData	Method, Operation, Parameter, Classifier, Namespace, Stereotype, TagDefinition, TaggedValue
PreparedModelElementData	ModelElement, Stereotype, TagDefiniton, TaggedValue
PreparedPackageData	Package, Stereotype, TagDefinition, TaggedValue

Table 2. Prepared elements

model elements, e.g., every class, a given template is to be applied and how to name the output file. Information required by the template's metacode is often dispersed across many model elements. Sturm et al. [35] propose a solution to this problem by aggregating information from many model elements into the so-called prepared elements. This solution was also adopted in our implementation. Table 2 shows the prepared elements that we use and the corresponding aggregated elements from the UML Metamodel.

4.4 Properties of the Templates

We investigated the 25 templates from Table 1 using the following measures: file size, *LOC*, number of references to values in the context, if-statements, loops, statement count, cyclomatic complexity [20], and Halstead effort [13]. Most tools that support the automatic application of the above measures can only deal with statements in general-purpose programming languages. For this reason, we used the following guidelines to transform every template to comply with this requirement.

- Every context value reference was transformed to an atomic *print* statement.
- Every static code output was transformed to an atomic print statement. A context value reference delimits static code outputs.
- Every *#if/#else* metacode statement is treated as an if/else statement and every *#foreach* metacode statement is treated as a while statement in a general-purpose language.

Altogether, 288 KB of template files contain 3,013 references to prepared classes, 542 if-statements, 128 loops, and 6,670 statements. A comparison of cyclomatic complexity values for the templates is given by Fig. 8. Major differences in the complexity between the templates can be explained with a diverse amount (and complexity) of metacode they contain. The template *VSObjAccBean*, which generates the class implementation of a component in the access layer reaches an especially high complexity value (237). This template also reaches the highest Halstead effort value (approx. 22.2 million). Comparing cyclomatic complexity to the LOC values of the templates shows a linear relationship (Fig. 9). A similar trend can be observed for the Halstead effort if a logarithmic scale is used for both axes. This suggests that even though the templates generate files used for very different purposes (interfaces, class implementations, JSPs, or deployment descriptors) complexity and effort are evenly distributed across the lines

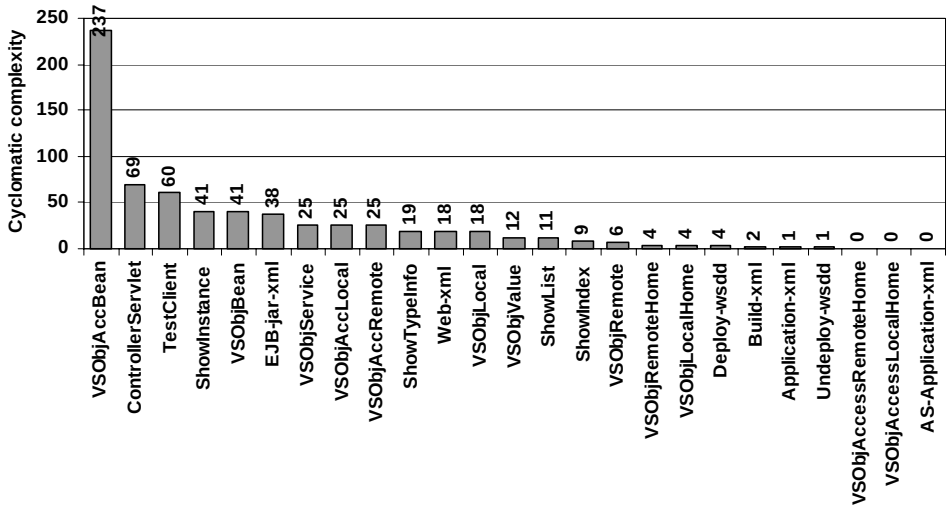

Fig. 8. A comparison of cyclomatic complexity values for the templates

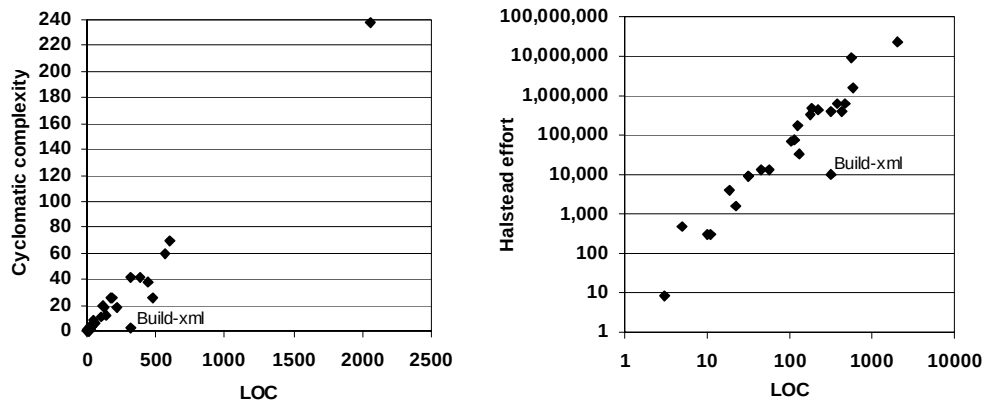

Fig. 9. Correlation of cyclomatic complexity and Halstead effort to LOC

of code. The specially marked outliner denotes the template *Build-xml* that generates the build script to compile and deploy the generated system. In comparison to its static parts, this template contains only a small portion of metacode.

For the example configuration illustrated in Fig. 7, 6,423 LOC of templates generate 17,514 LOC of source files. This proves that even for a very small information model a reuse factor of 2.73 in terms of LOC is achieved by the approach.

4.5 How Fast are Generated Versioning Systems?

Apart from reducing development time and eliminating possible errors due to manual development, is there a performance benefit in generating versioning systems? To observe this, we implemented a *framework for versioning systems*. We tried to shift most of the versioning functionality into the framework's generic parts and implement it so that it could be used with any information model. Differences between a framework-based system and a generated system are summarized by Table 3. Fig. 10 illustrates framework instantiation for the example configuration from Fig. 7.

As a result of using many generic parts, a versioning system obtained by framework instantiation uses a database schema that is far simpler than the schema of a generated system, but at the same time also shows some performance drawbacks. For example, the possibility of reducing the number of joins between the object-type tables for the case when the multiplicity of a relationship end is *one* is not considered. In a framework-based system, generic tables, e.g., tables for storing versioning information, attachments, or pin settings, are used for objects and relationships of every possible type (in contrast to the generated system, where this information is stored in type-specific tables). Such generic tables grow very fast and thus require high table-scan times. Finally, framework instantiation through hot spots causes an increased use of

Concept / feature	Generator	Framework
Unversioned-object types	Supported.	Not supported—every access component for a type extends a generic component *VSObjectAccess* that defines the methods for versioning behavior.
Versioning information (ancestor to a given version, information on whether a version is frozen)	Integrated directly in the entity component for the versioned-object type and the object-type table. The ancestor-successor relationships are supported as a reflexive one-to-many relation on the entity component.	Stored by a separate entity component called *Version*. The ancestor-successor relationships are supported by a reflexive one-to-many relation on this component.
Merging information (what versions have been merged with other versions)	Supported by a reflexive many-to-many relation on the entity component for the object type.	Supported by a reflexive many-to-many relation on the entity component *Version*.
Relationship types	Every relationship type is represented as a separate relation between entity components.	Supported by a common entity component called *Relationship*. The component is used for relationships of any possible relationship type.
Non-floating relationship ends	Supported.	Not supported—due to the representation of relationships using the component *Relationship*, all ends are floating.
Pin settings	A separate relation between the two entity components is used with every floating end.	Supported by a common entity component called *PinSetting*. The component is used for relationships of any possible relationship type.
Latest version selection	Materialized and represented as a relation between the two entities for every floating end.	Not materialized. Versions in a CVC need to be scanned to determine the latest version.
Attachment relationships and checkout locks	Two separate relations between entity components that represent the workspace type and the object type are used with every attachment-relationship type.	Supported by two entity components called *Attachment* and *Checkout*. The components are used for attachment relationships of any possible type.
Operation propagation settings	Hardwired directly in the access components.	Stored in a special database table that is queried as an operation that might propagate is carried out.
Maximum number of successors	Hardwired directly in the access components (methods *createSuccessor* and *merge*).	Stored in a special database table that is queried as methods *createSuccessor* and *merge* are carried out.

Table 3. Differences between generated and framework-based versioning system

subtype polymorphism and reflection which slow down the invocation of methods in the entity and access layers. To observe performance differences between a framework-based and a generated versioning system, we developed a benchmark client for the example configuration from Fig. 7. The client carried out 192,148 operations in three consecutive runs of a typical content management scenario. For every operation category, t_{fw} and t_{gen} denote the average time required to execute the operation in this category for the framework-based and the generated system, respectively. The speed-up of a generated system is defined as $(t_{fw}\text{-}t_{gen})/t_{fw}$. Fig. 11 gives an overview of speed-ups for observed operation categories. This overview confirms our hypothesis that the optimization decisions used in generated systems indeed have an impact on performance. At the same time, the separate representation of versioning information by the entity *Version* causes the operations for navigating version histories (*getAlternatives*, *getRoot*, *getAncestor*, and *getSuccessors*) as well as the operation *deleteVersion* to perform better in a framework-based system.

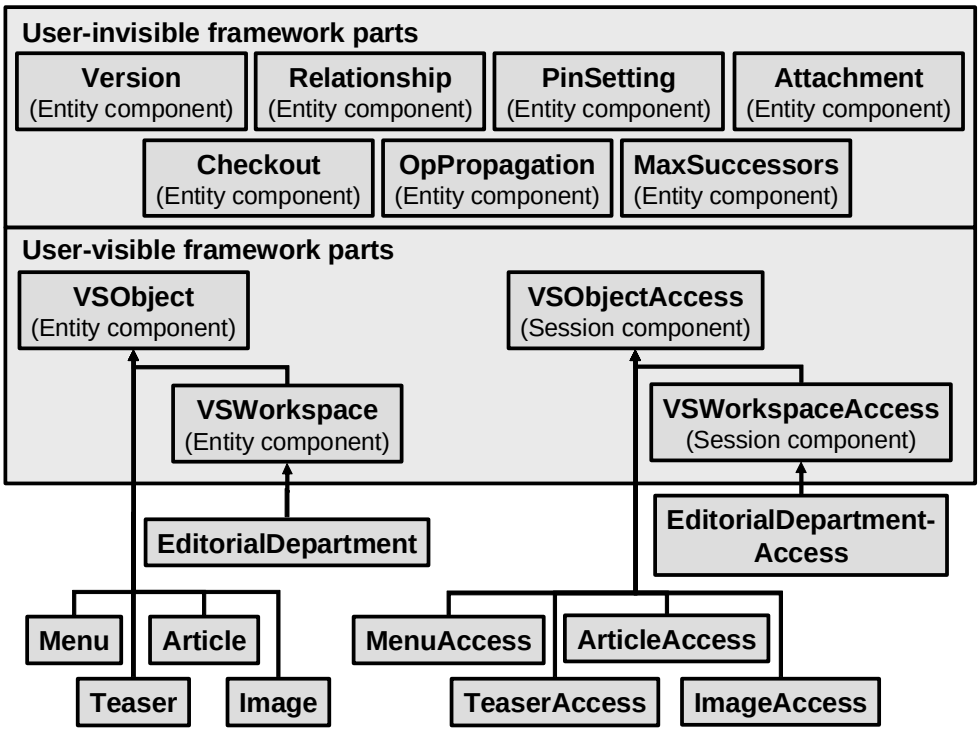

Fig. 10. Framework instantiation for the example configuration

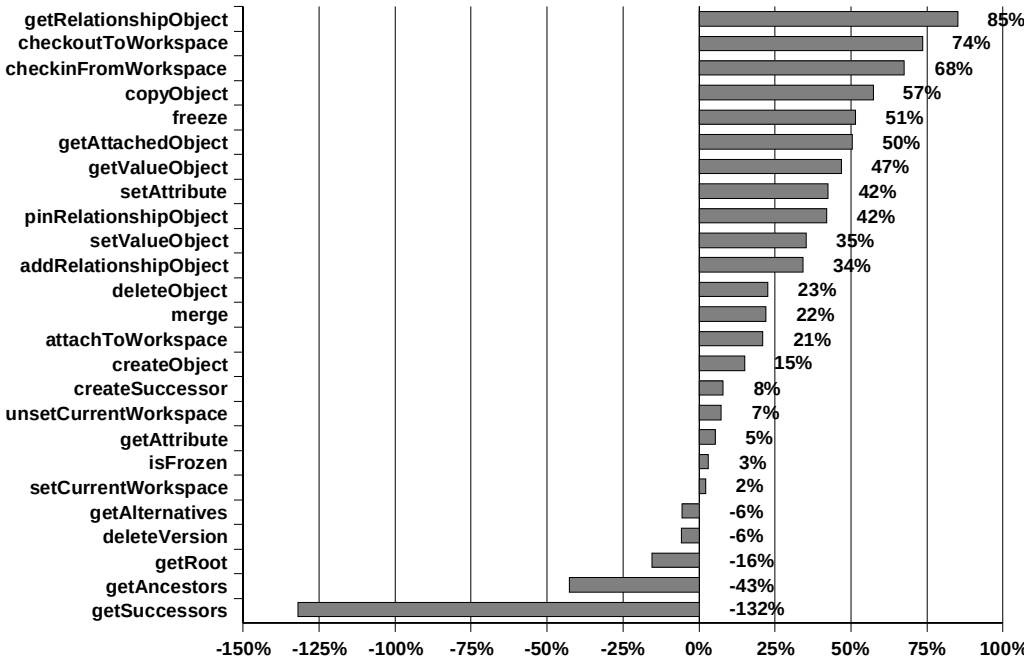

Fig. 11. Speed-up of a generated towards a framework-based system

4.6 Lessons Learned

The entire development activities for the generator took 12 man-months, the development of the templates taking approx. 75% of this time. The development of the framework began after the majority of template development was completed and took only six man-months, mostly

MCC	**Risk evaluation**
1–10	A simple method without much risk
11–20	More complex method, moderate risk
21–50	Complex, high risk method
Greater than 50	Untestable method (very high risk)

Table 4. Risk evaluation scale (based on van Doren [37])

due to our experience with the domain and the simplifications explained in Table 3. What are the most important lessons we learned from this project?

Return-on-Investment (ROI). Our measurements fail to answer the most important question: How many systems (with what features) do we need to sell (at what price) in order for the implementation of the templates and the generator to pay off? This question cannot be answered until a measure for estimating the total cost for template development is proposed and verified on a large set of product lines. The measure should reasonably combine the efforts related to developing the template's static parts and the metacode. In terms of LOC, our templates generate 2.73 times more code that their size even for a very small example configuration from Fig. 7. This result is to be treated with care due to its uniform treatment of a line of code in a template and a line of code in the generated implementation.

High Complexity and Effort Values for Some Templates. Templates with a large portion of application logic prove more difficult to develop and test. To classify cyclomatic complexity values, van Doren [37] proposes a risk evaluation scale, illustrated by Table 4. According to this classification, three of our templates are classified as untestable with very high risk, six templates are classified as complex with high risk. The other 16 templates have acceptable complexity values (20 or less). A possible remedy for a template with high complexity is to divide it into smaller parts which are then included in a superordinated template. The template language we use (VTL [3]), supports this kind of inclusion by *#include* and *#parse* directives. However, in our case, this action has the mere outcome of having many parts with lower complexities. We could not find the case where we could reuse one part across two or more templates.

Example-Based Template Development. We never developed a template by immediately implementing static code mixed with metacode. Instead, a series of examples for the generated code that covered diverse configurations was developed. Afterwards, we generalized these examples to obtain the template. In our opinion, this generalization can be automated by tools that compare the examples and automatically derive the template.

Relating Domain Analysis to Domain Implementation. A careful analysis of features for the product line (see Sect. 3) is a labor-intensive process. However, the results of this analysis are mostly neglected during domain implementation due to insufficient tool support. Our experience shows that features are often added to domain implementation gradually as the domain evolves. In this case, domain analysis and implementation need to be enhanced in many small iterations. To allow the tracking of changes and dependencies across the phases, tools should be available to relate parts of template implementation to corresponding features.

Is performance the only reason for implementing a generator? No—our experience shows that manual framework instantiation is still a labor-intensive task. For the example configuration, 45 derived classes require 3,822 LOC and a Halstead effort of 12,490. The developers need to get familiar with the way the configuration-specific parts interact with the generic parts (mostly through the components *VSObject* and *VSObjectAccess*, see Fig. 10). Naming conventions used to invoke methods by reflection are a common cause of errors. For this reason, we claim that even for the framework solution, a simple generator is necessary.

5 Domain-Specific Languages for SQL

In this project, we took a different approach than in generating middleware-based versioning systems. First, we changed the programming model for the systems, i.e., the systems generated in this section are configured and accessed through a *domain-specific language* (*DSL*) that is mapped to SQL. Statements in this language can be embedded into versioning system applications. The mapping to SQL is performed either by a domain-specific database driver or a precompiler. Second, the mapping is performed by object-oriented code in special translation routines which differ substantially from the templates presented in Sect. 4. Instead of implementing only the domain-specific language for versioning systems, we implemented a transformation system that allows easy definition of data definition and manipulation languages. From this definition, a metagenerator produces a modeling environment, a precompiler, and a domain-specific driver. This idea can thus be thought of as an extension of an increasingly popular field of *method engineering*. First, in Sect. 5.1, we give a brief introduction to method engineering. Sect. 5.2 outlines DSLs. Several examples for using DSLs for database applications are given in Sect. 5.3. Our system is described in detail in Sect. 5.4. Sect. 5.5 describes concrete use of domain-specific statements for versioning. The evaluation setup and results for this approach are presented by Sects. 5.6 and 5.7. Finally, in Sect. 5.8, we summarize the lessons we learned from this approach.

5.1 Method Engineering

According to Rumbaugh [31], a *method* is a combination of guidelines and rules that include *modeling concepts, views and notations, a step-by-step iterative development process*, as well as *hints and rules-of-thumb for performing development.* A method is also often seen as a combination of a *product model* (definition of modeling concepts) and a *process model* that defines the development activities. Popular methods include *Rational Unified Process* (*RUP*) [14], *Dynamic System Development Method* (*DSDM*) [34], and *Catalysis* [11].

Brinkkemper [7] defines *method engineering* as a discipline for exploring techniques to build project-specific methods. The term *computer-aided method engineering* (*CAME*) [32] emphasizes the use of software tools for building methods. CAME tools generally produce a CASE tool that supports a project-specific method. The CAME tool introduced in this section falls into the category of *ad hoc method engineering* [29] due to the fact that no reusable method pieces are available to the user as he starts building the method and the process of building is not limited in any way (except by a meta-metamodel).

5.2 Domain-Specific Languages

DSLs are specialized problem-oriented languages. Because they contain abstractions meaningful only within an observed domain, programs in DSL are easier to understand and communicate. According to Czarnecki and Eisenecker [9], DSLs can be implemented using *preprocessors, reflective languages,* or special *modularly extensible programming environments* (see the *Intentional Programming* system [33] as an example for such environment). A DSL is usually executed either by implementing a virtual machine or by translating programs to a lower-level language in a process called *synthesis* [38].

5.3 DSLs for Database Applications

Why use DSLs for database applications? Common database schemas in commercial applications span more than 10,000 predefined tables and contain 100+ lines of code for a single query. This kind of application design is difficult to communicate and very error-prone. Triggers, constraints, or denormalization decisions in database schemas are usually left undocumented and therefore difficult to understand whenever reverse engineering, reengineering, or information integration is necessary. Finally, by combining triggers, constraints, and database stored procedures with the host language code, numerous ways to achieve the same effect in the database are available. By using a DSL, execution decisions are postponed to translation time, leaving the application code unmodified. In this way, an additional level of independence

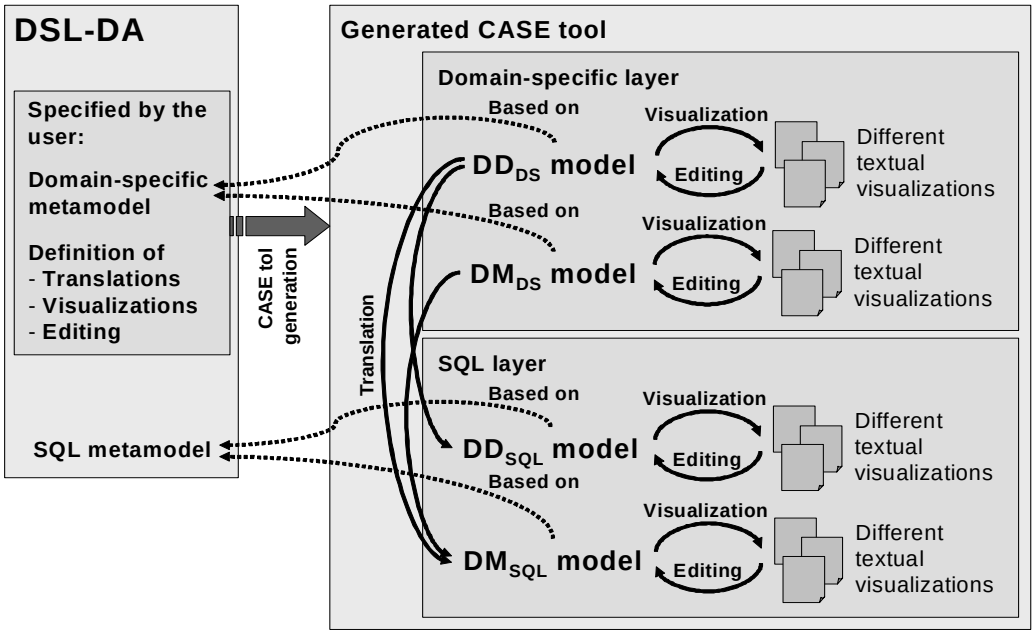

Fig. 12. Overview of the DSL-DA system

is achieved. Examples of existing DSLs to be used with database applications include *Preference SQL* [18] (a language for easy expression of customer preferences in e-commerce applications), *OrientSQL* [40] (a language for enhanced treatment of relationship types in database schemas), and *Visual SQL* [15] (a graphical query language based on the ER modeling paradigm).

5.4 A System to Define Custom DSLs

In this project we developed a system called DSL-DA (domain-specific languages for database applications) which allows the definition of a custom DSL that can be translated to SQL. An overview of the system is given by Fig. 12.

The definition of a DSL consists of two large parts, (i) the definition of a domain-specific metamodel and (ii) the definition of translations, visualizations, and editing functionality. Based on this two parts, DSL-DA generates a CASE tool as well as a domain-specific database driver and a precompiler for the DSL. The CASE tool consists of two layers, the domain-specific layer and the SQL layer. On the domain-specific layer, the user assembles data definition (DD_{DS}) and data manipulation (DM_{DS}) models using domain-specific modeling elements. The DD_{DS} model is used for schema definition while the DM_{DS} model is used for updating and querying data. These models can be visualized using diverse textual notations that allow us to work with different views on the same model. Textual notations can be edited, which results in the corresponding updates of both models. Translations are responsible for translating the

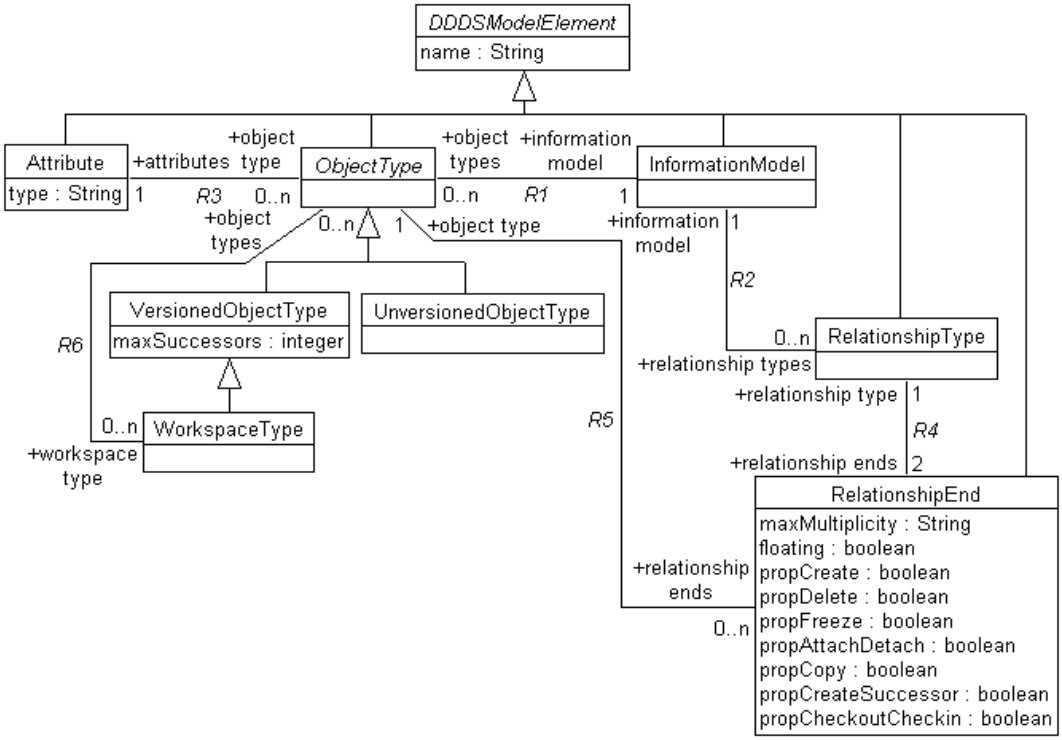

Fig. 13. Part of the domain-specific metamodel used for versioning systems

models onto the DD_{SQL} model (responsible for data definition in terms of SQL modeling elements) and the DM_{SQL} model (responsible for data manipulation). The metamodel for DD_{SQL} and DM_{SQL} models is the SQL metamodel, which is an extension of CWM [24]. Because CWM provides only the modeling elements for data definition, an extension is necessary to represent data manipulations.

The meta-metamodel for domain-specific metamodels defines a set of very general elements for object-oriented metamodeling. Fig. 13 illustrates the data-definition part of the domain-specific metamodel for versioning systems. Fig. 14 illustrates an example DD_{DS} model that is based on the domain-specific metamodel. The user interface of the generated CASE tool for manipulating the model (segment *A*) and its possible visualizations (segment *B*) is illustrated in Fig. 15.

5.5 Domain-Specific Statements

DD_{DS} and DM_{DS} models are good for an abstract representation of the schema of a versioning system and various data manipulations, but cannot be directly embedded into the host language code. For this purpose, developers use the surface syntax of the DSL (which corresponds to a possible visualization of the models) to embed the statements in the code. Four examples of

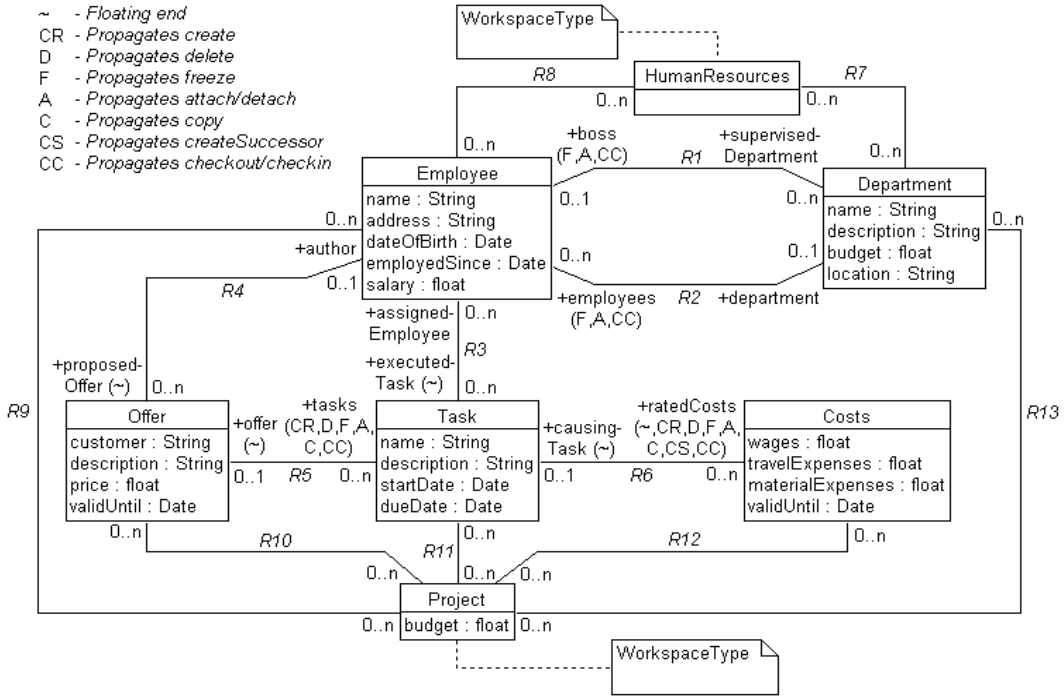

Fig. 14. Example DD_{DS} model

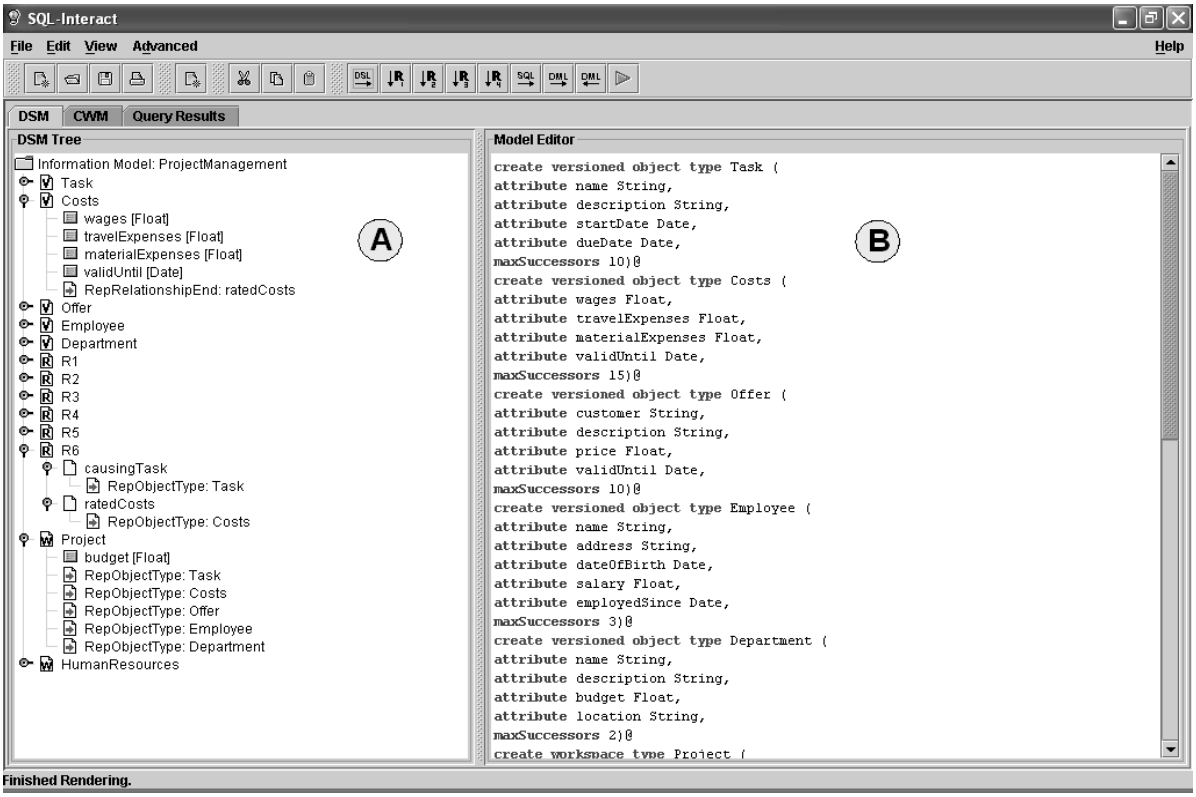

Fig. 15. Domain-specific layer of the generated CASE tool

such statements along with their explanations are given by Table 5 (the statements use the DD_{DS} model from Fig. 14).

We differentiate between the following two kinds of statements that appear in the host language code.

Statement	Explanation
create new object Offer (customer, description, validUntil) values ('DBIS Group', 'Development of DSLs', '2004-03-01')	Creates a new offer with the specified attribute values. Due to operation propagation settings, a new task will be created and connected to the offer.
create successor of object Task where objId = 875 use workspace Project where globId = 934521	Creates a successor to the task selected using the specified project workspace. The operation will propagate to the connected costs.
get successors of Task where objId = 875 use workspace Project where globId = 934521	Returns successors of the task version selected using the specified project workspace.
select Costs.materialExpenses from Offer--tasks-->Task--ratedCosts-->Costs where Task.startDate > '2004-01-01' use workspace Project where globId = 846833 and Offer where objId = 295	Selects the material expenses of all costs that can be reached from an offer with the specified *objId* using filtered navigation across the relationships of type R5 and R6. Only the tasks started later than Jan. 1, 2004 are considered. The navigation is performed within a project workspace with a *globId* 846833.

Table 5. Example DM_{DS} statements

- *Built-in statements.* These statements are known in advanced as the programmer develops the application code.
- *Dynamic statements.* These statements are assembled at run time depending on the user's actions.

A built-in statement can be handled both by a precompiler and a domain-specific driver (installed before the native database driver) while a dynamic statement always needs to be handled by a domain-specific driver. The driver solution also allows us to modify the translation definitions for an already compiled code (by changing only the driver). A drawback of the driver solution is the excess time spent in the translation process at run time. For our evaluations, presented in Sects. 5.6 and 5.7, we assumed the worst-case scenario, where all statements in our benchmark were executed using a domain-specific driver.

5.6 Evaluation Setup

The goal of our evaluations was two-fold. First, we wanted to estimate the effort needed for developing the translation routines for the DSL used for versioning systems. This estimate was performed by counting the statements in the routines as well as by determining the cyclomatic complexity and Halstead effort. Second, we were interested in performance characteristics of domain-specific statements.

The time required to execute a domain-specific statement t_{ds} is defined as a summary

$$t_{ds} = t_{par} + t_{tr} + t_{vis} + t_{SQL},$$

where t_{par} is the time used for parsing the statement to build a DM_{DS} model, t_{tr} the time needed for translating this model to a DM_{SQL} model, t_{vis} the time needed for the visualization,

and t_{SQL} the time needed to execute the domain-specific statements. Thus the overhead due to the use of domain-specific statements (the time spent in the domain-specific database driver) is the sum

$$t_{dr} = t_{par} + t_{tr} + t_{vis}.$$

To observe t_{dr} relative to t_{ds}, we define the overhead ratio o_{ds} as

$$o_{ds} = \frac{t_{dr}}{t_{ds}}.$$

The aim of the evaluations is to validate the following two hypotheses.

- *Hypothesis 1.* In general, different translations are possible for a domain-specific language. Translations that take certain optimizations into account are more complex to develop and also imply higher t_{tr} values.
- *Hypothesis 2.* o_{ds} is very small even for complex translations, indicating that the costs of translation at run time are acceptable.

In order to validate the two hypotheses, we need a sufficient set of different translation variants for our domain-specific language. The following four variants, which differ in their treatment of relationships, were used in our evaluations.

- *Variant 1.* Store all relationships, regardless of relationship type, in a single generic table.
- *Variant 2.* Use a separate table for every relationship type.
- *Variant 3.* Improve *Variant 2* by a special treatment of maximal multiplicity *one* on non-floating ends.
- *Variant 4.* Improve *Variant 3* by a special treatment of maximal multiplicity *one* on floating ends.

5.7 Evaluation Results

The implementation characteristics for the translation routines used for the four variants are illustrated by Fig. 16. Five of six measures (exception is the number of loops) confirm an increasing effort across the four variants.

Our next observation was that the translation code is distributed across the routines in a modular way, allowing easy maintainability and modification. Out of 32 translation routines implemented in *Variant 1*, only seven required an update to obtain *Variant 2*. Again, only seven routines from *Variant 2* required an update to obtain *Variant 3*. Finally, *Variant 4* was obtain by modifying seven routines and adding two routines to *Variant 3*.

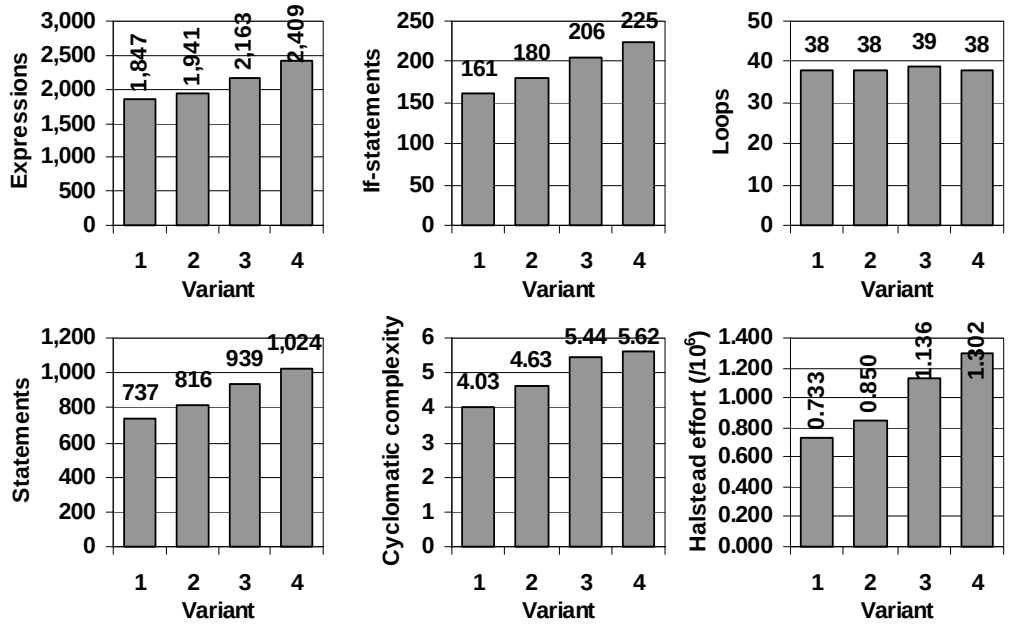

Fig. 16. Properties of translation routines

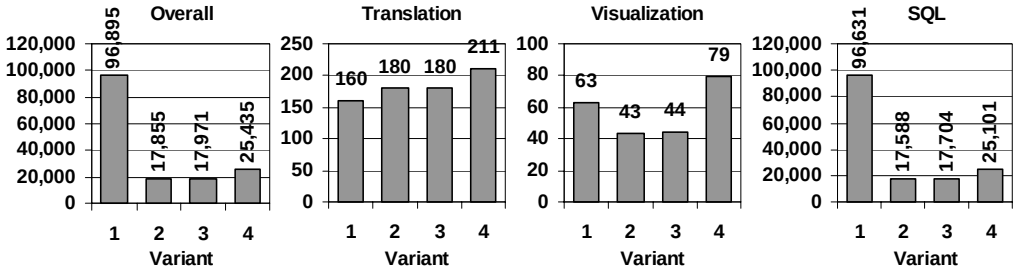

Fig. 17. Execution of *create relationship* statements

In the rest of this section we focus on performance of the four variants. The evaluation of performance was supported by a benchmark consisting of 115,775 domain-specific data manipulation statements. For every variant, the same benchmark was executed against the database schema obtained in the translation, logging t_{par}, t_{tr}, t_{vis}, and t_{SQL}. Every variant required a separate domain-specific database driver. Our first observation was that the parsing time t_{par} is minimal (between 15 and 127 microseconds). Since domain-specific statements do not change, t_{par} remains the same across all four variants. Fig. 17 illustrates the performance of the four variants for the category of *create relationship* statements. The first observation is that t_{tr} and t_{vis} are very small in absolute terms and take up only a small portion of t_{ds}. The values for t_{tr} gradually increase due to an increasingly complex treatment of relationships. A separate storage of information on the pinned and latest version comes into effect in *Variant 4*, thus *Variant 2* proves most efficient both in terms in t_{SQL} and t_{ds}. Fig. 18 illustrates the performance

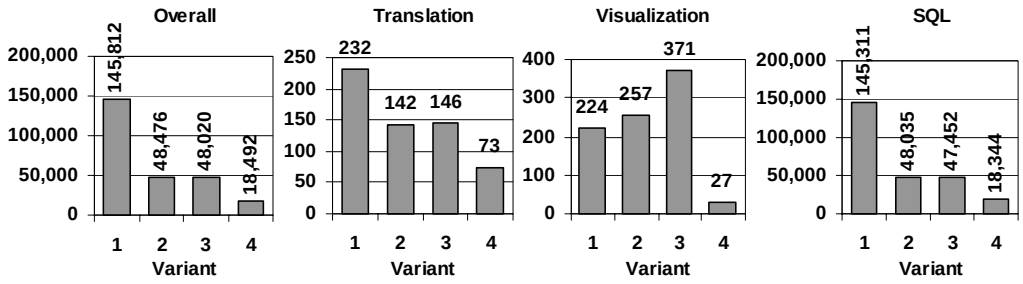

Fig. 18. Execution of *select* statements with four navigation steps outside workspaces

of the four variants for the category of *select* statements with four navigation steps carried out outside workspaces, respectively. In this category, t_{tr} decreases with an increased variant complexity. This implies that *Hypothesis 1* cannot be completely accepted: It is possible for translations that are more difficult to develop to also execute faster. Despite a large amount of decision-making code, once the driver discovers the correct translation for the relationship type, the production of the equivalent SQL statements can occur faster than for a more generic variant. For this category, *Variant 4* proves most efficient both in terms of t_{SQL} and t_{ds}, as expected.

The average overhead ratio o_{ds} over all statement categories takes values between 2.43% (*Variant 1*) and 2.82% (*Variant 3*). The low values confirm *Hypothesis 2*, predicting that the costs of executing the translation at run time are minimal.

5.8 Lessons Learned

What are the most important lessons that we learn from DSL-DA? A situation similar to the one encountered when developing the templates used in generating middleware-based versioning systems occurred. In DSL-DA, we rarely implemented a translation routine immediately. Instead, a number of example translations for statements in a certain category were implemented. By observing these examples and their outcomes (in terms of the SQL statements) we implemented the translation routines. For complex translations, a visualization technique that allows the tracing of the invoked translation routines also proves convenient.

The following ideas could be considered to make translation even faster.

- *Stored procedures.* Using a parameterized stored procedure that can answer many syntactic and DD_{DS} variations of a statement can make the execution of the obtained SQL statements even faster, in case the DBMS precompiles and optimizes the procedure's code. Using a procedure, we also avoid multiple trips through the native database driver.

- *Parallel execution.* In certain cases, some of the SQL statements can be executed in parallel.
- *Object pools.* The creation of objects used for a DM_{DS} model can be made more efficient by using object pools.
- *Integrating visualization into translation.* Translation and visualization need not necessarily be treated as two separate processes. In case explicit DM_{SQL} models are not required, translation routines that assemble SQL statements as character strings are possible.

6 Workflow-Supported Model Transformations

In certain cases, a general-purpose modeling language may be desired by the developers to present solutions in a particular domain. The main reason for this may be previous investments in modeling tools (editors, compilers, metrics tools) that support this language. In this section, we present a solution which allows us to use a general-purpose language but speeds up the development of models by applying a series of domain-specific system configuration steps. The configuration steps are organized in a workflow model. The general idea is described by Sect. 6.1. Sect. 6.2 discusses model transformations which are used by the approach to assemble models. Similar to DSL-DA, the proposed system is generic and can handle different product lines. Sect. 6.3 describes its application for the product line for versioning systems. A metamodel for workflow models is discussed by Sect. 6.4. We evaluate the approach in Sect. 6.5. Finally, a summary of lessons learned is presented in Sect. 6.6.

6.1 Using Workflows to Assemble Large Models

The system implemented in this approach allows the configuration of large models using a set of *transformation templates* and a *workflow model* (see Fig. 19). *Transformation templates*

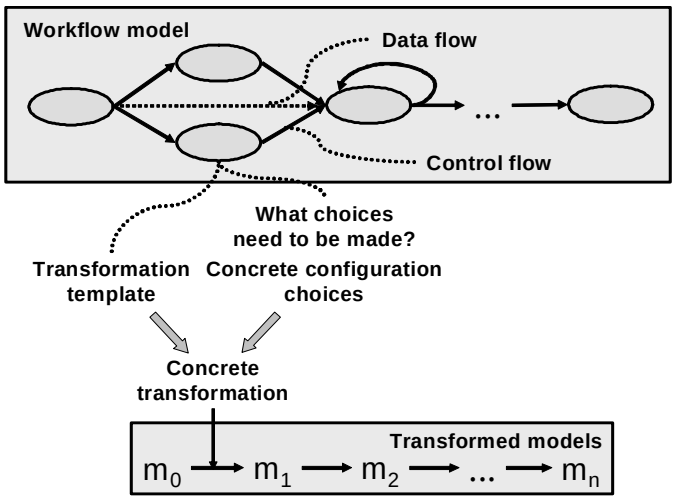

Fig. 19. Workflow-supported model transformations

produce concrete transformations that add elements to the current model, remove elements, or change properties of an existing element. The *workflow model* is an organization of steps in which a user enters his choices that represent the configuration of the final system. Every step is related to a transformation template. At the end of a step, a concrete transformation based on the choices is produced from the template and applied to the model. The workflow model also defines *data* and *control flows* between the steps. *Data flows* carry data to succeeding steps where this data is used for calculating default values or deciding on valid values for the choices. *Control flows* are used to decide whether a transition to a succeeding step is allowed. The main purpose of the system is to eliminate the need for manual modeling in a general-purpose modeling language and to assure the completeness and correctness of the final model.

6.2 Concrete Transformations and Transformation Templates

What is the appropriate notation for concrete transformations and transformation templates? OMG XML Metadata Interchange (XMI) [25] allows a representation of complete models and model differences in an XML-based format. Model differences are supported by the elements *XML.difference, XML.delete, XML.add,* and *XML.replace.* We choose to represent a concrete transformation as a model difference expressed in XMI, which is merged with the current model to obtain the updated model. Transformation templates are represented as a combination of static XMI code, placeholders for user-defined values, and control flow statements. Note that template evaluation is technically equivalent to the one applied in Sect. 4, where we use it for model-to-code transformations. Here, we use it to generate concrete model-to-model transformations.

6.3 Example

The example product line can be supported by four workflow steps (and thus four transformation templates) illustrated in Fig. 20. For the example product line we use Executable UML [21] as the general-purpose language used to express the transformed models. Executable UML requires an interconnected representation of system structure (defined in terms of

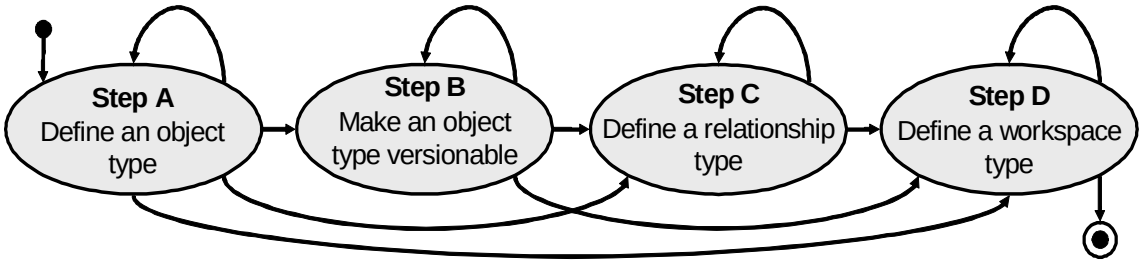

Fig. 20. Configuration steps for the example product line

classes, associations and attributes) and system behavior (defined in terms of operations on classes). The operations are defined in terms of actions, expressed in the *Object Action Language* (*OAL*) [27]. Even though an XMI representation of actions is possible through the modeling elements defined by the package *Actions* of the UML metamodel, this approach is very fine-grained. For example, a simple *if* statement with *elif* requires 32 model elements and 35 links (see [26] for more examples). This proves unacceptable for manual development of transformation templates, where such XMI code will be even mixed with template metacode. For this reason, we simply choose to introduce a separate XMI element for every OAL action, which contains the action's syntax as a character string. The following paragraphs describe the semantics of the four steps for the example product line.

Step A—Define an Object Type. In this step, the user makes the configuration choices about the name of the object type, the names of the attributes, and the primitive types for the attributes. The produced concrete transformation represents the object type as a class in the model and adds the *globId* and *objId* identifiers along with the user-defined attributes to the class. Afterwards, it adds the operations (implemented in OAL) *createObject*, *findByGlobId*, and *copy* to the class. The step can be repeated many times to define further object types.

Step B—Make an Object Type Versionable. In this step, the user chooses an object type that needs to support versioning operations. The maximum number of successors is defined for the type. In addition, the user specifies whether versions will include version name and version creation date. The produced concrete transformation adds the attributes *verId*, *frozen* (denoting whether a version is frozen), and *successorCount* (denoting the current number of successors) to the class representing the object type. If this was desired by the developer, it also adds the attributes *versionName* and *versionDate*. Afterwards, the transformation includes the ancestor-successor relationship type as an association in the model. Finally, version management operations (implemented in OAL) *createSuccessor, freeze*, *merge*, *getAncestor, getSuccessors*, *getPathToRoot,* and *getVersions* are added to the class. The step can be repeated many times to make different object types versionable.

Step C—Define a Relationship Type. In this step, the user defines a relationship type between two object types. For this purpose, the object types to relate need to be chosen and a name for the relationship type needs to be specified. The user decides on the multiplicities of both relationship ends, the floating property, and operation propagation settings. The produced concrete transformation adds a class for every CVC (only for floating ends), operations for creating and deleting relationships, and operations for manipulating pin settings (only for floating ends). The existing operations (added in previous transformations), e.g., *createSuccessor*, are

modified to support operation propagation, if required. The step can be repeated many times to define further relationship types.

Step D—Define a Workspace Type. In this step, the user defines a workspace type by choosing a name for the type and selecting the contained object types with their multiplicities. The produced concrete transformation represents the workspace type as a class in the model and the attachment-relationship types as associations. Afterwards, it adds *attach* and *detach* operations, operations for navigating to attached objects, and operations for selecting and unselecting the workspace. The existing operations for filtered navigation are modified to support filtered navigation within a selected workspace. The step can be repeated many times to define further workspace types.

6.4 A Metamodel for Workflows Models

The definition of workflow models and execution of the configuration process is supported by a separate metamodel which is organized in three packages, described in the following paragraphs.

Core Elements. This package is a backbone for the other two packages. It includes the abstract element *ModelElement* (the root superclass for every other element in the workflow model), and concrete elements for representing a workflow model, configuration steps, data containers, templates, and transitions.

User Choices. This package includes elements for representing user choices presented to the user in the graphic interface and hierarchically organizing the choices into groups.

Evaluations. This package contains elements for representing variables in data containers, types of these variables, and different functions. The functions are used for evaluating default values for user choices, transition conditions, and values in the context which are used by the transformation template.

6.5 Evaluations

We evaluated the approach of workflow-supported model transformations from different perspectives. First, we used the approach to automatically assemble a model of a versioning system that allows versioning of simple UML models, i.e., we used the *Core* package of the UML Metamodel as the information model. Every UML Metamodel element has been represented by a separate object type. Five workspaces, *backbone elements*, *relationship elements*, *dependencies elements*, *classifier elements*, and *auxiliary elements*, have been defined for semantically related elements within the *Core* package (see the UML specification [26] for a detailed

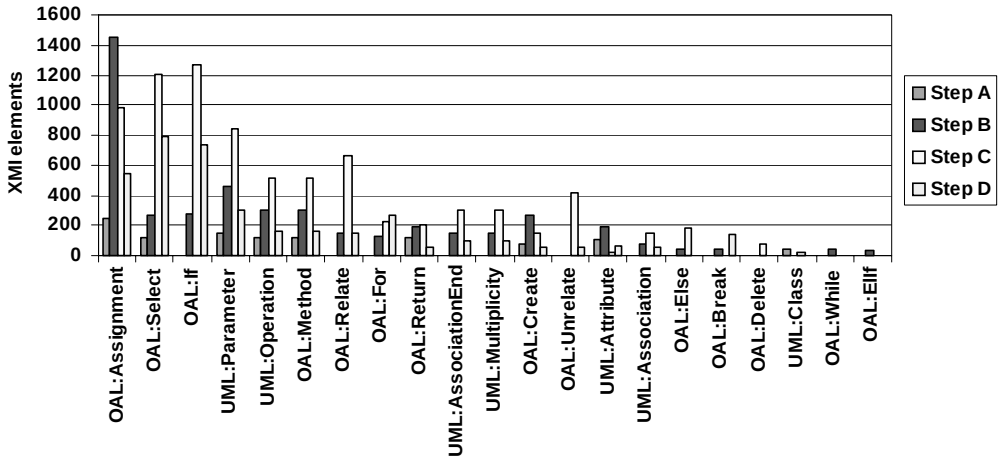

Fig. 21. XMI elements added in the configuration steps (UML Metamodel)

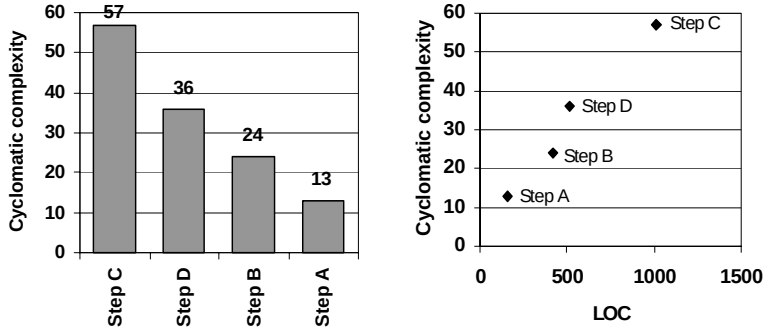

Fig. 22. Cyclomatic complexity of the templates

overview of elements that belong to each of these categories). For such a versioning system, we observed the number of XMI elements added in each step (see Fig. 21, not all element categories are included in the diagram). This comparison revealed that the steps are very diverse based on the type of the elements added to the model.

Comparing the number of elements added in every configuration step to the number of user choices required in a configuration step reveals that *Step D* is most productive with 70 elements per user choice while *Step C* is least productive with 12 elements per user choice.

In our further evaluation, we observed the properties of transformation templates. As illustrated in Fig. 22, the template used for *Step C* proves most complex in terms of cyclomatic complexity, followed by the templates used for *Steps D*, *B*, and *A*. The correlation diagram confirms a linear relationship between the cyclomatic complexity and a LOC-value of a template, indicating that despite the different semantics added to the model in every configuration step, the complexity remains evenly distributed across the templates.

As illustrated in Fig. 23, the templates contain between 416 and 65 XMI elements in their static parts (note that more complex templates also have larger static parts). However, for the

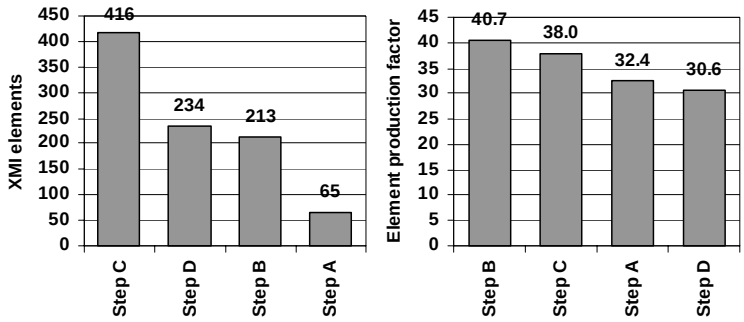

Fig. 23. XMI elements in the template's static parts and element production factor

versioning system used to version UML models, we found that the size of the static part is in no relationship to the average number of elements produced in a step (element production factor).

6.6 Lessons Learned

What are the most important lessons that we learn from workflow-supported model transformations? When comparing the productivity of the steps (the number of elements added in every step compared to the number of user choices), a major drawback is the uniform treatment of different XMI elements and choices. We agree that not all elements are the same and would thus require different effort to be included in the model in a manual modeling process. A possible solution would be to assign empirically-determined weights for the modeling elements. Likewise, the number of user choices in a configuration step should not be considered a direct criterion for configuration effort in the step, i.e., some configuration choices are more difficult to make than others. A linear growth trend of cyclomatic complexity with the LOC value can be observed for the templates. For the versioning system based on package *Core* of the UML Metamodel, the templates were capable of producing between 41 and 31 times as many XMI elements as contained in their static parts.

Many of the problems related to the use of templates that were recognized in generating middleware-based versioning systems also appear for workflow-supported model transformations. These include an advanced environment support for easily separating code from metacode in template development and running the template engine from within this environment for arbitrary context values. It should be possible to track configuration choices and templates parts back to the feature model. Finally, even in this project, we developed most of the metacode by first examining and comparing several examples of manually developed concrete transformations.

7 Related Work

We have limited our discussion on versioning systems only to the functionality needed to understand the type of systems we deal with in Sects. 4–6. For surveys on versioning systems, we refer to the following work. Katz [17] discusses a variety of existing terminologies and mechanisms for representing versioned data, primarily from the CAD domain, and proposes a unification of presented version models. Conradi and Westfechtel [8] examine different version models in commercial systems and research prototypes and discuss fundamental concepts used in versioning such as revisions, variants, configurations, and changes. Finally, Whitehead and Gordon [39] use containment data models (a specialized form of ER models) to represent the version models of 11 different versioning systems.

An extensive overview of different model-to-model and model-to-code transformation approaches is given by Czarnecki and Helsen [10]. The authors identify the template language we use in Sects. 4 and 6 as a template-based model-to-code approach. They note that, in general, the structure of a template closely resembles the code to be generated. On the other hand, the patterns contained by a template can be syntactically or semantically incorrect, which makes it difficult to assure the correctness of generated code during template development.

Braga and Masiero [6] propose a system for controlled framework instantiation. The authors claim that instantiation is a complex process which requires an in-depth knowledge of the framework's hot spots. For this purpose, they support the developer in the instantiation process by a set of mandatory and optional patterns which are applied by following predefined paths. The paths are defined using a formal model. This idea of system configuration is very similar to our workflow-supported model transformations which allow a controlled assembly of a model that describes the desired system.

8 Conclusion

We summarize the ideas presented in this paper by giving a comparison of presented approaches (Sect. 8.1), outlining the benefits of MDSD based on our experience (Sect. 8.2), and finally giving an outlook for this research domain (Sect. 8.3).

8.1 Comparison of Presented Approaches

The approaches described in this paper all deal with MDSD, but still differ in (i) the extent to which they support different product lines, (ii) specification and generation method, and

Characteristic	A generator for middleware-based versioning systems (Sect. 4)	Domain-specific languages for database applications (Sect. 5)	Workflow-supported model transformations (Sect. 6)
System used for...	generating middleware-based versioning systems	defining DSLs that map to SQL	automated assembly of specifications in a general-purpose modeling language
Product line support	Only the product line for versioning systems	Generic system, the product line for versioning systems is used as an example. Every product line requires a separate domain-specific metamodel and a set of translation, visualization, and editing routines.	Generic system, the product line for versioning systems is used as an example. Every product line requires a separate workflow model and a set of transformation templates.
Specification language	UML static structure models with a profile for the product line for versioning systems. The profile essentially represents a DSL.	Defined as a metamodel using a built-in meta-metamodel. The metamodel is essentially a DSL. A DSL for versioning systems is used as an example.	Any language with a MOF-based metamodel. Executable UML used as an example.
Target platform for generated systems	middleware and RDBMS	RDBMS	depends on the generator for the language used
Specification tool	any UML modeling tool	generated CASE tool	a workflow engine that guides the application of model transformations
Specification supported by...	no special support, except for the features of the UML modeling tool	visualization and editing routines specified in Java	model transformations implemented using templates
Generator / mapping tool	template-based generator	generated CASE tool, precompiler, or a domain-specific driver	any generator that understands the language used
Generation supported by...	code templates	translation routines specified in Java	depends on the generator for the language used

Table 6. Characteristics of MDSD approaches described in this paper

(iii) the target platforms for generated systems. The differences between the presented ideas are given by Table 6.

8.2 Benefits of Model-Driven Approaches

We divide the benefits of model-driven approaches into *productivity* and *performance benefits*.

Productivity Benefits. Both in Sects. 4 and 5, we obtained a complete implementation of a versioning system with substantially less effort than would normally be required for a manual development process. A generative approach is useful even when considering the development of a single system, because many templates are applied more than once in a specific configuration. In both cases the specification is easy to understand even for non-experts and easy to communicate. Domain-specific data manipulation statements in Sect. 5 as opposed to a combination of many SQL statements make the application code easier to understand and reengineer and less error-prone. In workflow-supported model transformations, a semantically correct specification of a versioning system in a general-purpose modeling language is assembled with substantially less effort than would otherwise be required for a manual assembly.

Performance Benefits. As illustrated in Sects. 4 and 5, complex optimization decisions can be implemented in the generator (the metacode) and made transparent for the user. The performance benefits were illustrated by the comparison of a generated towards a framework-based versioning system in Sect. 4 and the comparison of four translation variants in Sect. 5.

8.3 Outlook

Due to the fact that MDSD is a relatively new field, no verified measures exist that could precisely evaluate the return-on-investment. The fact that a piece of metacode can be reused even when generating a single system can be misleading because metacode development can prove far more demanding than the development of manual implementation, even in the case of many systems with similar properties. For this reason, we claim that a set of measures that would allow us a concise evaluation of a given MDSD approach is needed. The measures should be empirically defined on a large set of implemented product lines and cover distinct phases of domain engineering.

Acknowledgments

This work would not be possible without the support of Wolfgang Mahnke and Hans-Peter Steiert. Our students Christian Gebauer, Martin Husemann, Christian Weber, Amine Chatti, and Benedikt Eger invested great efforts in implementing and evaluating the approaches described in this paper.

References

1. Agrawal, A.: Graph Rewriting and Transformation (GReAT): A Solution for the Model Integrated Computing (MIC) Bottleneck, in: Proc. ASE'03, Montreal, Oct. 2003, pp. 364–368
2. Ambler, S.W.: Agile Modeling, John Wiley & Sons, 2002
3. The Apache Jakarta Project: Velocity User's Guide, available from: http://jakarta.apache.org/velocity/
4. Bernstein, P.A., Bergstraesser, T., Carlson, J, Pal, S., Sanders, P., Shutt, D.: Microsoft Repository Version 2 and the Open Information Model. In: Information Systems 22:2, 1999, pp. 71–98
5. Boocock, P.: Jamda Model Compiler Framework, available from: http://jamda.sourceforge.net/
6. Braga, R.T.V., Masiero, P.C.: Building a Wizard for Framework Instantiation, in: Proc. OOIS'03, Geneva, Sept. 2003, pp. 95–106
7. Brinkkemper, S.: Method Engineering: Engineering of Information Systems Development Methods and Tools, in: Information and Software Technology 38:4, 1996, pp. 275–280
8. Conradi, R., Westfechtel, B.: Version Models for Software Configuration Management, in: ACM Computing Surveys 30:2, 1998, pp. 232–282
9. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley, 2000
10. Czarnecki, K., Helsen, S.: Classification of Model Transformation Approaches, in: Proc. OOPSLA'03 Workshop on Generative Techniques in the Context of Model Driven Architecture, Anaheim, Oct. 2003
11. D'Souza, D.F., Wills, A.C.: Objects, Components, and Frameworks With UML: The Catalysis Approach, Addison-Wesley, 1998
12. Frankel, D.: Model Driven Architecture: Applying MDA to Enterprise Computing, Wiley Publishing, 2003
13. Halstead, M.H.: Elements of Software Science, Elsevier, 1977

14. IBM Corp. Rational Unified Process for System Engineering, RUP SE 1.1, IBM Rational White Paper TP 165A, Feb. 2002
15. Jaakkola, H., Thalheim, B. Visual SQL—High-Quality ER-Based Query Treatment, in: Workshop Proc. ER'03, Chicago, Oct. 2003, pp. 129–139
16. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study, Technical Report CMU/SEI-90-TR-21, SEI, Carnegie Mellon University, Nov. 1990
17. Katz, R.H.: Toward a Unified Framework for Version Modeling in Engineering Databases, in: ACM Computing Surveys 22:4, 1990, pp. 375–408
18. Kießling, W., Köstler, G.: Preference SQL - Design, Implementation, Experiences, in: Proc. VLDB'02, Hong Kong, Aug. 2002, pp. 990–1001
19. Marschall, F., Braun, P.: Model Transformations for the MDA with BOTL, in: Proc. MDAFA 2003, Enschede, May 2003
20. McCabe, T.J.: A Complexity Measure, in: IEEE Transactions on Software Engineering 2:4, 1976, pp. 308–320
21. Mellor, S.J., Balcer, M.: Executable UML: A Foundation for Model-Driven Architecture, Addison-Wesley, 2002
22. Microsoft Corp.: Microsoft Developer Network: Programming Meta Data Services Applications, available from: http://msdn.microsoft.com/library/default.asp
23. OMG: Model Driven Architecture (MDA), OMG document ormsc/2001-07-01, July 2001
24. OMG: Common Warehouse Metamodel (CWM) Specification, Version 1.0, Vol. 1, Oct. 2001
25. OMG: XML Metadata Interchange (XMI) Specification, Version 1.2, Jan. 2002
26. OMG: Unified Modeling Language (UML) Specification, Version 1.5, March 2003
27. Project Technology, Inc.: OAL Manual, Version 1.4, available from http://www.projtech.com/pubs/
28. QVT-Partners: MOF 2.0 Query / Views / Transformations RFP, Revised Submission, Version 1.1, Aug. 2003
29. Ralyté, J., Rolland, C., Deneckère, R: Towards a Meta-tool for Change-Centric Method Engineering: A Typology of Generic Operators, in: Proc. CAiSE'04, Riga, June 2004, pp. 202–218
30. Rumbaugh, J.E.: Controlling Propagation of Operations Using Attributes on Relations, in: Proc. OOPSLA'88, San Diego, Nov. 1988, pp. 285–296
31. Rumbaugh, J.E.: What is a Method? in: JOOP 8:6, 1995, pp. 10–16
32. Saeki, M.: Toward Automated Method Engineering: Supporting Method Assembly in CAME, presentation at EMSI-SE'03, Geneva, Sept. 2003
33. Simonyi, C. The Death of Computer Languages, the Birth of Intentional Programming, Tech. Report MSR-TR-95-52, Microsoft Research, Sept. 1995
34. Stapleton, J.: DSDM: Business Focused Development, Addison-Wesley, 2003
35. Sturm, T., v. Voss, J., Boger, M.: Generating Code from UML with Velocity Templates, in: Proc. UML 2002, Dresden, Oct. 2002
36. SysML Partners: Systems Modeling Language (SysML) Specification, Ver. 0.9 Draft, Jan. 2005
37. v. Doren, E.: Cyclomatic Complexity (Software Technology Roadmap). SEI, Carnegie Mellon University, July 2000, available from: http://www.sei.cmu.edu/
38. Visser, E.: A Survey of Rewriting Strategies in Program Transformation Systems, in: Electronic Notes in Theoretical Computer Science 57, 2001
39. Whitehead, E.J., Jr., Gordon, D.: Uniform Comparison of Configuration Management Data Models, in: Proc. SCM-11, Portland, May 2003, pp. 70–85
40. Zhang, N.: Enriched Relationship Processing in Object-Relational Database Management Systems, in. Proc. CODAS'01, Beijing, April 2001, pp. 53–62