Metaprogramming for Relational Databases

Jernej Kovse, Christian Weber, and Theo Härder

Department of Computer Science
Kaiserslautern University of Technology
P.O. Box 3049, D-67653 Kaiserslautern, Germany
{kovse, c_weber, haerder}@informatik.uni-kl.de

Abstract. For systems that share enough structural and functional commonalities, reuse in schema development and data manipulation can be achieved by defining problem-oriented languages. Such languages are often called domain-specific, because they introduce powerful abstractions meaningful only within the domain of observed systems. In order to use domain-specific languages for database applications, a mapping to SQL is required. In this paper, we deal with metaprogramming concepts required for easy definition of such mappings. Using an example domain-specific language, we provide an evaluation of mapping performance.

1 Introduction

A large variety of approaches use SQL as a language for interacting with the database, but at the same time provide a separate problem-oriented language for developing database schemas and formulating queries. A *translator* maps a statement in such problem-oriented language to a series of SQL statements that get executed by the DBMS. An example of such a system is *Preference SQL*, described by Kießling and Köstler [8]. Preference SQL is an SQL extension that provides a set of language constructs which support easy use of *soft preferences*. This kind of preferences is useful when searching for products and services in diverse e-commerce applications where a set of strictly observed hard constraints usually results in an empty result set, although products that approximately match the user's demands do exist. The supported constructs include *approximation* (clauses `AROUND` and `BETWEEN`), *minimization/maximization* (clauses `LOWEST`, `HIGHEST`), *favorites and dislikes* (clauses `POS`, `NEG`), *pareto accumulation* (clause `AND`), and *cascading of preferences* (clause `CASCADE`) (see [8] for examples).

In general, problem-oriented programming languages are also called *domain-specific languages* (DSLs), because they prove useful when developing and using systems from a predefined *domain*. The systems in a domain will exhibit a range of similar structural and functional features (see [4,5] for details), making it possible to describe them (and, in our case, query their data) using higher-level programming constructs. In turn, these constructs carry semantics meaningful only within this domain. As the activity of using these constructs is referred to as programming, defining such con-

structs and their mappings to languages that can be compiled or interpreted to allow their execution is referred to as *metaprogramming*.

This paper focuses on the application of metaprogramming for relational databases. In particular, we are interested in concepts that guide the implementation of fast mappings of custom languages, used for developing database schemas and manipulating data, onto SQL-DDL and SQL-DML. The paper is structured as follows. First, in Sect. 2, we further motivate the need for DSLs for data management. An overview of related work is given by Sect. 3. Our system prototype (DSL-DA – domain-specific languages for database applications) that supports the presented ideas is outlined in Sect. 4. A detailed performance evaluation of a DSL for the example product line will be presented in Sect. 5. Sect. 6 gives a detailed overview of metaprogramming concepts. Finally, in Sect. 7, we summarize our results and give some ideas for the future work related to our approach.

2 Domain-Specific Languages

The idea of DSLs is tightly related to *domain engineering*. According to Czarnecki and Eisenecker [5], domain engineering deals with collecting, organizing, and storing past experience in building systems in form of *reusable assets*. In general, we can rely that a given asset can be reused in a new system in case this system possesses some structural and functional similarity to previous systems. Indeed, systems that share enough common properties are said to constitute a *system family* (a more market-oriented term for a system family is a *software product-line*). Examples of software product-lines are extensively outlined by Clements and Northrop [4] and include satellite controllers, internal combustion engine controllers, and systems for displaying and tracing stock-market data. Further examples of more data-centric product lines include CRM and ERP systems. Our example *product line for versioning systems* will be introduced in Sect. 4.

Three approaches can be applied to allow the reuse of "assets" when developing database schemas for systems in a data-intensive product line.

Components: Schema components can be used to group larger reusable parts of a database schema to be used in diverse systems afterwards (see Thalheim [16] for an extensive overview of this approach). Generally, the modularity of system specification (which components are to be used) directly corresponds to the modularity of the resulting implementation, because a component does not influence the internal implementation of other components. This kind of specification transformations towards the implementation is referred to as *vertical transformations* or *forward refinements* [5].

Frameworks: Much like software frameworks in general (see, for example, Apache Struts [1] or IBM San Francisco [2]), schema frameworks rely on the user to extend them with system-specific parts. This step is called *framework instantiation* and requires certain knowledge of how the missing parts will be called by the framework. Most often, this is achieved by extending superclasses defined by the framework or implementing *call-back methods* which will be invoked by mechanisms such as *reflection*. In a DBMS, application logic otherwise captured by such methods can be defined by means of constraints, trigger conditions and actions, and stored procedures. A detailed

overview of schema frameworks is given by Mahnke [9]. Being more flexible than components, frameworks generally require more expertise from the user. Moreover, due to performance reasons, most DBMSs restrain from dynamic invocation possibilities through method overloading or reflection (otherwise supported in common OO programming languages). For this reason, schema frameworks are difficult to implement without middleware acting as a mediator for such calls.

Generators: Schema generators are, in our opinion, the most advanced approach to reuse and are the central topic of this paper. A schema generator acts much like a compiler: It transforms a high-level specification of the system to a schema definition, possibly equipped with constraints, triggers, and stored procedures. In general, the modularity of the specification does not have to be preserved. Two modular parts of the specification can be interwoven to obtain a single modular part in the schema (these transformations are called *horizontal transformations*; in case the obtained part in the schema is also refined, for example, columns not explicitly defined in the specification are added to a table, this is called an *oblique transformation*, i.e., a combination of a horizontal and a vertical transformation.)

It is important to note that there is no special "magic" associated with schema generators that allows them to obtain a ready-to-use schema out of a short specification. By narrowing the domain of systems, it is possible to introduce very powerful language abstractions that are used at the specification level. Due to similarities between systems, these abstractions aggregate a lot of semantics that is dispersed across many schema elements. Because defining this semantics in SQL-DDL proves labour-intensive, we rather choose to define a special domain-specific DDL (DS-DDL) for specifying the schema at a higher level of abstraction and implement the corresponding mapping to SQL-DDL. The mapping represents the "reusable asset" and can be used with any schema definition in this DS-DDL. The data manipulation part complementary to DS-DDL is called DS-DML and allows the use of domain-specific query and update statements in application programs. Defining custom DS-DDLs and their mappings to SQL-DDL as well as fast translation of DS-DML statements is the topic we explore in this paper.

3 Related Work

Generators are the central idea of the OMG's *Model Driven Architecture* (MDA) [13] which proposes the specification of systems using standardized modeling languages (UML) and automatic generation of implementations from models. However, even OMG notices the need of supporting custom domain-specific modeling languages. As noted by Frankel [6], this can be done in three different ways:

- *Completely new modeling languages*: A new DSL can be obtained by defining a new MOF-based metamodel.
- *Heavyweight language extensions*: A new DSL can be obtained by extending the elements of a standardized metamodel (e.g., the UML Metamodel).
- *Lightweight language extensions*: A new DSL can be obtained by defining new language abstractions *using the language itself*. In UML, this possibility is supported by *UML Profiles*.

The research area that deals with developing custom (domain-specific) software engineering methodologies well suited for particular systems is called *computer-aided method engineering* (CAME) [14]. CAME tools allow the user to describe an own modeling method and afterwards generate a CASE tool that supports this method. For an example of a tool supporting this approach, see MetaEdit+ [11].

The idea of a rapid definition of domain-specific programming languages and their mapping to a platform where they can be executed is materialized in Simonyi's work on *Intentional Programming* (IP) [5,15]. IP introduces an IDE based on *active libraries* that are used to import language abstractions (also called *intentions*) into this environment. Programs in the environment are represented as source graphs in which each node possesses a special pointer to a corresponding abstraction. The abstractions define *extension methods* which are metaprograms that specify the behavior of nodes. The following are the most important extension methods in IP.

- *Rendering and type-in methods*. Because it is cumbersome to edit the source graph directly, rendering methods are used to visualize the source graph in an editable notation. Type-in methods convert the code typed in this notation back to the source graph. This is especially convenient when different notations prove useful for a single source graph.
- *Refactoring methods*. These methods are used to restructure the source graph by factoring out repeating code parts to improve reuse.
- *Reduction methods*. The most important component of IP, these methods reduce the source graph to a graph of low-level abstractions (also called *reduced code* or *R-code*) that represent programs executable on a given platform. Different reduction methods can be used to obtain the R-code for different platforms.

How does this work relate to our problem? Similar as in IP, we want to support a custom definition of abstractions that form both a custom DS-DDL and a custom DS-DML. We want to support the rendering of source graphs for DS-DDL and DS-DML statements to (possibly diverse) domain-specific textual representations. Most importantly, we want to support the reduction of these graphs to graphs representing SQL statements that can be executed by a particular DBMS.

4 DSL-DA System

In our DSL-DA system, the user starts by defining a domain-specific (DS) metamodel that describes language abstractions that can appear in the source graph (the language used for defining metamodels is a simplified variant of the MOF Model) for the DS-DDL. We used the system to fully implement a DSL for the *example product line of versioning systems* which we also use in the next section for the evaluation of our approach. In this product line, each system is used to store and version objects (of some object type) and relationships (of some relationship type). Thus individual systems differ in their type definitions (also called information models [3]) as well as other features illustrated in the DS-DDL metamodel in Fig. 1 and explained below.

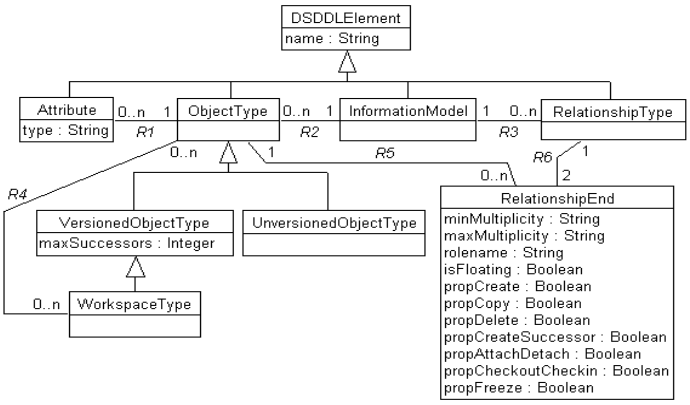

Fig. 1. DS-DDL metamodel for the example product line

- Object types can be *versioned* or *unversioned*. The number of direct successors to a version can be limited to some number (*maxSuccessors*) for a given versioned object type.
- Relationship types connect to object types using either *non-floating* or *floating* relationship ends. A non-floating relationship end connects directly to a particular version as if this version were a regular object. On the other hand, a floating relationship end maintains a user-managed subset of all object versions for each connected object. Such subsets are called *candidate version collections* (CVC) and prove useful for managing configurations. In *unfiltered navigation* from some origin object, all versions contained in every connected CVC will be returned. In *filtered navigation*, a version preselected for each CVC (also called the *pinned version*) will be returned. In case there is no pinned version, we return the latest version from the CVC.
- Workspace objects act as containers for other objects. However, only one version of a contained object can be present in the workspace at a time. In this way, workspaces allow a version-free view to the contents of a versioning system. When executed within a workspace, filtered navigation returns versions from the CVC that are connected to this workspace and ignores the pin setting of the CVC.
- Operations *create object*, *copy*, *delete*, *create successor*, *attach/detach* (connects/disconnects an object to/from a workspace), *freeze*, and *checkout/checkin* (locks/unlocks the object) can propagate across relationships.

A model expressed using the DS-DDL metamodel from Fig. 1 will represent a source graph for a particular DS-DDL schema definition used to describe a given versioning system. To work with these models (manipulate the graph nodes), DSL-DA uses the DS-DDL metamodel to generate a *schema editor* that displays the graphs in a tree-like form (see the left-hand side of Fig. 2). A more convenient graphical notation of a source graph for our example versioning system that we will use for the evaluation in the next section is illustrated in Fig. 3.

The metamodel classes define rendering and type-in methods that render the source graph to a textual representation and allow its editing (right-hand side of Fig. 2). More importantly, the metamodel classes define reduction methods that will reduce the

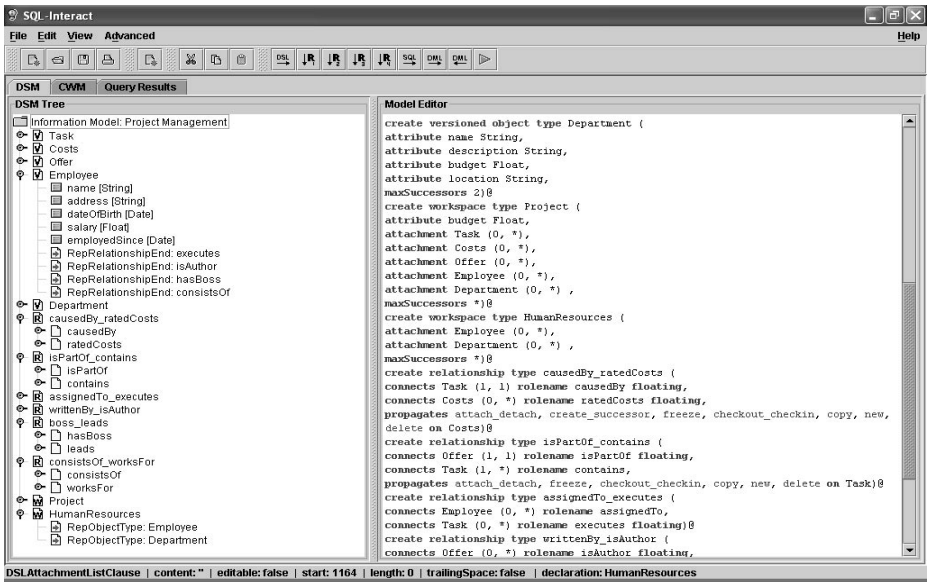

Fig. 2. DS-DDL schema development with the generated editor

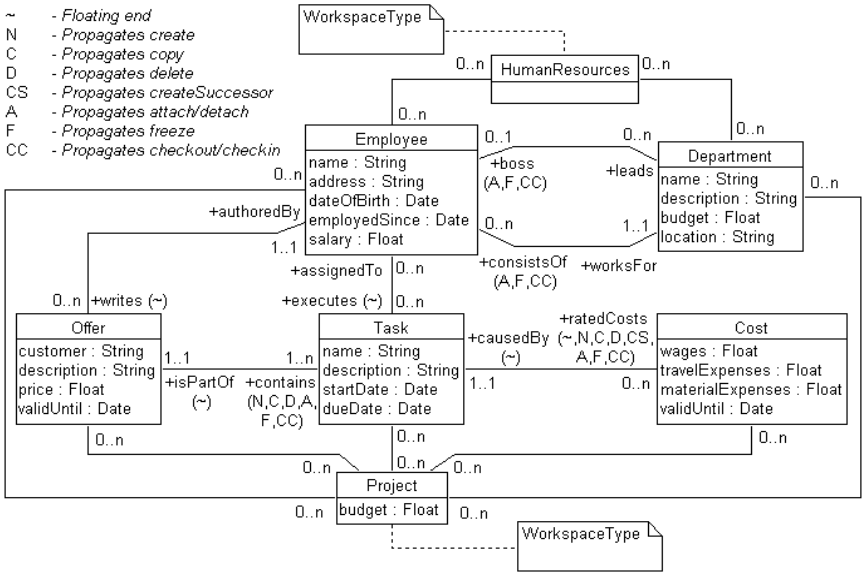

Fig. 3. Example DS-DDL schema used in performance evaluation

source graph to its representation in SQL-DDL. In analogy with the domain-specific level of the editor, the obtained SQL-DDL schema is also represented as a source graph; the classes used for this graph are the classes defined by the package *Relational* of the OMG's Common Warehouse Metamodel (CWM) [12]. The rendering methods of these

Table 1. Examples of DS-DML statements

Statement	Explanation
SELECT Task.* FROM Department-consistsOf->Employee-executes->Task WHERE Department.globalId = 502341	Get all tasks executed by employees of a given department (all three objects are versioned). Note that the fact that the relationship end *executes* is floating (i.e. filtered navigation will be used) is transparent for the user.
CREATE SUCCESSOR OF OBJECT Task USE WORKSPACE Project WHERE globalId = 235711 AND Task WHERE objectId = 982	Create a successor version to a version of a task. The version graph for the task is identified by the *objectId*. The successor is to be created to the version attached to the workspace with a given *globalId*. Note that according to the DS-DDL schema, the operation will propagate to connected costs.
GET ALTERNATIVES OF Employee WHERE globalId = 234229	Get the alternative versions (versions that have the same predecessor) of a given employee version

classes are customizable so that by rendering the SQL-DDL source graphs, SQL-DDL schemas in SQL dialects of diverse DBMS vendors can be obtained.

Once an SQL-DDL schema is installed in a database, how do we handle statements in DS-DML (three examples of such statements are given by Table 1)? As for the DS-DDL, there is a complementary DS-DML metamodel that describes language abstractions of the supported DS-DML statements. This metamodel can be simply defined by first coming up with an EBNF for DS-DML and afterwards translating the EBNF symbols to class definitions in a straightforward fashion. The EBNF of our DS-DML for the sample product line for versioning systems is available through [17]. DS-DML statements can then be represented as source graphs, where each node in the graph is an instance of some class from the DS-DML metamodel. Again, metamodel classes define reduction methods that reduce the corresponding DS-DML source graph to an SQL-DML source graph, out of which SQL-DML statements can be obtained through rendering.

DS-DML is used by an application programmer to embed domain-specific queries and data manipulation statements in the application code. In certain cases, the general structure of a DS-DML statement will be known at the time the application is written and the parameters of the statement will only need to be filled with user-provided values at run time. Since these parameters do not influence the reduction, the reduction from DS-DML to SQL-DML can take place using a *precompiler*. Sometimes, however, especially in the case of Web applications, the structure of the DS-DML query will depend on the user's search criteria and other preferences and is thus not known at compile time. The solution in this case is to wrap the native DBMS driver into a *domain-specific driver* that performs the reduction at run time, passes the SQL-DML statements to the native driver, and restructures the result sets before returning them to the user, if necessary. To handle both cases where query structure is known at compile time and when it is not, DSL-DA can generate both the precompiler and the domain-specific driver from the DS-DML metamodel, its reduction methods, and its rendering methods for SQL-DML. We assumed the worst-case scenario in which all SQL-DML statements need to be reduced at run time for our evaluation in the next section to examine the effect of run time reduction in detail.

5 Evaluation of the Example Product Line

The purpose of the evaluation presented in this section is to demonstrate the following.

- Even for structurally complex DS-DML statements, the reduction process carried out at run time represents a very small proportion of costs needed to carry out the SQL-DML statements obtained by reduction.
- DS-DDL schemas that have been reduced to SQL-DDL with certain optimizations in mind imply reduction that is more difficult to implement. Somewhat surprisingly, this does not necessarily mean that such reduction will also take more processing time. Optimization considerations can significantly contribute to a faster execution of DS-DML statements once reduced to SQL-DML.

To demonstrate both points, we implemented four very different variants of both DS-DDL and DS-DML reduction methods for the example product line. The DS-DDL schema from Fig. 3 has thus been reduced to four different SQL-DDL schemas. In all four variants, object types from Fig. 3 are mapped to tables (called *object tables*) with the specified attributes. An object version is then represented as a tuple in this table. The identifiers in each object table include an *objectId* (all versions of a particular object, i.e., all versions within the same version tree, possess the same *objectId*), a *versionId* (identifies a particular version within the version tree) and a *globalId*, which is a combination of an *objectId* and a *versionId*. The four reductions differ in the following way.

- *Variant 1*: Store all relationships, regardless of relationship type, using a single "generic" table. For a particular relationship, store the *origin globalId*, *objectId*, *versionId* and the *target rolename*, *globalId*, *objectId*, and *versionId* as columns. Use an additional column as a flag denoting whether the target version is pinned.
- *Variant 2*: Use separate tables for every relationship type. In case a relationship type defines no floating ends or two floating ends, this relationship type can be represented by a single table. In case only one relationship end is floating, such relationship type requires two tables, one for each direction of navigation.
- *Variant 3*: Improve *Variant 2* by considering maximal multiplicity of *1* on non-floating ends. For such ends, the *globalId* of the connected target object is stored as a column in the object table of the origin object.
- *Variant 4*: Improve *Variant 3* by considering maximal multiplicity of *1* of floating ends. For such ends, the *globalIds* of the pinned version and the latest version of the CVC for the target object can be stored as columns in the object table of the origin object.

Our benchmark, consisting of 141,775 DS-DML statements was then run using four different domain-specific drivers corresponding to four different variants of reduction. To eliminate the need of fetching metadata from the database, we assumed that, once defined, the DS-DDL schema does not change, so each driver accessed the DS-DDL schema defined in Fig. 3 directly in the main memory. The overall time for executing a DS-DML statement is defined as $t_{DS} = t_{par}+t_{red}+t_{ren}+t_{SQL}$, where t_{par} is the required DS-DML parsing time, t_{red} the time required for reduction, t_{ren} the time required for rendering all resulting SQL-DML statements, and t_{SQL} the time used to carry out these statements. Note that t_{par} is independent of the variant, so we were mainly interested in the remaining three times as well as the overall time. The average t_{DS}, t_{red}, t_{ren} and t_{SQL}

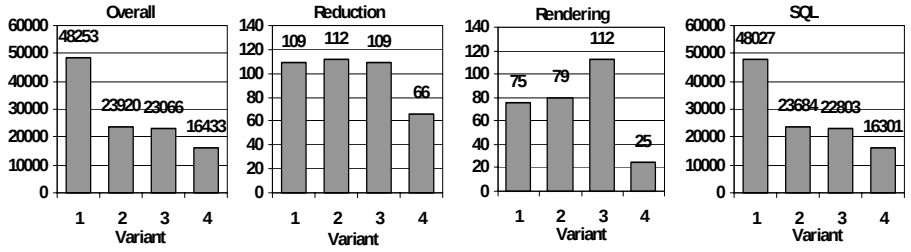

Fig. 4. Execution times for the category of *select* statements

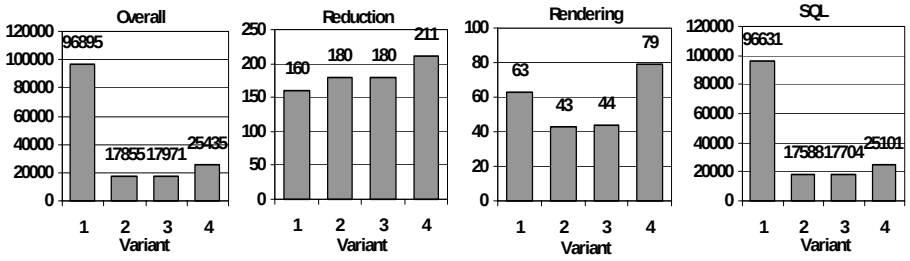

Fig. 5. Execution times for the category of *create relationship* statements

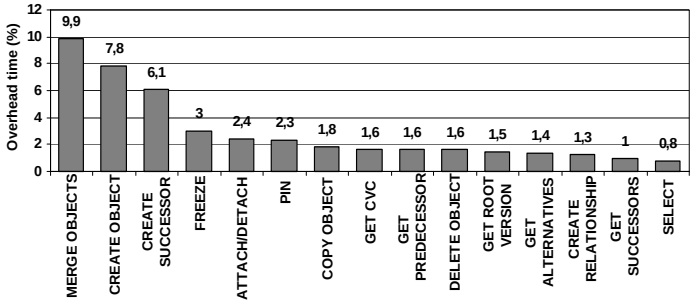

Fig. 6. Overhead due to DS-DML parsing, reduction and rendering

values (in µs) for the category of *select* statements are illustrated in Fig. 4. This category included queries over versioned data within and outside workspaces that contained up to four navigation steps. As evident from Fig. 4, *Variant 4* demonstrates a very good t_{SQL} performance and also allows the fastest reduction. On the other hand, due to materialization of the *globalIds* of pinned and latest versions for CVCs in *Variant 4*, *Variant 2* proves faster for manipulation (i.e., creation and deletion of relationships). The values for the category of *create relationship* statements are illustrated in Fig. 5.

Most importantly, the overhead time required due to the domain-specific driver $t_{dr}=t_{par}+t_{red}+t_{ren}$ proves to be only a small portion of t_{DS}. As illustrated in Fig. 6, when using *Variant 4*, the portion t_{dr}/t_{DS} is lowest (0.8%) for the category of *select* statements

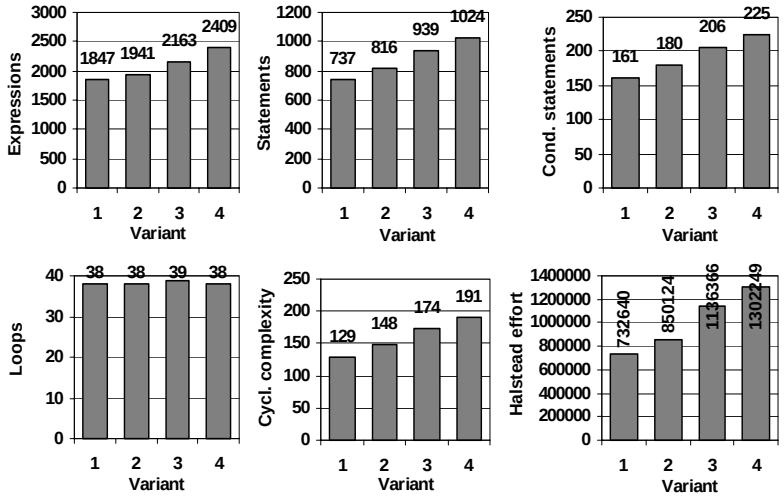

Fig. 7. Properties of reduction methods

and highest (9.9%) for *merge* statements. When merging two versions (denoted as primary and secondary version), their attribute values have to be compared to their so-called *base (latest common) version* in the version graph to decide which values should be used for the result of the merging. This comparison, which is performed in the driver, accounts for a high t_{red} value (9.1% of t_{DS}). Note that t_{SQL} is the minimal time an application spends executing SQL-DML statements in any case (with or without DS-DML available) to provide the user with equivalent results: Even without DS-DML, the programmer would have to implement data flows to connect sequences of SQL-DML statements to perform a given operation (in our evaluation, we treat data flows as part of t_{red}).

How difficult is it to implement the DS-DML reduction methods? To estimate this aspect, we used measures such as the count of expressions, statements, conditional statements, loops, as well as McCabe's cyclomatic complexity [10] and Halstead effort [7] on our Java implementation of reduction methods. The summarized results obtained using these measures are illustrated by Fig. 7. All measures, except for the count of loops confirm an increasing difficulty to implement the reduction (e.g., the Halstead effort almost doubles from *Variant 1* to *Variant 4*). Is there a correlation between the Halstead effort for writing a method and the times t_{red} and t_{SQL}? We try to answer this question in Fig. 8. Somewhat surprisingly, a statement with a reduction more difficult to implement will sometimes also reduce faster (i.e., an increase in Halstead effort does not necessarily imply an increase in t_{red}), which is most evident for the category of *select* statements. The explanation is that even though the developer has to consider a large variety of different reductions for a complex variant (e.g., *Variant 4*), once the driver has found the right reduction (see Sect. 6), the reduction can proceed even faster than for a variant with less optimization considerations (e.g., *Variant 1*). For all categories in Fig. 8, a decreasing trend for t_{SQL} values can be observed. However, in categories that manipulate the state of the CVC (note that operations from the category

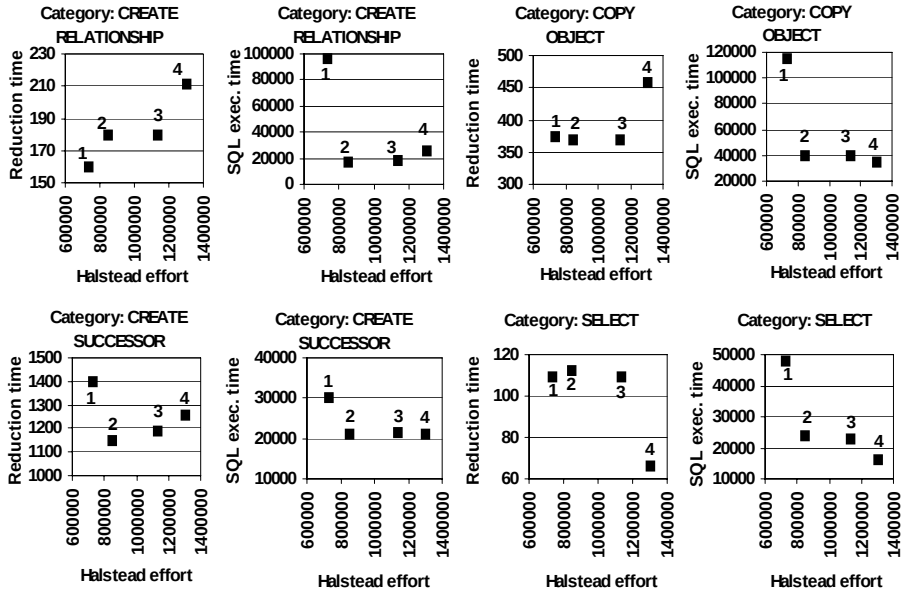

Fig. 8. Correlation of t_{red} and t_{SQL} to the Halstead effort

copy object propagate across relationships and thus manipulate the CVCs), impedance due to materializing the pin setting and the latest version comes into effect and often results in only minor differences in t_{SQL} values among *Variants 2-4*.

6 Metaprogramming Concepts

Writing metacode is different and more difficult than writing code, because the programmer has to consider a large variety of cases that may occur depending on the form of the statement and the properties defined in the DS-DDL schema.

Our key idea to developing reduction methods is the so-called *reduction polymorphism*. In OO programming languages, polymorphism supports dynamic selection of the "right" method depending on the type of object held by a reference (since the type is not known until run time, this is usually called *late binding*). In this way, it is possible to avoid disturbing conditional statements (explicit type checking by the programmer) in the code. In a similar way, we use reduction polymorphism to avoid explicit use of conditional statements in metacode. This means that for an incoming DS-DML statement, the domain-specific driver will execute reduction methods that (a) match the syntactic structure of the statement and (b) apply for the specifics of the DS-DDL schema constructs used in the statement. We illustrate both concepts using a practical example.

Suppose the following DS-DML statement.

```
1:  SELECT Cost.*
2:  FROM Offer-contains->Task-ratedCosts->Cost
3:  USE WORKSPACE Project WHERE globalId = 435532 AND Offer WHERE objectId = 122;
```

Using our DS-DDL schema from Fig. 3 and reduction *Variant 4*, the statement gets reduced to the following SQL-DML statement (`OT` denotes object table, `ATT` the attachment relationship table, `F` a floating end, and `NF` a non-floating end).

```
 1: SELECT CostOT.globalId, CostOT.objectId, CostOT.versionId, CostOT.wages, CostOT.travelExpenses,
 2:     CostOT.materialExpenses, CostOT.validUntil, CostOT.isFrozen
 3: FROM OfferOT, TaskOT, CostOT, isPartOfF_containsNF, causedByF_ratedCostsF,
 4:     ProjectOT, Project_OfferATT, Project_TaskATT, Project_CostATT
 5: WHERE OfferOT.globalId = isPartOfF_containsNF.isPartOfGlobalId
 6:     AND isPartOfF_containsNF.containsGlobalId = TaskOT.globalId
 7:     AND TaskOT.globalId = causedByF_ratedCostF.causedByGlobalId
 8:     AND causedByF_ratedCostF.ratedCostsGlobalId = CostOT.globalId
 9:     AND Project_OfferATT.projectGlobalId = 435532
10:     AND Project_OfferATT.offerGlobalId = OfferOT.globalId
11:     AND Project_TaskATT.projectGlobalId = 435532
12:     AND Project_TaskATT.taskGlobalId = TaskOT.globalId
13:     AND Project_CostATT.projectGlobalId = 435532
14:     AND Project_CostATT.costGlobalId = CostOT.globalId
15:     AND Offer.objectId = 122
```

First, any `SELECT` statement will match a very generic reduction method that will insert `SELECT` and `FROM` clauses into the SQL-DML source graph. A reduction method on the projection clause (`Cost.*`) will reduce to a projection of identifiers (*globalId*, *objectId*, and *versionId*), user-defined attributes and the flag denoting whether the version is frozen. Note that because the maximal multiplicity of the end *causedBy* pointing from *Cost* to *Task* is *1*, the table `CostOT` also contains the materialization of a pinned or latest version of some *task*, but the column for this materialization is left out in the projection, because it is irrelevant for the user. Next, a reduction method is invoked on the DS-DML `FROM` clause, which itself calls reduction methods on two DS-DML subnodes, one for each navigation step. Thus, the reduction of `Offer-contains->Task` results in conditions in lines 5–6 and the reduction of `Task-ratedCosts->Cost` results in conditions in lines 7–8. The reductions carried out in this example rely on two mechanisms, *DS-DDL schema divergence* and *source-graph divergence*.

DS-DDL schema divergence is applied in the following way. The relationship type used in the first navigation step defines only one floating end while the one used in the second navigation step defines both ends as floating. Thus in the reduction of DS-DDL, we had to map the first relationship type to two distinct tables (because relationships with only one floating end are not necessarily symmetric). Therefore, the choice of the table we use (`isPartOfF_containsNF`) is based on the direction of navigation. The situation would be even more different in case the multiplicity defined for the non-floating end would be *1*, where we would have to use a foreign key column in the object table. Another important situation where schema divergence is used in our example product line is operation propagation. To deal with DS-DDL schema divergence, each reduction method for a given node comes with a set of preconditions related to DS-DDL schema that have to be satisfied for method execution.

Source-graph divergence is applied in the following way. In *filtered navigation within a workspace*, we have to use the table `causedByF_ratedCostsF` to arrive at *costs*. The obtained versions are further filtered in lines 9, 11, and 13 to arrive only at *costs* attached to the workspace with *globalId* 435532. The situation would be different *outside a workspace*, where another table which stores the materialized *globalIds* of versions of costs that are either pinned or latest in the corresponding CVC would have to be used for the join. Thus the reduction of the second navigation step depends on

whether the clause `USE WORKSPACE` is used. To deal with source-graph divergence, each reduction method for a given node comes with a set of preconditions related to node neighborhood in the source graph that have to be satisfies for method execution.

Due to source-graph divergence, line 3 of the DS-DML statement gets reduced to lines 9–15 of the SQL-DML statement.

Obviously, it is a good choice for the developer to shift decisions due to divergence to many "very specialized" reduction methods that can be reused in diverse superordinated methods and thus abstract from both types of divergence. In this way, the subordinated methods can be explicitly invoked by the developer using generic calls and the driver itself selects the matching method. Four different APIs are available to the developer within a reduction method.

- *Source tree traversal*. This API is used to explicitly traverse the neighboring nodes to make reduction decisions not automatically captured by source-graph polymorphism. The API is automatically generated from the DS-DML metamodel.
- *DS-DDL schema traversal*. This API is used to explicitly query the DS-DDL schema to make reduction decisions not automatically captured by DS-DDL schema polymorphism. The API is automatically generated from the DS-DDL metamodel.
- *SQL-DML API*. This API is used to manipulate the SQL-DML source graphs.
- *Reduction API*. This API is used for explicit invocation of reduction methods on subordinated nodes in the DS-DML source graph.

7 Conclusion and Future Work

In this paper, we examined the topic of custom schema development and data manipulation languages which facilitate increased reuse within database-oriented software product lines. Our empirical evaluation, based on an example product line for versioning systems, shows that the portion of time required for mapping domain-specific statements to SQL at run time is below 9.9%. For this reason, we claim that domain-specific languages introduce great benefits in terms of raising the abstraction level in schema development and data queries at practically no cost.

There is a range of topics we want to focus on in our future work. Is there a way to make DS-DMLs even faster? Complex reduction methods can clearly benefit from the following ideas.

- Source graphs typically consist of an unusually large number of objects that have to be created at run time. Thus the approach could benefit from *instance pools* for objects to minimize object creation overhead.
- Caching of SQL-DML source graphs can be applied to reuse them when reducing upcoming statements.
- Would it be possible to use parameterized stored procedures to answer DS-DML statements? This makes the reduction of DS-DML statements simpler, because a statement can be reduced to a single stored procedure call. On the other hand, it makes the reduction of DS-DDL schema more complex, because stored procedures capable of answering the queries have to be prepared. We assume this approach is especially useful when many SQL-DML statements are needed to execute a DS-

DML statement. Implementing a stored procedure for a sequence of statements avoids excessive communication between (a) the domain-specific and the native driver and (b) between the native driver and the database.

- In a number of cases where a sequence of SQL-DML statements is produced as a result of reduction, these statements need not necessarily be executed sequentially. Thus developers of reduction methods should be given the possibility to explicitly mark situations where the driver could take advantage of parallel execution.

In addition, dealing with DS-DDL schemas raises two important questions.

- *DS-DDL schema evolution*. Clearly, supplementary approaches are required to deal with modifications in a DS-DDL schema which imply a number of changes in existing SQL-DDL constructs.
- *Product-line mining*. Many companies develop and market a number of systems implemented independently despite their structural and functional similarities, i.e., without the proper product-line support. Existing schemas for these systems could be mined to extract common domain-specific abstractions and possible reductions, which can afterwards be used in future development of new systems.

References

1. Apache Jakarta Project: Struts, available as: http://jakarta.apache.org/struts/
2. Ben-Natan, R., Sasson, O.: IBM San Francisco Developer's Guider, McGraw-Hill, 1999
3. Bernstein, P.A.: Repositories and Object-Oriented Databases, in: SIGMOD Record 27:1 (1998), 34-46
4. Clements, P., Northrop, L.: Software Product Lines, Addison-Wesley, 2001
5. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000
6. Frankel, D.S.: Model Driven Architecture: Applying MDA to Enterprise Computing, Wiley Publishing, 2003.
7. Halstead, M.H.: Elements of Software Science, Elsevier, 1977
8. Kießling, W., Köstler, G.: Preference SQL – Design, Implementation, Experiences, in: Proc. VLDB 2002, Hong Kong, Aug. 2002, 990-1001
9. Mahnke, W.: Towards a Modular, Object-Relational Schema Design, in: Proc. CAiSE 2002 Doctoral Consortium, Toronto, May 2002, 61-71
10. McCabe, T.J.: A Complexity Measure, in: IEEE Transactions on Software Engineering 2:4 (1976), 308-320
11. MetaCase: MetaEdit+ Product Website, available as: http://www.metacase.com/mep/
12. OMG: Common Warehouse Metamodel (CWM) Specification, Vol. 1, Oct. 2001
13. OMG: Model Driven Architecture (MDA) – A Technical Perspective, July 2001
14. Saeki, M.: Toward Automated Method Engineering: Supporting Method Assembly in CAME, presentation at EMSISE'03 workshop, Geneva, Sept. 2003
15. Simonyi, C.: The Death of Computer Languages, the Birth of Intentional Programming, Tech. Report MSR-TR-95-52, Microsoft Research, Sept. 1995
16. Thalheim, B.: Component Construction of Database Schemes, in: Proc. ER 2002, Tampere, Oct. 2002, 20-34
17. Weber, C., Kovse, J.: A Domain-Specific Language for Versioning, Jan. 2004, available as: http://wwwdvs.informatik.uni-kl.de/agdbis/staff/Kovse/DSVers/DSVers.pdf