DSL-DIA - An Environment for Domain-Specific Languages for Database-Intensive Applications

Jernej Kovse and Theo Härder

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

Abstract. This paper presents DSL-DIA, an environment that lets a system-family vendor define a metamodel for a custom domain-specific language used by customers for specifying properties of family members. Once the metamodel is imported in the environment, the environment allows the customer a flexible way to program in the domain-specific language and translates obtained programs to implementations of family members. In our case, family members are always database-intensive applications with application logic executed in the database server.

1 Introduction

The most important features of the new generation of object-relational database systems depicted by the recent SQL:1999 [1] standard and its upcoming successor SQL:2003 are the possibility of executing application logic in the database server and using object-relational extensions with the relational data model. In particular, *user-defined routines* (UDRs) enable the manipulation of data in the database in a language native to the database system. Complex *user-defined types* (UDTs) can be used to structure multiple data fields and afterwards be used as column or table types. *Triggers* define SQL statements that get executed when a trigger event takes place. Finally, semantic integrity of data can be enforced by *constraints* and *assertions*. By using these concepts, engineering of self-contained applications that run completely in the database server is made possible.

Recently, the area of *software product lines* [4] has gained a lot of research attention. The term refers to a group of software systems sharing a common set of features that satisfy the needs of a particular market. It is cost-effective if the product line is implemented as a *system family*, meaning that the systems in the product line (family members) can be built from a common set of implementation assets. *Domain engineering* is the key enabling approach for designing and implementing system families.

When provided with a system family, the user has to somehow specify the concrete functionality of a family member he or she wishes to obtain. In an ideal case, by using a common set of reusable assets (e.g., components, classes, or code templates), the member can be automatically generated from this specification. A possible way to

write a specification is to use a *domain-specific language* (DSL) that contains abstractions capable of describing the member within the domain of its family. Our DSL-DIA (Domain-Specific Languages for Database Intensive Applications) environment, presented in this paper, allows system-family vendors easy definition of DSLs for their system families. Once a DSL is defined, the environment supports highly intuitive programming in this DSL. The environment translates a DSL program to a primitive set of SQL:1999 constructs, normally used in database-intensive applications, e.g. UDTs, table definitions, UDRs, triggers and constraints.

Sect. 2 of this paper will describe the DSL-DIA environment in detail. Sect. 3 illustrates the use of the environment on a practical example, while Sect. 4 gives an overview of related work. In Sect. 5, we make a conclusion and discuss some ideas for the future work related to the approach.

2 Using the DSL-DIA Environment

The DSL-DIA environment is used as illustrated in Fig. 1. First, the product-line vendor defines a *DSL metamodel*, which is a MOF-based [9] metamodel, to describe the constructs that can appear in DSL programs. There are two ways the product line customer can enter a *DSL program*: The customer can instantiate DSL metamodel constructs to obtain a *DSL model*, which is represented in a tree-like form, called the *DSL tree*. Alternatively, the user may enter a DSL program in textual form using a *DSL editor*. There is a one-to-one mapping between the DSL model and the DSL program, so the changes to the model made in the DSL tree affect the DSL program displayed in the editor and vice versa. To enable this mapping, the vendor has to specify *DSL rendering rules*, which define how an instance in the DSL model will be represented in the DSL program.

DSL programs themselves are not executable and have to be translated to a set of primitive constructs to obtain a corresponding *SQL schema* with a set of UDTs, table definitions, or UDRs for the database-intensive application. In the same manner as DSL programs have their model equivalents in DSL models, we want to have SQL schemas also represented as models in order to be able to define the translation on the

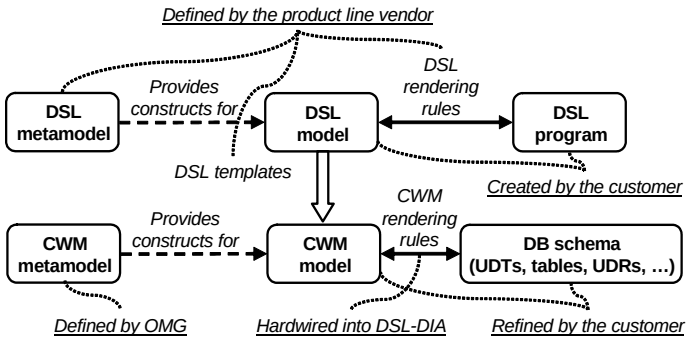


Fig. 1. Using the DSL-DIA environment

model-to-model basis. OMG's Common Warehouse Metamodel (CWM) [10] is a metamodel dedicated to easy interchange of business intelligence metadata between warehouse tools, warehouse platforms, and warehouse metadata repositories. CWM's package `Relational` defines modeling constructs that appear in database schemas of object-relational databases. For this reason, we choose to represent the schema of the obtained database-intensive application - the product family member whose properties were described in the DSL model - as a CWM model.

To support the translation, *DSL templates* that map the DSL model to the CWM model are defined by the product line vendor. Similar to DSL rendering rules, there are *CWM rendering rules* which render the obtained CWM model (represented as the CWM tree) to textual representation of the database schema in SQL (displayed in the SQL editor). If desired, the customer can enhance the obtained schema with custom functionality that could not be expressed in the DSL. This may be done either by manipulating the CWM tree or modifying the schema in the SQL editor. This process corresponds to Frankel's [5] description of partial round-trip engineering in model-driven development, where it is allowed to enhance the generated artifacts with implemetation parts that could not be sufficiently described at the specification level.

3 Case Study: A DSL for Version Management

Repository systems [2] are generally used for managing data in team-oriented engineering activities. Version management provided by a repository system will encompass functions for representing versions of engineered artifacts and combining these versions into configurations. Since the ACID transaction model proves inappropriate for longlife (design) transactions, version management supports locking of versions in a configuration via checkout and checkin operations [3]. In an object-oriented repository, versioned and unversioned artifacts are represented as *repository objects*. Each repository object has a *repository object type*. Repository object types can be associated by *relationship types* (which, in our example, are always binary). A *repository relationship* is an instance of a relationship type and denotes a semantic connection among two repository objects. Repository object types and repository relationship types are defined by a repository *information model*. A repository system implemented as a database-intensive application will attempt to provide its operations as UDRs and structure its data as UDTs.

Version management is highly variable! This leads us (the vendor) to the idea to provide a product line for repository systems, where customers will have the possibility to customize versioning semantics for their repositories. In a very simplified scenario, starting from some initial information model (defined by the customer), the customer has the following customization options.

- A given repository object type may or may not support versioning.
- If versioning is supported, the customer wants to specify the permitted number of successor versions to a given object version.
- The customer wants to have the possibility to define own configuration types and choose the types of repository objects these configuration types may contain. A

configuration is a special type of a repository object, since only one version of a given repository objects may be present in a configuration at a time.

- For a given relationship type, the customer wants to decide whether or not the *attach* operation (which attaches an object to a configuration) will propagate across the relationships of this type.
- For a given relationship type, the customer wants to decide whether or not the operations *createSuccessor* (which creates a successor version to a given version), *freeze* (which makes a version immutable), *checkout*, *checkin*, *copy* and *new* (which creates a new object instance) propagate across the relationships of this type.

In accordance with the above domain analysis, the product line vendor will construct the DSL metamodel illustrated in Fig. 2. As this DSL metamodel is imported in the DSL-DIA environment, users can create DSL models that conform to the metamodel. The environment displays these models as trees, where instances of metamodel classes are represented as tree nodes. Generally, DSL models can contain cycles, which are impossible to represent in tree-like structures where each node has exactly one parent. To overcome this problem, certain nodes are equipped with hyperlink-like pointers that enable the user to navigate within the model graph without the origin and source node of the navigation being directly connected in the DSL tree.

Suppose a customer requires a repository system used for an OO development environment that stores implementations of classes. Since the developed system stores persistent data in the database, some classes access database tables (a class can access zero or many tables and a table can be accessed by zero or many classes). We expect that the development path for the system will be mirrored by successive versions of classes and tables. Because of semantic dependencies, we require that at the event of freezing a table version, all related class versions are frozen as well. In terms of the

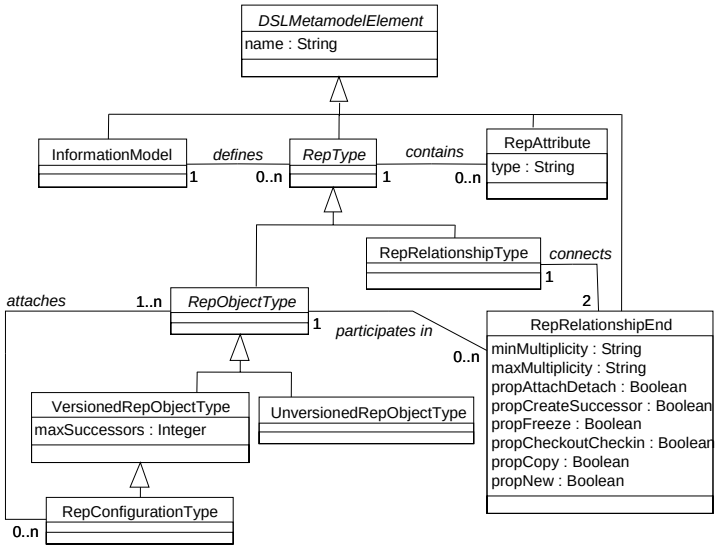


Fig. 2. DSL metamodel for version management

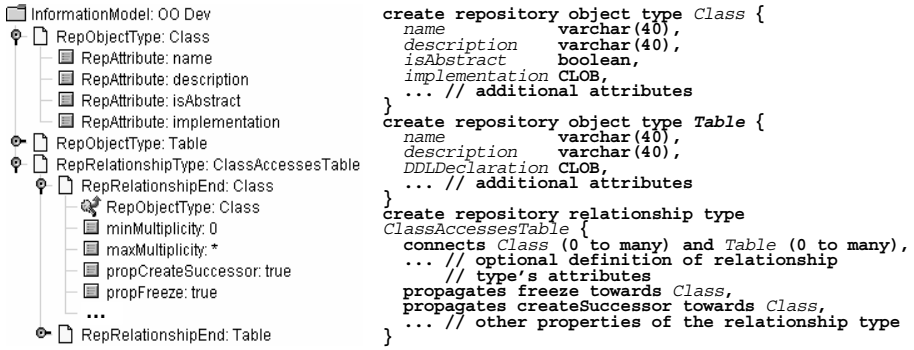
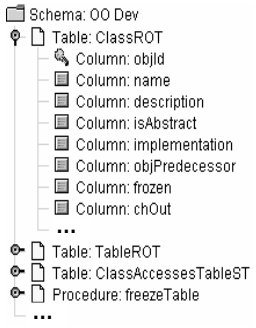


Fig. 3. A sample DSL tree and DSL program

DSL metamodel, there will be a `RepRelationshipType` *ClassAccessesTable* whose `RepRelationshipEnd` attached to *Class* will propagate the *freeze* operation. Portions from the DSL tree and DSL program are illustrated in Fig. 3.

In the translation phase, a CWM model is generated from the DSL model. This process requires reusable CWM model parts that are generic, meaning that such a part acts as a blueprint capable of producing different CWM models for different DSL models but at the same time captures the commonalities among CWM models that exist within the domain of the system family. A feasible way to implement such generic parts is by using templates. The outcome of the application of a DSL template should be a CWM model, which we choose to express using XML Metadata Interchange (XMI) [11]. Thus, within a DSL template, commonalities can be expressed using standard CWM XMI tags. However, to support variation among CWM models that can be generated from the template, the template has to support *placeholders* (for filling places in the template with user-defined values), *repetition* and *conditional statements*. We have defined X-CWM DTD by extending the CWM XMI DTD with tags used to express these concepts. A template is then expressed as a X-CWM document that gets processed by an XSLT template to produce a CWM model expressed in XMI. This approach is a variation of generic model-to-model transformations described in [8].

For example, for the case of the version management DSL, a template will generate repository database tables from repository object type definitions. These tables retain fields for the attributes specified for the type in the DSL metamodel and acquire additional fields used for version management, e.g. `objId` primary key for storing repository object's identity, `objPredecessor` foreign key for enabling relationships to a predecessor version, `frozen` field of type boolean to denote whether a version is frozen, `chOut` foreign key for enabling relationships to the configuration the version is currently checked out to, and others. Tables generated from repository object type definitions are called repository object tables (ROTs). Additional tables, e.g. those generated from relationship types of multiplicity many-to-many are called supplementary tables (STs). A portion from the CWM tree and the obtained stored procedure for freezing table versions, which propagates the freeze operation across the table's relationship to classes is illustrated in Fig. 4.




```
CREATE PROCEDURE freezeTable(tableIdArg INT8)
  DEFINE checkedOut   INT8;
  DEFINE isFrozen     BOOLEAN;
  DEFINE currentClass INT8;
  FOREACH tableCursor FOR SELECT frozen, chOut
                          INTO isFrozen, checkedOut
                          FROM TableROT WHERE objId = tableIdArg
    IF chOut IS NOT NULL THEN
      RAISE EXCEPTION 9999, 0,
        'Can not freeze a checked out repository object.';
    ELIF isFrozen = 'f' THEN
      UPDATE TableROT SET frozen = 't' WHERE CURRENT OF tableCursor;
      FOREACH SELECT classId INTO currentClass
              FROM ClassAccessesTableST WHERE tableId = tableIdArg
              EXECUTE PROCEDURE freezeClass(currentClass);
      END FOREACH;
    END IF;
  END FOREACH;
END PROCEDURE;
```

Fig. 4. A sample CWM tree and generated user-defined procedure

4 Related Work

The use of domain-specific languages for formally specifying a member of a system family is outlined by Czarnecki and Eisenecker [4]. The authors call such a DSL a *configuration DSL* and note that the language can be derived from the feature model [7] constructed in the domain analysis phase. As in our case, the authors emphasize the need for a translator, which translates a program in a configuration DSL into a concrete configuration of implementation components. In our case, the translator performs a model-to-model mapping, since both the initial specification (the DSL program) and the implementation of the family member are represented as models.

The notion of extensible programming environments, where programmers can define and use custom high-level abstractions, which get translated to a program in a general-purpose programming language, such as C, is materialized in Simonyi's work on Intentional Programming (IP) [12]. The IP environment allows the programmer to define rendering methods (which determine how new abstractions will be displayed), editing methods (which determine how abstractions will be entered), reduction methods (which determine how programs using abstractions will be translated to a general-purpose programming language), and others.

Automatic generation of program code artifacts from models is the goal of OMG's Model Driven Architecture (MDA) [11]. Although formal UML models, expressed using Executable UML [6], often prove appropriate as MDA's platform-independent models (PIMs), a major drawback of this approach is that these models are too verbose, since Executable UML is kept as general as possible so that it can be used for a wide variety of different domains. A possible solution to this problem is to use a UML profile as a lightweight extension of the UML metamodel and define domain-specific abstractions via *stereotypes* and *tagged values*. Thus, in the context of MDA, it is possible to consider our DSL metamodels as UML domain profiles, which are equipped with DSL rendering rules (to display DSL programs) and DSL templates that assure a domain-specific mapping to the implementation via the CWM metamodel.

5 Conclusion and Future Work

DSL-DIA provides customers within a given software system domain with an intuitive way to specify the properties of system-family members using a set of orthogonal language abstractions provided via a metamodel. Using a set of high-level abstractions for system specification alleviates the design of generated database-intensive applications, since the properties of the desired system are expressed in a brief and straightforward form, shielding the customer from implementation level details that arrive in the implementation via template-based model translation.

In our future work, we attempt to focus on the following topics.

- *Maintenance of DSL templates*: As the complexity of the domain increases, product-line vendors are faced with large DSL templates that are difficult to implement and maintain. In our opinion, a special technique alleviating DSL template development should supplement DSL-DIA.
- *Integrated development environments (IDEs) for product line vendors*: DSL-DIA requires representing an appropriate set of high-level abstractions via a metamodel, defining DSL rendering rules, DSL templates and testing the DSL prior to releasing it into customer use. We will explore the implementation of a dedicated IDE for product-line vendors with support for feature-oriented domain analysis, a DSL metamodel repository, and automated testing facilities.

References

1. ANSI/ISO/IEC 9075-2:1999. Information Technology - Database Languages - SQL - Part 2: Foundation (SQL/Foundation), 1999.
2. Bernstein, P.A.: Repositories and Object-Oriented Databases, in: ACM SIGMOD Record 27:1, 1998, pp. 34-46.
3. Bernstein, P.A.: Design Transactions and Serializability, in: Proc. 7th Int. Workshop on High Performance Transaction Systems (HPTS 1997), Pacific Grove, Sept. 1997.
4. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000.
5. Frankel, D.S.: Model Driven Architecture: Applying MDA to Enterprise Computing, Wiley Publishing, 2003.
6. Mellor, S.J., Balcer, M.: Executable UML, Addison-Wesley, 2002.
7. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study, Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Nov. 1990.
8. Kovse, J., Härder, T.: Generic XMI-Based UML Model Transformations, in: Proc. OOIS 2002, Montpellier, Sept. 2002, pp. 192-197.
9. OMG: Meta Object Facility (MOF) Specification, Vers. 1.4, April 2002.
10. OMG: Common Warehouse Metamodel (CWM) Specification, Vol. 1, Vers. 1.0, Oct. 2001.
11. OMG: Model Driven Architecture (MDA), Draft Document, July 2001.
12. Simonyi, C.: The Death of Computer Languages, the Birth of Intentional Programming, Tech. Report MSR-TR-95-52, Microsoft Research, Sept. 1995.