Enforcing Modeling Guidelines in an ORDBMS-based UML-Repository

N.Ritter, H.-P. Steiert
University of Kaiserslautern, Dept. of Computer Science
P. O. Box 3049, D-67663 Kaiserslautern, Germany
{ritter,steiert}@informatik.uni-kl.de

Abstract

Due to its rich set of modeling concepts and its broad application spectrum the Unified Modeling Language (UML) has become widely accepted for modeling many aspects of software systems. Since UML is not related to any particular design method, each software development project has to establish its own modeling guidelines. Hence, tool support is needed for guiding the developer throughout the modeling process and for enforcing project-related integrity of UML models. In this paper, we present our approach for enforcing guidelines in UML-based software development processes. For managing UML models, we implemented a UML repository on top of an object-relational database management system (ORDBMS). Guidelines are expressed as OCL constraints and are enforced either automatically, i. e., by the UML repository, or on user demand. For this purpose, we take advantage of ORDBMS query facilities for checking guidelines by automated mapping of OCL constraints to SQL expressions.

Keywords: UML, OCL, Modeling Guidelines, Repository, Object-Relational Database Systems

1 Introduction

In our SENSOR project[i] we consider new object-relational database technology for software systems which have to support data management tasks in software engineering projects. In detail we aim at two goals. First, we are developing a shared UML repository (UML, Unified Modeling Language, [4][5][12]) based on an object-relational database management system (ORDBMS) [11] in order to support cooperation of developers and reuse of design. Second, we want to generate database schemas and object-oriented database application programming interfaces (API) for engineering applications from a graphically specified UML model. This paper deals with some important aspects of our UML repository: exploiting OCL constraints for preserving consistency of UML models and for enforcing guidelines during the modeling process.

UML is becoming a de-facto standard for object-oriented modeling. The Object Management Group (OMG) has adopted the UML into its Object Management Architecture (OMA). Furthermore, UML has become broadly supported by the vendors of graphical modeling tools. In comparison to other information models, e. g., the Entity-Relationship-model [1], UML has lots of advantages. It is object-oriented and object-orientation has become the leading software development technology. Also, UML offers a large set of structural modeling elements including class structures and several options to define the semantics of relationships. In addition, it comes along with modeling elements for describing the behaviour of a system and for state-oriented aspects. The OCL (Object Constraint Language, [6][13]) enables developers to specify constraints in a descriptive manner. Unfortunately, OCL is only weakly represented in many descriptions of UML [3]. In this paper we will focus on the use of OCL for our purposes.

Our implementation of a UML repository is based on the UML meta-model and is implemented by exploiting an ORDBMS. The enhanced type system, the powerful SQL facilities and the extensibility features of ORDBMSs have proven to be very helpful for our purposes. The repository manages UML models. The implementation of the UML repository, i. e., the mapping of the UML meta-model to an object-relational database schema, is described in Sect. 2. In this section, we also outline how OCL constraints can be mapped to

SQL constraints. The usage of OCL in our project is not limited to global integrity constraints. We also exploit OCL for enforcing project-related design guidelines. A short classification of guidelines with examples and their implementation as SQL constraints is given in Sect. 3. The sample constraints illustrated in Sect. 2 and Sect. 3 are manually mapped from OCL to SQL. In order to provide adequate tool support for our approach we developed an OCL-to-SQL compiler for automatic mapping of OCL constraints to SQL. This compiler is outlined in Sect. 4. Sect. 5 concludes the paper.

2 The UML Repository

As mentioned before one of our research goals is to find out whether or not ORDBMSs provide adequate mechanisms for managing engineering data. As a sample application we are developing a UML repository based on an ORDBMS. Managing UML models designed as part of the software development process within a UML repository has several advantages. First, a shared database eases cooperation of developers involved in the development process. Second, the repository serves as a basis for the reuse of design decisions documented as UML models. Third, higher software quality can be achieved by analyzing UML models. This way design errors can be detected early and design guidelines can be enforced. Query facilities provided by ORDBMSs seem to be helpful for this (analyzing) task. Fourth, UML models can be used to generate database schemas and APIs [8][10].

In [5], UML itself is used to describe the UML meta-model. Since the graphical modeling elements are not powerful enough to completely determine the semantics of UML, additional invariants are used. These invariants are expressed in OCL, which is a descriptive object-oriented constraint language. A textual comment in a natural language completes the specification of UML.

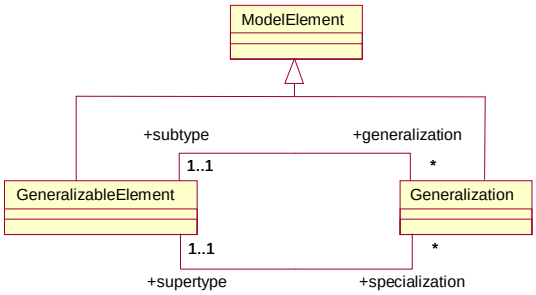

Figure 1: UML class diagram: ‘Generalization Hierarchy’

Our UML repository is based on the UML meta-model [5] and is implemented by exploiting an ORDBMS [11]. We have mapped the UML meta-model to an object-relational database schema. In order to enforce data integrity in the UML repository we have implemented the invariants as SQL constraints. The powerful SQL facilities and the extensibility features of ORDBMS have proven to be very helpful for these purposes. The current implementation only supports manipulating UML models via the SQL interface, but we intend to additionally provide an API which is compliant to the UML CORBAfacility Interface Definition [7].

Mapping the Meta-Model

Due to space restrictions, we cannot describe the features of ORDBMS in detail in this paper. Nethertheless, a short introduction into the object-relational data model and the extensibility features of ORDBMS is essential for a deeper understanding. In [11] Stonebraker

reclaims an ORDBMS to provide at least user-defined types (UDT), user-defined routines (UDR), a rule system and inheritance hierarchies for types and tables. The implementation of our UML repository exploits all these features.

Figure 1 shows a simplified excerpt of the UML meta-model, which will server us as an example for demonstrating the principles of our approach (throughout this paper we use the SQL dialect of the ORDBMS 'Informix Dynamic Server'). Instances of the class 'GeneralizalbeElement' represent modeling elements which are able to participate in generalization relationships. The relationships themselves are represented by instances of the class 'Generalization'. In the following we outline how these structures can be mapped to an object-relational database schema.

In a first step each class of the UML meta-model is mapped to a user-defined type. We exploit the type hierarchies provided by the ORDBMS in order to implement the inheritance relationships in the UML model. This results in the following ROW TYPES:

```
CREATE ROW TYPE model_element_ty
(   id              oid_ty,
    name            name_ty         );

CREATE ROW TYPE generalizable_element_ty
(   is_root         BOOLEAN,
    is_leaf         BOOLEAN,
    is_abstract     BOOLEAN
) UNDER model_element_ty;

CREATE ROW TYPE generaliztion_ty
(   discriminatorname_ty,
    subtype             oid_ty,
    supertype           oid_ty
) UNDER model_element_ty;
```

Each ROW TYPE has an additional attribute 'id', which does not stem from the UML meta-model. Values of this attribute uniquely identify instances in the database. Its type, 'oid_ty', is not a build-in type, but an user-defined OPAQUE TYPE. In contrast to ROW TYPEs, the internal representation of an OPAQUE TYPE is hidden. A value of this type is not only unique in the whole database, it also contains additional information. First, the name of the table used for storing the instance and, second, the name of the type 'oid_ty' is contained. User defined functions provide access to both.

Although references are included in the standard SQL:1999 [2], the commercial ORDBMS used does not support references. Hence, the relationships between the classes are implemented by foreign keys, i. e., the attributes 'subtype' and 'supertype' of 'generalization_ty'.

In an ORDBMS an instance of a ROW TYPE can not live for itself. It has to be stored in a table. Therefore, each ROW TYPE is associated with a corresponding database table. ORDBMSs also support inheritance relationships among tables. Hence, the type hierarchy is reflected by the following table hierarchy:

```
CREATE TABLE model_element_ta OF TYPE model_element_ty
(   PRIMARY KEY( id )   );

CREATE TABLE generalizable_element_ta OF TYPE generalizable_element_ty
(   PRIMARY KEY( id )   ) UNDER model_element_ta;
```

```
CREATE TABLE generaliztion_ta OF TYPE generalization_ty
(   PRIMARY KEY ( id ),
    FOREIGN KEY ( subtype ) REFERENCES generalizable_element_ta ( id ),
    FOREIGN KEY ( supertype ) REFERENCES generalizable_element_ta ( id )
) UNDER model_element_ta;
```

The attribute ‘id’ is used as a primary key. In order to assign a correct value to this attribute for each row, we exploit the rule system of the ORDBMS. If a new instance is inserted or an existing instance is modified, then a trigger is executed. This trigger assigns a new identifier to attribute ‘id’ if its value is NULL, otherwise it checks whether or not the value is correct. Additionally, the foreign key constraints enforce referential integrity.

Invariants

In addition, the OCL [6] invariants defined in the UML meta-model are mapped to SQL constraints (more precisely, we map OCL constraints to SQL predicates, which can be used in SQL constraints, triggers, and WHERE clauses). Hence, we preserve the consistency of UML models managed by the repository. In the following, the mapping of OCL constraints is demonstrated by two examples.

In [5], an OCL constraint is given defining the semantics of the attribute ‘isRoot’ of class ‘GeneralizableElement’. If the value of this attribute is ‘true’ for any instance of ‘GeneralizableElement’, the instance must not have any relationship to an instance of ‘GeneralizableElement’ in role ‘generalization’. These semantics is precisely represented by the following OCL constraint:

```
context GeneralizableElement inv:
    self.isRoot implies self.generalization->isEmpty;
```

Using an ORDBMS for storing UML models enables us to exploit its query facilities for the evaluation of OCL constraints. The OCL constraint above results in the following SQL check-constraint:

```
CHECK NOT EXISTS
    (   SELECT *
        FROM generalizable_element_ta AS t1
        WHERE NOT ( 0 = (   SELECT count(*)
                            FROM generalization_ta AS t2
                            WHERE t1.id = t2.subtype ) ) );
```

Unfortunately, not all constraints are as simple as this one. Often, OCL constraints include complex computations. For example, the operation ‘allSupertypes’ of class ‘GeneralizableElement’ computes the transitive closure of all supertypes of a given instance of ‘GeneralizableElement’. In order to map the following OCL constraint to an SQL constraint, it is required to previously map the operation ‘allSupertypes’ to a UDR.

```
context GeneralizableElement inv:
        self.isRoot implies self.generalization->isEmpty;
```

The UDR ‘allSupertypes’ (signature see below) can be implemented either in Java, C, or a proprietary stored-procedure language and registered in the database.

```
CREATE PROCEDURE all_supertypes ( generalizable_element ge )
    RETURNING SET( generalizable_element_ty NOT NULL );
```

In order to map the previous OCL constraint given above ‘all_supertypes’ can be used as follows:

```
CHECK NOT EXISTS
        (   SELECT *
            FROM generalizable_element_ta AS t1
            WHERE NOT ( t1 IN all_supertypes( t1 ) );
```

The discussions of this section clarify that in order to capture the semantics of the UML meta-model completely, OCL constraints can be mapped to SQL. In the following sections, we will see that these mapping mechanisms can also be used for other purposes further supporting the modeling process with UML.

3 Enforcing Design Guidelines

In the previous section, we outlined how we map the well-formedness rules to SQL. Now we want to detail this discussion by considering the objectives OCL can contribute to achieve. Exploitation of constraints in the UML repository is not limited to enforcing the class invariants specified in [5]. Furthermore, constraints are a helpful support for guiding the developers through the process of modeling [9]. We intend to exploit OCL constraints for the following purposes:

- ***Design Guidelines***

  Design guidelines are additional constraints on UML models. Hence, the repository enforces that only valid UML models are stored, i. e., UML models which fulfil both, the invariants and the guidelines. We distinguish two kinds of design guidelines:

  Global Design Guidelines

  Global design guidelines hold throughout the entire process of modeling. In contrast to the invariants specified in [5], they are strongly related to a particular project.

  As an example, assume that your team is using Java which does not support multiple inheritance. Thus, a global design guideline is supposed to control those UML models which, mapped to Java, do not exploit multiple inheritance. Such a guideline is strongly related to Java projects. It may be directly expressed as an OCL constraint, restricting the number of superclasses for each specified class to at most one:

  ```
  context: GeneralizableElement inv:
    self.generalization->size <= 1
  ```

  The resulting SQL constraint is given below:

  ```
  CHECK NOT EXISTS (  SELECT *
                      FROM generalizable_element_ta ge
                      WHERE NOT ( 1 >= ( SELECT count(*)
                                         FROM generalization_ta g
                                         WHERE ge.id = g.subtype ) ) )
  ```

  Temporary Design Guidelines

  The use of global design guidelines may be too restrictive in some cases. Often, in early modeling phases the guidelines should be less restrictive than in the final phases.

  For example, object-oriented programming languages (OOPL) like Java and Smalltalk do not directly support n-ary associations, because relationships are expressed through references or collections of references. Usually, such associations are implemented by an

additional class connecting the associated classes. In early analysis phases, such n-ary associations may be helpful. In later phases of the development process, however, it is more reasonable to have classes allowing a straight implementation. The following constraint can be added to the invariants and design guidelines if avoidance of any n-ary associations is wanted:

```
context Association inv:
    self.connection ->size = 2
```

This constraint restricts the amount of instances of 'AssociationEnd' connected to one instance of 'Association' to exactly two. In the UML repository it may be implemented by a check constraint:

```
CHECK NOT EXISTS (   SELECT *
                     FROM association_ta a
                     WHERE NOT( 2 = (   SELECT count(*)
                                        FROM association_end_ta ae
                                        WHERE ae.association = a.id   ) ) )
```

- ***Process-related Design Rules***

While design guidelines are strongly related to the UML models stored in the UML repository, process-related design rules are used to control the modeling process itself. We intend to exploit OCL for design rules in the following ways:

Pre- & Postconditions

These rules are related to operations. Preconditions describe the states of a system, in which the execution of an operation is allowed. By the same token, postconditions describe states correct after execution.

OCL supports pre- and postconditions for operations, but it is not possible to refer to the before-state of an operation in the specification of a postcondition. Preconditions may involve the parameters of the operation and postconditions may involve the result. Therefore, checking pre- and postconditions in the repository is similar to checking guidelines with the exception that the SQL constraints may include parameters. A sample mapping is given below.

```
context GeneralizableElement::remove_generalization( Generalization gen )
    pre: self->generalization->includes( gen )
```

```
CHECK NOT EXISTS (   SELECT *
                     FROM generalizable_element_ta AS ge
                     WHERE $gen IN (  SELECT subtype
                                      FROM generalization_ta AS g
                                      WHERE g.subtype = $id) )
```

This precondition is associated with 'GeneralizableElement::remove_generalization' which removes an generalization relationship for a given instance of class 'GeneralizableElement'. It checks whether or not the relationship exists. The term '$id' in the SQL statement is a variable which needs to be bound to the value of the identifier of the object the operation is applied to. Additionally, the variable '$gen' has to be bound to the identifier of the generalization relationship to be removed.

Design Goals

Design goals are related to long-term activities. They are used to describe the intended final state of a design activity. In contrast to guidelines and pre- and postconditions, which are enforced by the system, the developer itself may decide when to examine a design goal. Thus, checking constraints on demand is supported in our approach.

Following our approach, OCL can serve as a powerful tool for enforcing the preciseness of UML models in general, for enforcing design guidelines and for guiding the developer through the process of modeling.

4 Mapping OCL constraints

In order to exploit OCL constraints as described so far it is not suitable to convert the constraints to SQL manually. Especially regarding that guidelines evolve over time, which is reflected by the temporary design guidelines, and that developers may want to check design goals in an ad-hoc manner, manual mapping is not acceptable, because the work of translating the constraints is exhausting and error-prone. Additionally, developers would need to be experts not only in UML and OCL but also for SQL and ORDBMS functionality. In order to avoid these complexities we want to support automatic mapping of OCL constraints (expressing invariants, design guidelines, etc.) by an OCL-to-SQL compiler. This compiler works as follows.

After parsing the OCL constraint, the compiler generates an intermediate representation, called ***translation graph***. The UML object diagram shown in Figure 2 illustrates a sample translation graph. It results from parsing the design guideline introduced in Sect. 3, which restricts the upper bound of superclasses for each class in a UML model to at most one.

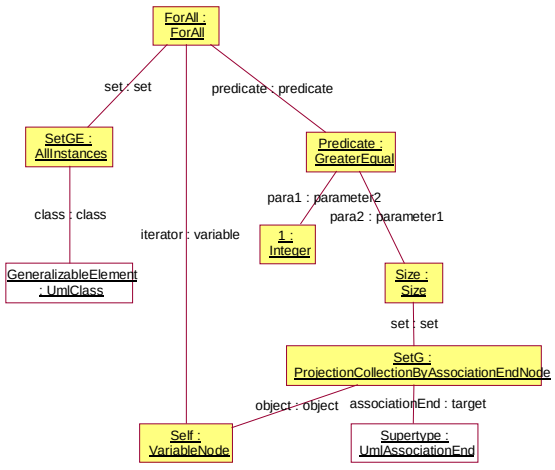

Figure 2: UML object diagram: ‘Sample Translation Tree’

The translation graph consists of two kinds of nodes, translation nodes and meta-data nodes. In our example, the objects ‘ForAll’, ‘SetGE’, ‘Size’, ‘SetG’, ‘Self’ and ‘1’ are translation nodes (dark coloured). Translation nodes implement the code generation algorithm, which depends on information about the UML model and its mapping to the database schema. Meta-data nodes (‘GeneralizableElement’ and ‘Supertype’ (light coloured)) provide this information. These nodes represent instances of the classes of the UML meta-model.

The classes for building and representing this translation graph are illustrated by the UML class diagram shown in Figure 3. Of course, this is not the whole UML model for translation

graphs but it suffices for explanation. For example, an instance of class ‘ForAll’ has to be connected to instances of ‘SetNode’, ‘VariableNode’ and ‘PredicateNode’. In the sample translation graph (see UML object diagram (Figure 2)) the node ‘ForAll’ is an instance of class ‘ForAll’. It is connected to an instance of class ‘AllInstances’, named ‘SetGE’. This relationship is valid because class ‘AllInstances’ is subclass of abstract class ‘SetNode’. For the same reasons, the objects ‘Predicate’ and ‘Self’, instances of the classes ‘GreaterEqual’ and ‘Variable’, are connected to object ‘ForAll’. Also, the node ‘SetGE’ has to be connected with an instance of class ‘UmlClass’. This is reflected by the relationship to ‘GeneralizableElement’, an instance of class ‘UmlClass’, representing class ‘GeneralizableElement’ in the UML meta-model. This instance provides the meta-data needed by ‘SetGE’ in the translation process.

The translation process is based on SQL templates and construction rules, implemented by the classes associated with the translation nodes. The following SQL template is associated

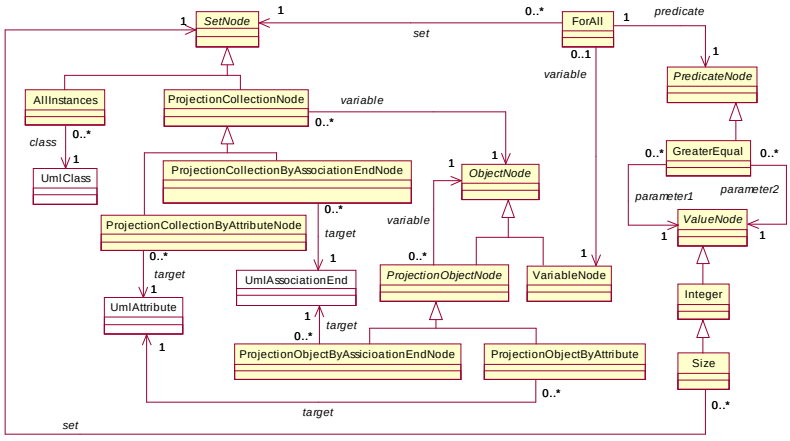

Figure 3: UML class diagram ‘Translation Graph Nodes’

with the ‘ForAll’ node:

```
NOT EXISTS(  SELECT *
             FROM ( $SetNode$ ) AS $VariableNode$
             WHERE NOT ( $PredicateNode$ ) )
```

SQL templates may contain generic parameters, enclosed by ‘$’. As part of the translation process, each node asks its subnodes to provide an appropriate SQL fragment in order to replace the generic parameters. In the sample translation graph above, the object ‘ForAll’ receives three SQL fragments from its subnodes ‘SetGE’, ‘Self’ and ‘Predicate’.

The ‘SetGE’ node depends on meta-data in order to deliver an appropriate SQL fragment. It needs the name of the database table in which the tuples representing instances of the class are stored. In our example, this name is provided by an instance of meta-class ‘UmlClass’ representing the meta-data of the class named ‘GeneralizableElement’. Hence, parameter $tablename$ in the SQL template below is replaced by ‘generalizable_element_ta’. The resulting SQL fragment is used to replace parameter $SetNode$ in the SQL template above.

```
SELECT *
FROM $tablename$
```

Another interesting kind of nodes are the projection nodes. We have identified four kinds of projection nodes, ‘ProjectObjectByAttribute’, ‘ProjectObjectByAssociationEnd’, ‘ProjectCollectionByAttribute’ and ‘ProjectCollectionByAssoziationEnd’. These nodes are related to the two different ways of accessing structural features in OCL (either by accessing an attribute or by navigating along an association) and the two different kinds of results (either a single object or a collection of objects). The translation graph in our example contains an instance of class ‘ProjectCollectionByAssociationEnd’, because the evaluation of the sample constraints needs to navigate from class ‘GeneralizableElement’ to the class ‘Generalization’ and the expected result is a collection of instances of class ‘Generalization’. If asked by node ‘Size’ to provide an SQL fragment it has to complete one of the following templates:

```
SELECT T2.*
FROM $sourcetablename AS T1, $targettablename$ AS T2
WHERE T1.$foreignkey$ = T2.id AND T1.id = $object$.id

SELECT T2.*
FROM $sourcetablename$ AS T1, $targettablename$ AS T2
WHERE T1.id = T2.$foreignkey$ AND T1.id = $object$.id

SELECT T3.*
FROM $sourcetablename$ AS T1, $relationtablename$ AS T2, $targettablename$ AS T2
WHERE  T1.id = T2.$sourceforeignkey$
       AND T2.$targetforeignkey$ = T3.id
       AND T1.id = $object$.id
```

The first template is related to a (1:1)-relationship, the second to a (1:n)-relationship and the last is related to an (n:m)-relationship. The projection node chooses the template by evaluating the meta-data provided by object ‘supertype’. In our example, ‘SetG’ chooses the second template. Parameter $sourcetablename$ is replaced by ‘generalization_element_ta’, $targettablename$ by ‘generalization_ta’, $foreignkey$ by ‘subtype’ and $object$ by ‘self’.

After processing all translation graph nodes as outlined in this section we obtain the following SQL search expression:

```
NOT EXISTS
(   SELECT * FROM generalizable_element_ta AS self
    WHERE NOT
    (   1 <= (     SELECT COUNT(*)
                   FROM TABLE ( MULTISET (
                            (   SELECT T2.*
                                FROM generalizable_element_ta AS T1, generallization AS T2
                                WHERE T1.id = T2.subtype AND T1.id = self.id ) ) ) ) ) )
```

Compared to the SQL constraint presented in Sect. 3 which resulted from a manual mapping of the same OCL constraint, the SQL expression resulting from the algorithm discussed in this section is much more complex. So far, we did not consider any optimization and performance issues, but the SQL constraints created by our compiler seem to be a challenge for every DBMS optimizer. Thus, generation of more efficient SQL constraints will be a major issue of future work.

We have to admit that there are some OCL constructs which are difficult to map, e. g., the generic iterator operator. So far, we do not allow to use such operators in OCL constraints. However, we think that the extensibility features of ORDBMS will help us to fix this problem.

5 Conclusions

In this paper, we have reported on our UML repository, which is based on the UML meta-model and manages UML models. The UML repository is implemented by using an ORDBMS. We have taken advantage of the enhanced type system in order to map the UML meta-model to a database schema. Additionally, the examples presented throughout the paper show that the powerful query facilities of an ORDBMS are very helpful for the mapping of OCL constraints to SQL constraints. In contrast to a hard-coded implementation of the class invariants in modeling tools, our approach is much more flexible. It allows to adapt the invariants to the needs of a particular project and its development phases. Some tools offer an extensibility interface and a scripting language, which may be used for this purpose, forcing the modeler to be an expert in yet another programming language. We prefer to use a single language (OCL) for both, tailoring the modeling tools and the modeling itself. In addition, we can take advantage from the ORDBMS optimizer, in order to achieve efficient evaluation of constraints. To the best of our knowledge our approach is the first one taking advantage of the new database technology for the purposes of checking several kinds of constraints.

Our approach allows us to use OCL for both checking validity of UML models (invariants specified in [5]) and maintaining consistency regarding design guidelines. We have introduced two kinds of design guidelines, global design guidelines and temporary guidelines. In addition, two kinds of OCL constraints related to the modeling process have been examined (pre- and postconditions, design goals).

Currently, a compiler which maps OCL constraints to SQL constraints is under development. We have explained the translation algorithm by an example. It is based on a translation graph consisting of translation nodes and meta-data nodes. Each translation node in the graph represents a building block of the algorithm. An SQL constraint is recursively aggregated by expanding SQL templates, whereas the parameters in the template are substituted by SQL fragments provided by subnodes. Choosing the appropriate SQL template and expanding the parameters depends on meta-data, which is provided by meta-data nodes. So far the compiler is limited to class invariants. The next version will also accept pre- and postconditions.

Finally, we want to mention that our efforts in a semantic-preserving mapping UML/OCL to an ORDBMS interface has given us the opportunity to gain lots of experience about UML/ OCL and to learn much about its deficiencies and weaknesses. In addition we expect our approach to provide a foundation for automated model analysis.

---

[i] Subproject A3 *Supporting Software Engineering Processes by Object-Relational Database Technology* of the Sonderforschungsbereich 501 *Development of Large Systems by Generic Methods*, funded by the German Science Foundation.

References

[1] P. P. Chen: The Entity-Relationship-Model – Towards a Unified View of Data, ACM Transactions on Database Systems, 1(1), 1976

[2] ISO Final Commitee Draft – Database Language SQL, ftp://jerry.ece.umassd.edu/isowg3/dbl/BASEdocs/public, 1999

[3] P. F. Linington: Options for Expressing ODP Enterprise Communities and Their Policies by Using UML, Proceedings Third International Enterprise Distributed Object Computing Conference (EDOC), Mannheim, Germany, September 1999

[4] OMG, UML Notation Guide, Version 1.1, OMG Document ad/97-08-05, September 1997

[5] OMG, UML Semantics,Version 1.1, OMG Document ad/97-08-04, September 1997

[6] OMG, Object Constraint Language Specification, Version 1.1, OMG Document ad/97-08-08, September 1997

[7] OMG, OA&D CORBAfacility Interface Definition, Version 1.1, OMG Document ad/97-08-09, September 1997

[8] W. Mahnke, N. Ritter, H.-P. Steiert: Towards Generating Object-Relational Software Engineering Repositories, 8. Fachtagung Datenbanken in Büro, Technik und Wissenschaft, Freiburg, Germany, March 1998

[9] N. Ritter: DB-based Cooperation Services for Engineering Applications, Ph. D. thesis (in german), 1997

[10] N. Ritter, H.-P. Steiert, W. Mahnke, R. Feldmann: An Object-Relational SE-Repository with Generated Services, Proc. Managing Information Technology Resources in Organizations in the Next Millenium (Computer-Aided Software Engineering Track of IRMA'99), IDEA Group Publ., May 1999

[11] M. Stonebraker, P. Brown: Object-Relational DBMSs - Tracking the next great Wave, Morgan Kaufmann Publishers Inc., San Francisco, 1999

[12] UML Speficiation (draft), version 1.3 beta R7, 'http://www.rational.com/uml/resources/documentation/'

[13] J. Warner, A. Kleppe: The Object Constraint Language – Precise Modeling with UML, Addison Wesley Longman, Inc., Reading, Massachusetts, 1999