Using UML/OCL Constraints for Relational Database Design

Birgit Demuth and Heinrich Hussmann

Dresden University of Technology, Department of Computer Science

Abstract. Integrating relational databases into object-oriented applications is state of the art in software development practice. In database applications, it is beneficial if constraints like business rules are encoded as part of the database schema and not in the application programs. The Object Constraint Language (OCL) as part of the Unified Modeling Language (UML) provides the posssibility to express constraints in a conceptual model unambiguously. We show how OCL, UML and SQL can be used in database constraint modeling, and discuss their advantages and limitations. Furthermore, we present patterns for mapping OCL expressions to SQL code.

1 Introduction

Integrating relational databases into object-oriented applications is state of the art in software development practice. Many object-oriented projects choose to use a relational database management system (RDBMS) for a variety of reasons, e.g. compatibility with existing systems and databases. In the last years, some authors have described transformations from object-oriented class diagrams to database schemas, in order to support the systematic object-oriented development of database applications [1], [3]. However, these approaches are restricted to a simple class-to-table mapping, mainly based on attributes and associations. They do not at all exploit the full power of relational database technology. Advanced static database concepts like automatic check of assertions or maintenance of database integrity by automatically triggered operations are not covered by these mappings. On the other hand, it is well known that the development of database applications benefits from business rules being encoded as part of the database schema, using assertions and triggers. This reduces the amount of required coding and ensures that all applications working on the same database adhere to the same business rules, in particular the same integrity constraints. This paper discusses the use of UML and its constraint language OCL for the design of integrity constraints for relational databases. There are several advantages of such an approach:

– The UML as a widely accepted standard language for object-oriented models ensures support by many CASE tools.
– The OCL provides an abstract and precise language for specifying integrity constraints as invariants for object models. The navigation-oriented approach

of OCL fits somehow with the concept of databases (although it does not completely correspond to relational query languages).
- Current RDBMS implementations often use product-specific dialects of SQL to denote assertions and triggers. The usage of OCL makes the specification of integrity constraints independent of the choice of the RDBMS.
- To write down constraints in a constraint language like OCL during object-oriented design is a methodically recommended but practically time-consuming step. An automatic generation of SQL code from OCL preserves the effort that is invested into writing the OCL specification.
- Finally, specifications of operations by pre- and postconditions in OCL form are a good starting point for the automatic generation of SQL code for database queries, updates or triggers corresponding to these operations.

This paper is structured as follows: Section 2 introduces constraints from a more generalized point of view and provides an overview about relational database constraints expressed in SQL. The essential part of this paper, Section 3, presents a family of patterns for mapping OCL constraints to relational database integrity constraints by means of a UML model example. We illustrate how OCL, UML and SQL-92 can be used in database constraint modeling, and discuss their advantages and limitations. Section 4 gives conclusions and future directions.

2 Constraints from a Database Perspective

2.1 Classification and Comparison of Constraint Languages

A **constraint** is defined as a restriction on one or more values of (part of) an object-oriented model or system [19]. This generic definition can be interpreted in different ways. But, by definition, a constraint is always coupled with a model.

From a methodological viewpoint, constraints are required to specify information, in particular business rules, that otherwise cannot be expressed in the model. In current UML, there is no clear borderline between the object-oriented modeling language and the constraint language. An example is the restriction of the multiplicity of an association, which can be expressed either directly by using a syntactical construct from UML class diagrams or by an OCL constraint (see also [2]). These UML constructs can be understood as convenient shorthand notation for OCL constraints.

In general, constraints can be classified according to various criteria. For a first comparison of OCL and database constraints, the following criteria are useful:

- Classification according to the system view: Constraints can be defined in various views of the system, as the static, dynamic and functional view. These three views correspond to three different types of constraints: In a static view, a constraint usually is an invariant, i.e. a condition which has to hold for any "snapshot" of the state of the whole system or a part of it. In

a dynamic view, constraints are used mainly to express the condition under which a transition from one state into another is allowed (guards). In a functional view, finally, the output values and the induced state transformation of an operation are described with respect to the input values. This is done in OCL by pre- and postconditions, but there are also other possibilities (e.g. axiomatic specification).
- Classification according to the policy for dealing with constraint violations: There are various interpretations for the situation where a constraint, in particular a static constraint (invariant), is not fulfilled. The implementation can be considered as faulty (as is the view in software verification), the recent modification to the state can be made undone (as is the view in database systems), or actions can be taken to automatically correct the state (a view also used in database systems). If a constraint is intended to automatically re-establish a correct state, it is called an *operational* constraint. A *declarative* constraint just states the condition that has to be fulfilled without specifying any consecutive action.

There are other classifications for constraints that are not relevant here. For example, in the most general meaning of the term, a constraint may also be a logical property which is not necessarily decidable (i.e. stating that an operation always has to terminate). Here we assume that all constraints are executable algorithms on a system state.

The following table uses the criteria from above to compare OCL with the integrity constraint checking mechanisms implemented in modern RDBMS:

	OCL	RDBMS
System view	static, dynamic, functional	static
Violation policy	declarative	declarative, operational

The table shows clearly that there is a common intersection between the constraint mechanisms of OCL and RDBMS. Constraints in this intersection are static (i.e. invariants) and declarative which we will consider in this paper. The difference between the two constraint mechanisms is that OCL is defined on a very abstract conceptual (UML) model of the system, whereas database constraints are formulated in rather low-level terms. Therefore, it is an interesting question how and to which extent OCL can be used for the high-level specification of relational database constraints.

2.2 Relational Database Constraints

What UML and OCL calls a constraint has been known in database technology for a long time under various names, for example integrity constraints [8], [18] [15], integrity rule [4], and consistency constraints [10]. We will use the term *integrity constraint* to refer to constraints in relational databases. In [8], types of integrity constraints are identified that are encountered frequently in database schemas:

– An *implicit* constraint represents an integrity rule which is part of the data model and must be specified on individual relations.
– An *explicit* constraint may occur in an application. It is often called a *business rule*.
– An *inherent* constraint does not have to be specified in a schema but are assumed to hold by the definition of the relational model. It can be compared with a UML meta model constraint.

The following table summarizes the semantics of implicit, explicit and inherent constraints in the relational model and their specification in SQL-92 [11]. Only the very well-known trigger concept has been standardized later (SQL-99 [7]). Every SQL constraint is checked either at the end of every SQL insert, update or delete statement (IMMEDIATE mode) or at the end of the transaction (DEFERRED mode).

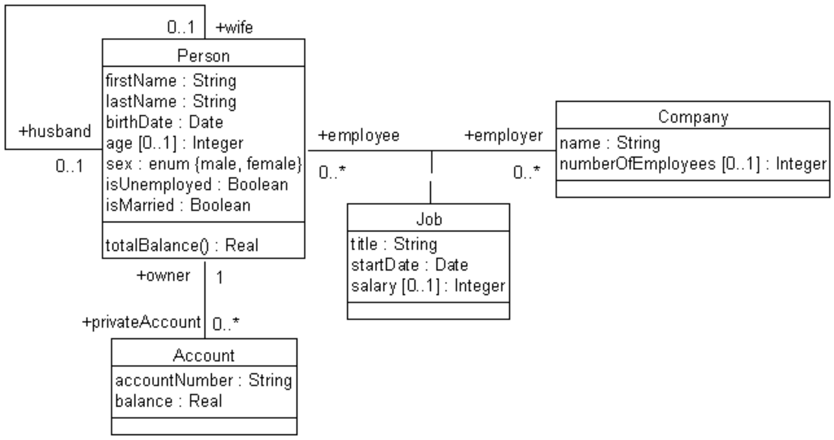
type of integrity constraint	relational model	specification in SQL
implicit	primary key/ entity integrity rule [4]	PRIMARY KEY (NOT NULL)
	foreign key/ referential integrity rule [4]	FOREIGN KEY REFERENCES
explicit	column constraint and table constraint	CHECK, NOT NULL, UNIQUE
	assertion	CONSTRAINT ... CHECK, CREATE ASSERTION
	domain	CREATE DOMAIN
	trigger	CREATE TRIGGER
inherent	atomic attribute values	implicit

Only trigger and referential constraints with actions (SET DEFAULT, SET NULL, CASCADE) are operational constraints. The other ones (including FOREIGN KEY REFERENCES ... NO ACTION) represent declarative constraints.

3 Mapping of UML and OCL Constraints to Relational Database Integrity Constraints

3.1 A Relational Database Schema for a UML Model

Considering the usage of UML and OCL constraints for relational database design we have to deal at first with the impedance mismatch between the object-oriented and the relational model. However, if we bridge the gap between the two models we are able to exploit powerful integrity maintenance mechanisms of advanced RDBMS in object-oriented applications. The figure below contains a similiar UML model example of the OCL specification [14] we will use as an example. In the translation to a database schema, we assume the simple and



commonly used class-to-table mapping [1]. Every class is represented as one table, called class table, on the relational side. Relationships in class diagrams are implemented by foreign key references. According to this mapping, the following tables are generated for our example:

```
create table PERSON  ( PID integer primary key,
                       FIRSTNAME varchar not null,
                       LASTNAME varchar not null,
                       BIRTHDATE date not null,
                       AGE integer,
                       SEX SEXTYPE not null,
                       ISUNEMPLOYED boolean not null,
                       ISMARRIED boolean not null,
                       WIFE_HUSBAND integer references PERSON )

create table COMPANY ( CID integer primary key,
                       NAME varchar not null,
                       NUMBEROFEMPLOYEES integer )

create table ACCOUNT ( AID integer primary key,
                       ACCOUNTNUMBER varchar not null,
                       BALANCE float not null,
                       OWNER integer not null references PERSON )

create table JOB     ( PID integer references PERSON,
                       CID integer references COMPANY,
                       TITLE string not null,
                     STARTDATE date not null,
                     SALARY float,
                     primary key (PID,CID) )

create domain SEXTYPE character check (value in ('m','f'))
```

For dealing with object identifiers, we have chosen the existence-based identity approach [1]. An object identifier attribute is added to each class table and is made the primary key. Therefore, identifiers such as PID in Person are defined as integers. Some RDBMS provide semantic support for such identifiers. It should be noticed that booleans are supported not before SQL-99 [7]. A many-to-many relationship including an association class such as the association between Person and Company is mapped to a distinct table (JOB). The primary key of this table is the combination of the primary keys from each class table. A one-to-many association such as the relationship between Person and Account is implemented with a buried foreign key (OWNER) in the table corresponding to the class on the *many* side (ACCOUNT). Because the multiplicity of Person is exactly one in the UML model, we do not allow the OWNER foreign key to be null. A one-to-one relationship is generally implemented by a buried foreign key in either class. Because our one-to-one relationship is a symmetric association that models the marriage between persons, we did not introduce an asymmetric WIFE or HUSBAND attribute in PERSON, but, as can be seen above, a symmetric WIFE_HUSBAND attribute. Enumerations such as the sex datatype in the class Person can be mapped to SQL domains (SEXTYPE). It should be noticed that some of the important UML model elements and constraints like multiplicity are already encoded by SQL integrity constraints. Besides the above explained integrity constraints, the multiplicity zero of attributes is modeled by null values in relational databases. The default UML value of exactly one is specified as a *not null* constraint in the attribute definition. Note that all SQL integrity constraints are checked automatically at any write transaction.

The translation described here is rather straightforward and based on standard literature. However it should be emphasized that most current commercial CASE tools only generate a significantly simpler database schema.

3.2 OCL Mapping Patterns

We decribe the mapping of OCL to SQL constraints by a family of patterns, where each pattern encapsulates the idea for translation of an OCL language concept. Each pattern is given by its name, a short description and an example in OCL and SQL notation followed by a discussion. Whereas the example illustrates a typical use case of the pattern, the discussion especially points out mapping problems. We use the OCL syntax of the OMG draft 1.3 [14]. The patterns are founded on a comprehensive study of possible transformations of OCL expressions to SQL [16].

OCL Invariants. As stated above, we only consider OCL invariants to transform them to SQL integrity constraints. A naive approach would be the mapping of OCL invariants to SQL table constraints in the following manner:
OCL:

```
context Company inv enoughEmployees:
self.numberOfEmployees > 50
```

SQL:

```
create table COMPANY( ... ,
constraint ENOUGHEMPLOYEES check (NUMBEROFEMPLOYEES > 50))
```

The OCL context is expressed by the create table statement. If we refer to any column of that table (e.g., NUMBEROFEMPLOYEES) just by its name, the context is always the current row of that table. The current row could be seen as the contextual instance `self` in OCL. The problem, however, is to refer to the current row in nested subqueries when we map more complex OCL expressions (see the NAVIGATION pattern below). Though SQL's correlation names could solve this problem, they can not be used for the *context table* of the constraint (in our example COMPANY). Therefore, we must find a mapping where the equivalent SQL constraint includes the context table. This leads to the definition of our first transformation pattern.

Name: OCL INVARIANT
Description: An OCL invariant of the form

```
context <class name> inv <constraint name>:
<OCL expression(self)>
```

is transformed to

```
<class name>.allInstances -> forAll (<OCL expression(self)>).
```

Then, the OCL forAll operation can be mapped to a SQL predicate, and the whole constraint is written as:

```
create assertion <constraint name>
check (not exists
(select * from <context table> SELF
 where not (<OCL expression(self)>))
```

The evaluation of the subquery nested in the exist predicate searches for objects which violate the OCL constraint. If there are no such objects, the constraint is satisfied. Since integrity constraints often involve multiple tables, they should be specified as standalone constraints (assertions). Alternatively to assertions, table constraints or column constraints can be used.

Example :

OCL:

```
context Company inv enoughEmployees:
self.numberOfEmployees > 50
```

SQL:

```
create  assertion ENOUGHEMPLOYEES
check (not exists (select * from COMPANY SELF where
                   SELF.NUMBEROFEMPLOYEES > 50)))
```

Discussion: A SQL constraint can be evaluated based on the null values and a three-valued logic to *unknown*. Although there are some uncertainties in the OCL definition [9], the same three-valued logic is specified for OCL (in OCL, an unknown value is called *undefined*). OCL says nothing about how undefined is handled when true or false is expected. For example, is a constraint broken if it is evaluated to undefined? Because of this uncertainty we assume the same treatment of OCL constraints like in SQL. That is, if the expression is evaluated to undefined, then the constraint is satisfied.

An OCL invariant is specified as an expression over different OCL types. The grouping of OCL types into Basic Types and Collection-Related Types [14] is not very helpful when considering the usage and mapping of OCL expressions in the context of relational databases. OCL Basic Types include such very basic predefined types like Integer as well as types defined in a UML model. In contrary to the OCL specification, Warmer and Kleppe [19] distinguish basic types and model types to separate user-defined types from real basic types. Although the type hierarchy of OCL is somehow confusing [9], it is clear that OCLAny is the supertype of all types in the UML model (*model types*). Therefore, we distinguish model types from basic and collection types in the following. The access to the features of UML classes (attributes, query operations, enumerations defined as attribute types and navigations that are derived from relationships between instances) and its transformation to SQL depends on the class-to-table mapping. All other OCL operations can be transformed in a model independent and therewith database schema independent way if we evaluate them on the basis of the results for each subexpression over model types (see [16]). Collection-related types comprise the abstract supertype Collection as well as the concrete Set, Bag and Sequence types.

Another issue is the result of an OCL expression. The expression that represents the OCL constraint is always of type Boolean and can be specified by a SQL predicate resp. a SQL search condition (see OCL INVARIANT). Each OCL subexpression results either in a Boolean value or in an object of any other OCL type. All expressions that are not of type Boolean should be encoded in SQL by a basic value expression resp. a column reference, or a query expression. If this descriptive specification is not possible, one may always encode the constraint in SQL extended by procedural statements in a computationally complete language and store it as a *stored procedure* in the database [12].

If we consider the OCL types and the results of OCL expressions from the SQL perspective we can identify further OCL mapping patterns. The table below indicates which fields are covered by the different OCL mapping patterns. Metamodel properties of OCL types such as `Person.attributes` are not taken into account because it mostly makes no sense to specify database integrity constraints on the database schema itself.

OCL expression over	Result type of an OCL expression		
	Boolean	Basic value excluding Boolean	Collection
Basic Types	BASIC TYPE	BASIC TYPE	
Model types:			
- Class - Association class	CLASS AND ATTRIBUTE		CLASS AND ATTRIBUTE
- Attribute	CLASS AND ATTRIBUTE	CLASS AND ATTRIBUTE	
- Association end	NAVIGATION	NAVIGATION	NAVIGATION
- Operation	OPERATION	OPERATION	OPERATION
Collection types	COMPLEX PREDICATE	BASIC VALUE	QUERY

In the following patterns, we assume the above described OCL INVARIANT pattern. We consider the transformation of `<OCL expression(self)>` resp. of its subexpressions and use the correlation name SELF in SQL code to map the contextual instance `self`.

Basic Types. There is only one simple pattern. Expressions over basic objects can not result in collections.

Name: BASIC TYPE

Description: The basic predefined OCL types (Real, Integer, String, Boolean), their values as well as their unary, multiplicative, additive, relational and logical operations mostly have a direct SQL counterpart. If the OCL subexpression is of type Boolean like in the example we specify a SQL search condition. Other primitive subexpression, such as the integer value 18, are part of a SQL predicate.

Example :

OCL:

```
[context Person]
self.age > 18 and self.isUnemployed = true
```

SQL:

```
[from PERSON SELF]
self.AGE > 18 and SELF.ISUNEMPLOYED = true
```

Discussion: Operators that have no direct SQL counterpart can be transformed into equivalent expressions, for example `<b1> implies <b2>` into `not <b1> or <b2>`.

Model Types. If we want to map the access to model types, we have to deal with objects of all model types and their properties, which means in database context classes, association classes, attributes, associationEnds and operations.

Name: CLASS AND ATTRIBUTE

Description: Basically, each OCL subexpression which refers to a class, association class or an attribute can be mapped to a SQL query. In the above explained class-to-table-mapping, the access to an attribute is simplified by the SQL notation `<class_table>.<attribute>` resp. `SELF.<attribute>` like we have already used it in the patterns above.

Example : see the OCL INVARIANT pattern

OCL:

```
Person.allInstances
```

SQL:

```
select PID from PERSON
```

Discussion: Here two serious problems arise which are based on the impedance mismatch of the relational and the object-oriented paradigm. The first one is the identification of objects. We have chosen the existence-based approach. Therefore the set of all instances of Person is evaluated by selecting the PID attribute. However, the semantics of an artificial primary key of a table is not exactly the same as that of an object identifier. A primary key only identifies tuples representing objects within the table scope. Furthermore, a primary key can basically be altered and reused. The second problem is the equality of objects. What does

```
object = (object2: OCLAny): Boolean
```

exactly mean? From a database point of view it is useful to distinguish between values (in OCL only basic values such as integers) and objects (for example instances of Person) [5]. Any object is uniquely identified by an object identifier (OID). In our relational database schema mapping this means that the equality of persons has to be checked by the primary key PID of PERSON.

Name: NAVIGATION

Description: Basically, each OCL subexpression that refers to an association end can be mapped to a SQL query. Starting from an specific object, we often have to navigate associations to other objects and their properties to specify a constraint. Mapping such navigations is done by queries with nested subqueries and depends on the kind of association (one-to-one, many-to-one or many-to-many relationship) and its mapping to the database schema.

Example :

OCL:

```
[context Company]
self.employee
```

SQL:

```
[from COMPANY SELF]
select PID from PERSON where PID in
       (select PID from JOB where CID in
               (select CID from COMPANY where CID = SELF.CID))
```

Discussion: The value of a navigation expression is one object (in our relational context one tuple) if the multiplicity of the association end has a maximum of one, otherwise it is a set of objects. After evaluation of any expression, one can always apply another property to the result to get a new result value.

Name: OPERATION
Description: Only operations (or methods) with *isQuery* being true can be used in OCL expressions. In advanced RDBMS, operations can be modeled as user-defined functions and implemented in different ways. Currently, RDBMS vendors integrate Java into their systems to implement user-defined functions as static methods [13]. These methods can be called in a SQL statement.
Example :
OCL:

```
[context Person]
self.totalBalance() > 0
```

Java/SQL:

```
public class Example {
 //the method will be called in a SQL query
 //pid  represents the contextual instance self
 public static double totalBalance(int pid)
  throws SQLException {
  Connection c =
   DriverManager.getConnection("JDBC:DEFAULT:CONNECTION");
  PreparedStatement stmt = c.prepareStatement(
   "select sum(BALANCE)from ACCOUNT where OWNER =?");
  stmt.setInt(1, pid);
  ResultSet sum = stmt.executeQuery();
  sum.next();
  return sum.getDouble(1);
  }
}
```

Discussion: The example shows the implementation of an operation by a Java method that uses JDBC (Java Database Connectivity) to perform database operations. These techniques are currently under standardization. The first RDBMS prototypes with integrated Java are available [13].

OCL Collection Types. Apart from the specification as a literal, the only way to get a Collection is by navigation or by the allInstances operation. These collections are represented as tables resp. relations in the relational model. In opposite to the pure relational model where all relations are sets, the SQL approach provides the possibility to construct bags and sequences beside sets. For example,

```
select distinct TITLE from JOB
```

evaluates a set of titles of all job instances whereas the deletion of the keyword distinct creates a bag.

Name: COMPLEX PREDICATE

Description: The pattern covers all collection operations that result in a Boolean value (includes, excludes, includesAll, excludesAll, isEmpty, notEmpty, exists, forAll, isUnique, sortedby, equality of collections (=)). Such OCL (sub)expressions can be basically mapped to a SQL predicate with or without nested subqueries.

Example :

OCL:

```
[context Company]
self.employee->forAll(isUnemployed = false)
```

SQL:

```
[from COMPANY SELF]
not exists
(select PID from
              (select PID from PERSON where PID in
              (select PID from JOB where CID in
              (select CID from COMPANY where CID = SELF.CID)))
            where
              PID in (select PID from PERSON
                      where  not (ISUNEMPLOYEED = false)))
```

Discussion: Notice that the evaluation of `self.employee` as a set of objects (see NAVIGATION pattern) is given as derived table in the from clause of the exists predicate query. Such complex queries are only supported by a few RDBMS.

Name: BASIC VALUE

Description: The pattern covers all collection operations that result in a Real, Integer or String value (size, count, sum). Such OCL (sub)expressions can be basically mapped to a scalar subquery using SQL aggregate functions (degree and cardinality of the result = 1).

Example :
OCL:

```
[context Person]
self.privateAccount.balance -> sum
```

SQL:

```
[from PERSON SELF]
select case when sum(BALANCE) is null then 0 else sum(BALANCE)
from ACCOUNT where OWNER = SELF.PID)
```

Discussion: Notice that Account objects could not exist. Then, `sum(BALANCE)` results in unknown. According to the semantics of the OCL property sum, it must be transformed to the Real value zero. Such a SQL query is generated using the NAVIGATION pattern specialized for many-to-one relationships.

Name: QUERY
Description: The pattern covers all collection operations that result in a collection again:
- construction of collections (Set, Bag, Sequence) including transformation of different collections (asSet, asBag, asSequence) and flattening of collections
- specification of a subset of a collection (select, reject)
- specification of a collection which is derived from some other collections (collect)
- typical set operations such as union, intersect and difference adopted to sets, bags and sequences.

Such OCL (sub)expressions can be basically mapped to SQL queries with or without nested subqueries.
Example :
OCL:

```
[context Company]
self.employee-> select(p: Person | p.sex = female)
```

SQL:

```
[from COMPANY SELF]
select PID from (select PID from PERSON where PID in
                (select PID from JOB where CID in
                (select CID from COMPANY
                           where CID = SELF.CID)))
           where PID in
                (select P.PID from PERSON P
                             where  P.SEX = 'f')
```

Discussion: A detailed representation of possible transformations is given in [16]. Such OCL (sub)expressions can be basically mapped to a SQL predicate with or without nested subqueries.

A special role is played by the not yet mentioned iterate operation. The iterate operation is very generic and therefore more complicated than the other collection operations which can be partially described in terms of iterate. In its most general form it can only be transformed to SQL by a so-called *procedural mapping pattern*. In this case we need a computationally complete language. That means, an iterate operation must be encoded by a stored procedure. In [16] it is shown how this can be done by Sybase' stored procedures using Transact SQL.

3.3 Problems and Limitations

We have shown that it is possible to generate equivalent SQL code for OCL constraints. Our goal to avoid procedural SQL code has been achieved for the full OCL language, with the only exception of the iterate operation. It must be noticed that our model type mapping patterns are dependent of the assumed class-to-table mapping. We have used the whole SQL-92 language (full level) to encode complex OCL expressions. Especially, the specification of a derived table in the from clause is a fundamental feature to formulate any OCL expressions in SQL in a declarative manner. We know only one of the major commercial RDBMS that provides this expressiveness. Using other (current versions of) RDBMS makes it necessary to map many further OCL operations besides iterate to stored procedures. This has been studied for a less expressive commercial SQL dialect in detail in [16]. A stored procedure for database integrity checking must be called at the end of every transaction. But, there is the hope that more RDBMS vendors provide full SQL-92 conformant database servers very soon.

In consideration of OCL we have referred to the most recent OMG draft [14]. Existing unclarities such as the type hierarchy and the undefined values problem have to be adopted to our patterns after their clarification.

Although some authors emphasize that a UML class diagram and a data model are different things [17], UML is a powerful tool to model persistent objects and therewith databases [[2], [1]]. But for UML to achieve a wide acceptance in database modeling some refinements of UML may be helpful. For example, it is useful to define a standard UML constraint *unique* to model key attributes. A unique attribute constraint could be mapped very simple to a (predefined) unique SQL constraint:

```
create table ACCOUNT (..,ACCOUNTNUMBER varchar not null unique,..)
```

In OCL one can formulate this very often needed constraint by an iterate based expression such as

```
[context Account]
Account.allInstances -> forAll ( a1, a2 |
     a1<>a2 implies a1.accountNumber<>a2.accountNumber )
```

This OCL expression would be mapped to a complex SQL assertion. The following shorthand notation introduced by the most recent OCL specification makes the above OCL constraint more readable and easier to transform to SQL.

```
[context Account]
Account.allInstances -> isUnique (accountNumber)
```

There are many equivalent UML/OCL expressions and notations, within one notation (OCL resp. UML) as well as between both notations [19]. If a code generator will be developed we have to consider these equivalences. A first step should be a normalization of UML and OCL constraints.

4 Conclusion

In this paper, we reported on a systematic study of the use of OCL for the specification of database integrity constraints. It has turned out that OCL is essentially adequate for this purpose. We have proposed a minor addition to UML/OCL which would make specification and transformation into SQL easier. When assuming the most advanced SQL standard available, there is only one language construct of OCL (iterate) which cannot be mapped to declarative SQL in every case. However, this seems not to be a serious problem, since in practical OCL specification the iterate operator is rarely used, and all OCL constructs derived from iterate (like forAll, select) can be mapped properly. The practical value of UML-to-SQL mappings as described in this paper can only be judged with the help of tools, in order to create sufficiently large examples. So we plan as a next step the realization of a code generation tool for SQL code from OCL, based on our Dresden UML Toolset [6]. Dresden UML Toolset provides a platform for experimental research on the basis of UML (code generation, reverse engineering from legacy applications and extensions of UML for formal specification such as OCL). It integrates commercial UML tools with experimental tools developed at the university and combines experimental UML tools. One of our first steps in development of Dresden UML Toolset was the implementation of an OCL basic library. Currently, we are developing a modular OCL compiler which generates Java code to evaluate OCL expressions using the OCL basic library. The OCL compiler has an interface for the code generation of SQL constraints. Such planned OCL-to-SQL tool will enable also experiments with different RDBMS systems and evaluation of the performance achieved by the generated integrity constraints. Further theoretical investigations are required in several directions. It would be useful to make the OCL-to-SQL mapping more independent of the code generation algorithm for the database schema. Moreover, the generation of SQL code can be extended from integrity constraints to the specification of query and update operations. There should be a sublanguage of OCL pre- and post-conditions from which executable SQL code can be generated. This approach can form a basic technology for the automatic generation of the core part of a database application from a high-level specification. The fact that UML and OCL are standardized languages gives this approach a potential for much higher impact than the proprietary application generation tools available today.

Acknowledgment: The authors thank Alexander Schmidt for his extensive study of OCL-to-SQL transformations and valuable contributions during his diploma project in 1998.

References

1. Blaha, M., Premerlani, W.: Object-Oriented Modeling and Design for Database Applications. Prentice Hall, 1998
2. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley, 1999
3. Bruce, K., Whitenack, B.: Crossing Chasms - A Pattern Language for Object-RDBMS Integration. Knowledge Systems Corp., `ftp://members.aol.com/kgb1001001/Chasms/chasms.pdf`
4. Date, C.: An Introduction to Database Systems. Volume I, Fifth Edition, Addison-Wesley, 1990
5. Demuth, B., Geppert, A., Gorchs, T.: Algebraic Query Optimization in the CoOMS Structurally Object-oriented Database System. in: Freytag, Ch., Maier, D., Vossen, G. (Ed.): Query Processing For Advanced Database Systems. Morgan Kaufmann, 1994
6. Dresden UML Toolset, `http://www-st.inf.tu-dresden.de/UMLToolset`
7. Eisenberg, A., Melton, J.: SQL: 1999, formerly known as SQL-3. ACM SIGMOD Record, 22(1999)1, 131-138
8. Elmasri, R., Navathe, S.: Fundamentals of Database Systems. Benjamin/Cummings, 1989
9. Gogolla, M., Richters, M.: On Constraints and Queries in UML. in: Schader, M., Korthaus, A., (Ed.): The Unified Modeling Language. Technical Aspects and Applications. Physica-Verlag, 1998
10. Korth, H., Silberschatz, A.: Database System Concepts. Second Edition. McGraw-Hill, 1991
11. Melton, J., Simon, A.: Understanding the New SQL: A Complete Guide. Morgan Kaufmann, 1993
12. Melton, J.: SQL's Stored Procedures. A Complete Guide to SQL/PSM. Morgan Kaufmann, 1998
13. Olson, St. et. al.: The Sybase Architecture for Extensible Data Management. Bulletin of the Technical Committee on Data Engineering, IEEE Computer Society, 21(1998)3, 12-24
14. OMG UML Specification v. 1.3 draft
15. Ricardo, C.: Database Systems. Principles, Design, and Implementation. Macmillan, 1990
16. Schmidt, A.: Untersuchungen zur Abbildung von OCL-Ausdruecken auf SQL. Technische Universitaet Dresden, Diplomarbeit, 1998
17. Simons, A., Graham, I.: 37 Things that Don't Work in Object-Oriented Modeling with UML. in: Kilov, H., Rumpe, B. (Ed.): Second ECOOP Workshop on Precise Behavioral Semantics. Technical Report, Technische Universitaet Muenchen, TUM-19813, Juni 1998
18. Teorey, T.: Database Modeling & Design. The Fundamental Principles. Second Edition, Morgan Kaufmann, 1990
19. Warmer, J., Kleppe, A.: The Object Constraint Language. Precise Modeling with UML. Addison-Wesley, 1999