DOI: 10.2298/CSIS121210067R

Possible Realizations of Multiplicity Constraints

Zdeněk Rybola[1] and Karel Richta[23]

[1] Faculty of Information Technology, Czech Technical University in Prague
Thákurova 9, 160 00 Prague
zdenek.rybola@fit.cvut.cz
[2] Faculty of Mathematics and Physics, Charles University in Prague
Malostranské nám. 25, 118 00 Prague
richta@ksi.mff.cuni.cz
[3] Faculty of Electrical Engineering, Czech Technical University in Prague
Technická 2, 160 00 Prague
richta@fel.cvut.cz

Abstract. Model Driven Development (MDD) approach is often used to model application data and behavior by a Platform Independent Model (PIM) and to generate Platform Specific Models (PSMs) and even the source code by model transformations. However, these transformations usually omit constraints of the binary association multiplicities, especially the source class optionality constraint.
This paper is an extended version of the paper 'Transformation of Special Multiplicity Constraints - Comparison of Possible Realizations' presented at MDASD workshop at the FedCSIS 2012 conference. In this paper, we summarize the process of the transformation of a binary association from a PIM into a PSM for relational databases. We suggest several possible realizations of the source class optionality constraint to encourage the automatically transformation and discuss their advantages and disadvantages. We also provide experimental comparison of our suggested realizations to the common realization where this constraint is omitted.

Keywords: MDD, UML, transformation, multiplicity constraints, source class optionality constraint, OCL, SQL.

1. Introduction

Model Driven Development (MDD) is a development process that is based on modeling and transformations. In our case, it is based on the Model Driven Architecture (MDA) developed by the Object Management Group (OMG) [8, 10]. This process usually consists of creating a set of models of various abstraction levels and points of view. The process also consists of various transformations between these models. These transformations usually support both forward engineering and reverse engineering, the processes of transforming abstract models into more specific models or source code, or specific models into more abstract models, respectively.

The most common use case of MDD approach is the development of a Platform Independent Model (PIM) of the application data and its transformation to a Platform Specific Model (PSM) for a relational database, as well as

the generation of SQL scripts to create the database schema. However, these transformations usually do not take the multiplicity constraints into account, and therefore a database schema created according to the generated PSM can be inconsistent according to the defined multiplicity constraints and the database can contain invalid data.

Therefore, in this paper, we deal with the transformation of binary associations along with their multiplicity constraints from a PIM into a PSM for relational databases. Many CASE tools such as Enterprise Architect [16] support the model transformation and the source code generation. However, they have many limitations regarding the integrity and multiplicity constraints [3]. The tools usually do not transform these constraints to an implementation.

In particular, we focus on a special case of a multiplicity constraint – the source class optionality constraint – that we consider the most often neglected constraint during the transformations. We define this constraint using another formalism then the grafical notation of the class diagram of the Unified Modeling Language (UML) – as an invariant in the Object Constraint Language (OCL). We believe such a definition can be transformed into the implementation more straightforwardly than it is done so far. For instance, OCL tools such as DresdenOCL Toolkit [4] can be used to transform such a constraint into an implementation.

Our motivation for this research is the intent to bring this issue in attention of the community of data analysts and database designers and to show that this constraint can be quite easily realized in common relational databases. We also believe that the integration of the suggested realizations in the transformation processes of CASE tools may save a lot of effort of analysts and database designers when trying to design a consistent database and even improve the database consistency as this effort is usually neglected. Therefore, we want to stimulate the motivation of CASE tool and transformation tool builders to include a realization for such a constraint in their tools to support this case of the MDD approach. Therefore, we propose several possible implementations for this constraint in relational databases and we discuss advantages and disadvantaged of each suggested implementation. Finally, we provide an experimental comparison of suggested implementations to the common approach, without the source entity optionality constraint implemented. The comparison is done from the point of view of database operations – inserts to the database, queries to the database and deletes of the data from the database.

This paper is an extended version of [14]. It extends the work presented in [13] and [11] where the rules for the transformation of a binary association from a PIM into a PSM are discussed. The contributions of this paper are the constraint implementation, including the update and delete operations, and new experiments for the delete operation and more suitable examples.

The paper is structured as follows: In Section 2, we present a running example to define basic assumptions and illustrate our approach. In Section 3, we discuss related work and existing tools and their problems in comparison to our approach. In Section 4, the transformation of a binary association and

its multiplicity constraints from a PIM into a PSM for a relational database is discussed using an example. Various possible realizations of the special constraint for the source entity optionality are defined and discussed in Section 5. Experiments and their results are discussed in Section 6. Finally, in Section 7, the conclusions are given.

2. Running Example

In the running example, we use UML to express models and we use SQL as the domain specific language for relational databases used in the implementation. UML [9, 2] is a general-purpose visual modeling language for specifying, constructing, and documenting the artifacts of systems. Additional constraints for UML models are usually defined in OCL [7], which is a part of UML specification. OCL is a specification language used to define restrictions, such as invariants, pre- and post-conditions for the connected model elements. The invariants are conditions that must be satisfied by all instances of the element. OCL can also be used as a general object query language.

In the PIM, each object of a problem domain is represented by a class – in some languages called *an entity* – with a set of attributes and its instances [2]. The classes are linked together by associations to represent the relationships between the objects – instances of the respective classes. Each association has its name to describe the meaning of the relationship and multiplicities to define the number of instances of each class related to each other. Fig. 1 shows a general form of modeling a binary association by the means of a UML class diagram [2].

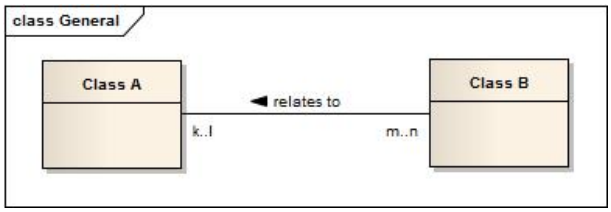

Fig. 1. Labeling of the multiplicities of an association between two classes

The *minimal multiplicity* defines the minimal number of instances of one class related to a single instance of another class. In Fig. 1, value *k* denotes the minimal multiplicity of instances of the *ClassA* for a single instance of the *ClassB* and the value *m* denotes the minimal multiplicity of instances of *ClassB* for a single instance of the *ClassA*. Although this constraint can be generally used to restrict the minimal number of instances to any value possible, for instance at least 11 members for a soccer team, usually the constraint is only used to restrict the *optionality* of the instances – if there needs to be at least one instance

related – value *k = 1* – or if there can be no instances related at all – value *k = 0*.

The *maximal multiplicity* – also called *cardinality* – defines maximal number of instances of one class related to a single instance of another class. In Fig. 1, the values *l* and *n* denote the maximal multiplicities of the *ClassA* and *ClassB*, respectively. Although this constraint can be generally used to restrict the maximal number to any possible value, for instance at most 11 members of a soccer team playing in a match at a time, usually the constraint is used just to distinguish if there can be just one instance related – value *l = 1* – or there can be a collection of instances related to the same instance – value *l = **.

Further on, we will deal only with the minimal multiplicity values of *0* and *1* and the maximal multiplicity values of *1* and ***. However, our approach can be generalized for any special multiplicity values, whenever we want to restrict the number to other values.

When transforming the PIM into a PSM for a relational database, each one–to–many association is transformed to a foreign key constraint. Because the foreign key is unidirectional, we need to distinguish between the *source* and *target* class or table. The *source* class of an association is the class that is transformed into the table where the foreign key value is situated. The *target* class of an association is the class that is transformed into the table that is referred by the foreign key constraint. Usually, the *source* class is the class at the end of the association where the maximal multiplicity value is *n = ** and the *target* class is the class at the end of the association where the maximal multiplicity value is *l = 1*. Also notice that the association in the PIM is non-directional. That is because on the PIM level we only define that two classes of instances are related and define the association multiplicities but we do not define the direction of the association's realization – the direction is defined on the PSM level or during the transformation. The determination of the source and target classes of an association are discussed in more detail in [13].

In this paper, we focus mainly on the source class multiplicity constraint used in one-to-many relationships where the minimal multiplicity value of the *many*-class is equal to *one*. This constraint is often used in models when we need to restrict the required existence of both related entities in such a relationship – none of them can exist without the other one. Our approach will be illustrated on an example of ordered items, where each order must include at least one item and each item must be part of an order. The PIM of the example is shown in Fig. 2. According to the maximal multiplicities of the association, the *OrderItem* class is the *source* class and the *Order* class is the *target* class.

3. Related Work

The problem of the transformation of a PIM of the application data into the relational database is not new. There is a lot of books such as Rob and Coronel [12] describing the principles of the data modeling and the transformation techniques to the database. It is also a part of the information technology education

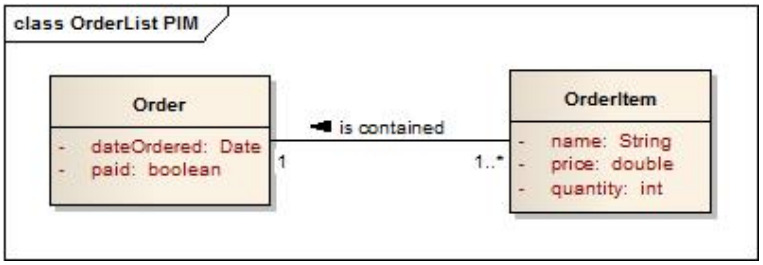

Fig. 2. PIM of one-to-many relationship of an order and its items

in most universities worldwide. Tools such as DresdenOCL [4] and Enterprise Architect [16] provide the support for such a modeling and transformations.

Rob and Coronel [12] presented the basic transformation of an ER model into database tables. They utilized FOREIGN KEY constraint to realize binary relationships and UNIQUE and NOT NULL constraints to restrict the multiplicities. They also suggested using an ON DELETE RESTRICT clause for the FOREIGN KEY constraint to prevent violation of the target entity optionality constraint, if required. However, this clause restricts only the target entity optionality. Furthermore, they suggested no solution to restrict the source entity optionality. Their suggested transformation can be also used for the transformation of a PIM into a PSM for a relational database as discussed in Section 4 with additional constraint for the *source* class optionality constraint.

In [3], Cabot and Teniente identify various limitations of a current code generation tools. The limitations concern the integrity constraints defined in PIMs, including OCL constraints and multiplicity constraints. In our paper, we focus on the multiplicity constraints and propose possible realizations of such constraints in relational databases. Regarding these constraints, Cabot and Teniente [3] identified only one tool called Objecteering/UML [15] that is able to correctly transform multiplicity constraints. In addition to the tools compared in [3], we also identify another CASE tool with similar limitations. Enterprise Architect (EA) [16] is a complex commercial CASE tool for maintenance of models, their transformations, source code generation and reverse engineering process from a source code into PSM. Besides, it provides transformations from PIM data model to a specific database PSM model, and a generation of SQL source code from such a PSM model. However, the default transformations of Enterprise Architect do not consider the optionality of associations to determine neither the direction of the relationship implementation by the FOREIGN KEY constraint nor the required multiplicity restrictions. It does not support special multiplicity values either. Although EA allows the definition of OCL constraints, the constraints are not realized by the transformations.

In [1], the authors also identify a problem of current relational databases in the realization of a source entity optionality constraint – they call this constraint in the database an *inverse referential integrity constraint (IRIC)*. The authors also present an approach to the automated implementation of the IRICs by

database triggers in a tool called IIS*Case. This tool is designed to provide a complete support for developing database schemes including the check of the consistency of constraints embedded into the DB [1] and the integration of subschemas into a relational DB schema [5].

DresdenOCL Toolkit [4, 17] is a research project at the Technical University of Dresden. After loading a model and its instance along with a set of OCL constraints, the tool provides OCL syntax checking and OCL constraints evaluation. It also provides generation of SQL tables and views according to the model. OCL constraints are transformed into database views containing only records satisfying the constraint. The tool also offers transformation of the model with constraints into AspectJ for the Java source code. However, the DresdenOCL Toolkit does not consider the minimal multiplicity constraints of associations in the PIM to determine neither the *source* and *target* tables for the FOREIGN KEY constraint nor the other multiplicity constraints' realization.

4. Transformation of PIM into PSM for Relational Databases

Our approach to the transformation of a data PIM into a PSM for relational databases has been introduced in [13, 11]. This section briefly summarizes our approach.

In general, data is stored as rows in tables with a set of columns to store specific data in a relational database. Therefore, the classes of PIM are transformed into database tables with the columns corresponding to the attributes. Each row in a database table is identified by a PRIMARY KEY. The PSM generated by the transformation of the PIM of our running example (see Fig. 2) is shown in 3. The class *Order* is transformed into the *Order* table and the class *OrderItem* is transformed into the *OrderItem* table. Also notice the PRIMARY KEY columns *orderID* and *orderItemID* and constraints denoted with *PK* stereotype and prefix to identify individual rows in the *Order* and the *OrderItem* tables, respectively. In the following, we will use the *source* and *target* tables as the tables generated by the transformation of the *source* and *target* classes of the PIM, respectively, to discuss the realization of the multiplicity constraints in the PSM.

Associations defined in the PIM are realized by a mechanism called FOREIGN KEY [12]. This mechanism adds a special column or columns to the *source* table and defines the FOREIGN KEY constraint linking the FOREIGN KEY column or columns of the *source* table to the PRIMARY KEY column or columns of the *target* table. In the Fig. 3, the *orderID* column in the *OrderItem* table is defined for the FOREIGN KEY value and the FOREIGN KEY constraint is defined for that column to refer to the *orderID* column of the *Order* table. Using this mechanism, each row can refer only to a single target row, thus we can realize only one-to-one and one-to-many associations and the cardinality of the *target* table is always restricted to *1* [6]. However, many-to-many associations can be transformed into two many-to-one associations and an association table and these can be then transformed as usual [12].

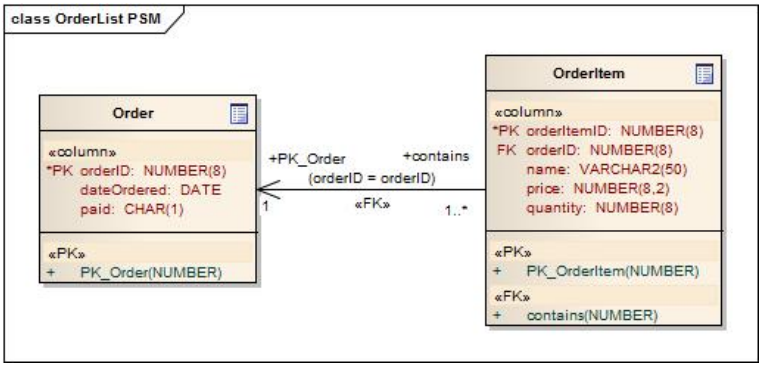

Fig. 3. PSM of one-to-many relationship of an order and its items

In fact, this restriction of the foreign key mechanism is the most important clue to determine the direction of the association. In the running example in Fig. 2, the cardinality $n = *$ requires the FOREIGN KEY in the table *OrderItem* which refers to table *Order* as shown in Fig. 3, and therefore it automatically restricts the cardinality of the target table to $l = 1$.

The *target* table optionality $k = 1$ can be realized by the NOT NULL constraint defined on the FOREIGN KEY column *orderID* in the *OrderItem* table. This constraint enforces each row in the *source* table *OrderItem* to refer to a row in the *target* table *Order* and thus restricting the *target* table optionality. Furthermore, for the completeness of the multiplicity constraints discussed, a *UNIQUE* constraint on the FOREIGN KEY column of the *source* table may be used to restrict the *source* table cardinality $n = 1$ for one-to-one associations, as the constraint prevents the insertion of more rows in the *source* table with the same FOREIGN KEY value. However, this is not the case of our running example.

The only multiplicity value we have not restricted yet is the *source* table optionality $m = 1$. There is no possible way to restrict the source entity optionality by the means of the FOREIGN KEY. As mentioned before, the usual method is to omit this restriction and to provide the constraint checking by the application that uses the database schema [12, 13]. However, we suggest a method to express this constraint by an OCL invariant, and realize it in various ways in SQL to keep the database consistent, independently of the application. The OCL invariant is shown in Fig. 4.

```
context o:Order inv minItems:
OrderItem.allInstances()->exists(i|i.orderID = o.orderID)
```

Fig. 4. OCL constraint for the required source entity optionality

This constraint can be violated only by three operations:

1. If a new order is inserted with no items referring to this new order.
2. If the last item of an order is updated, changing its order to another one.
3. If the last item of an order is deleted.

Therefore, when executing these operations, the checks of the defined OCL invariant must be executed to ensure the data consistency. Moreover, in a relational database, one more operation can violate the constraint: if the order's ID is changed to a new value with no items referring to it. But, this operation also violates the *FOREIGN KEY* constraint, and therefore it is not possible to execute such an operation without changing the order's items, as well.

5. Realization of the Source Table Optionality Constraint

SQL scripts for creating database tables can be generated from the PSM by many tools including the EA. The creation scripts for the database tables used in the following examples of realizations of the source table optionality constraint are shown in Fig. 5. All examples are given in the Oracle SQL syntax.

```
CREATE TABLE Order (
orderID      NUMBER(8) NOT NULL,
dateOrdered  DATE,
paid         CHAR(1));

CREATE TABLE OrderItem (
orderItemID  NUMBER(8) NOT NULL,
orderID      NUMBER(8) NOT NULL,
name         VARCHAR2(50),
price        NUMBER(8,2),
quantity     NUMBER(8));

ALTER TABLE Order ADD CONSTRAINT PK_Order
PRIMARY KEY (orderID) USING INDEX;

ALTER TABLE OrderItem ADD CONSTRAINT PK_OrderItem
PRIMARY KEY (orderItemID) USING INDEX;

ALTER TABLE OrderItem ADD CONSTRAINT isContained
FOREIGN KEY (orderID) REFERENCES Order (orderID);
```

Fig. 5. SQL script for creating database tables of the running example

In some cases, after adding another constraint for checking the existing items for an order, we could not be able to insert new data because of two

mutually dependent checks – the constraint checking existing items for an order, and the *FOREIGN KEY* constraint *isContained* requiring an existing order for each of the order item. This conflict can be solved by deferring one of the constraints [6]. Defining a constraint as deferrable causes the database engine to check the constraint at the end of the transaction instead of checking it in the time of inserting the data. By the deferred *FOREIGN KEY* constraint, we can insert the order items referring to the order not inserted yet, and then to insert this order. The other constraint would be evaluated when inserting the order but, in that time, there already exist the items referring to it. On the other hand, the *FOREIGN KEY* constraint is not evaluated while inserting the items, it is evaluated at the end of the transaction when the order has already been inserted. The deferred *FOREIGN KEY* constraint can be defined as shown in Fig. 6.

```
ALTER TABLE OrderItem ADD CONSTRAINT isContained
FOREIGN KEY (orderID) REFERENCES Order (orderID)
DEFERRABLE INITIALLY DEFERRED;
```

Fig. 6. SQL script for creating the deferrable FOREIGN KEY constraint

The following subsections deal with the possible implementations of the required source table optionality constraint and their pros and cons.

5.1. Database Views

The most straightforward realization of the constraint are the database views [13, 11]. Each constraint is transformed into a database view to filter only the valid data stored in a table. This approach is inspired by DresdenOCL Toolkit [4] that transforms defined OCL constraints into the database views. These views contain only the rows that satisfy the defined constraint using the WHERE clause. The realization of the constraint for the required optionality of OrderItem in Fig. 3 can be defined as shown in Fig. 7.

```
CREATE VIEW valid_orders AS
SELECT o.* FROM Order o WHERE EXISTS
(SELECT 1 FROM OrderItem i WHERE i.orderID = o.orderID)
```

Fig. 7. SQL script for creating the view to select only valid orders

The realization by the database views does not increase the time required for inserting new entries to the tables because the data is inserted directly into

the table without any additional constraints checks. On the other hand, the selection of valid data contains the evaluation of the condition of the view, which increases the time required to query the data.

This approach does not automatically ensure consistency of the data stored in the database. We are still able to insert invalid data, which can violate the multiplicity constraints defined in the PIM. The application itself must use the view to work with the valid data only and must provide support for the correction of the invalid data. For this process, an inverse view can be useful to detect the invalid data violating the constraints. Such an inverse view can be defined as shown in Fig. 8.

```
CREATE VIEW invalid_orders AS
SELECT o.* FROM Order o WHERE NOT EXISTS
(SELECT 1 FROM OrderItem i WHERE i.orderID = o.orderID)
```

Fig. 8. SQL script for creating the view to select invalid orders

Updatable Database Views. To overcome this problem of the invalid data being hidden by the view, DML operations should be executed on the view instead of executing them over the tables directly. To be able to execute DML operations on the view, the view must be updatable. A view is *updatable*, if:

- it does not use a *DISTINCT* quantifier, a *GROUP-BY* or a *HAVING* clause,
- all derived columns appear only once in the *SELECT* list,
- each column of the view is derived from exactly one table,
- and the table is used in the query expression in such a way that its primary key or other candidate key relationships are preserved [6].

```
CREATE VIEW valid_orders AS
SELECT o.* FROM Order o WHERE EXISTS
(SELECT 1 FROM OrderItem i WHERE i.orderID = o.orderID)
WITH CHECK OPTION
```

Fig. 9. SQL script for creating the view to select only valid orders with CHECK OPTION clause

If the view is updatable, then DML operations like inserts, updates and deletes can be executed on the view. In fact, the operations are translated to the corresponding underlying table or tables, and executed on the data directly in these tables. Therefore, it is possible not only to manipulate with the data which is

not accessible through the view, but it is also possible to violate the source table optionality constraint. To prevent such operations that affect the data which is not selected by the view, the view must be defined with the *WITH CHECK OPTION* clause [6]. This clause prevents an insertion of not accessible records and update operations that make accessible records inaccessible by the view. Example of the view definition with the check option is shown in Fig. 9.

```
INSERT INTO valid_orders (ORDERID, ORDER_DATE, PAID)
VALUES (3, sysdate, 'N');
```

Fig. 10. SQL script for inserting a new order using the view with CHECK OPTION clause

If we try to insert a new order to the database using the view as shown in Fig. 10, an exception is thrown as shown in Fig. 11.

```
Error report:
SQL Error: ORA-01402: view WITH CHECK OPTION
  where-clause violation
01402. 00000 -  "view WITH CHECK OPTION where-clause violation"
*Cause:
*Action:
```

Fig. 11. Exception thrown by Oracle database when trying to insert new record that is not accessible by the view used for insertion

Using the updatable view with a check constraint, we can ensure that no invalid data is inserted into the *Order* table. However, we are still able to violate the source entity optionality constraint either by deleting the last item in the order or by updating the last item to another order. To prevent such operations, a view with *CHECK OPTION* should be defined joining the *Order* table and the *OrderItem* table as shown in Fig. 12. In this view, an *OUTER JOIN* must be used to filter out the orders without any item, and thus violating the *WHERE*-clause as shown in Fig. 11. However, this view is not updatable because of that *OUTER JOIN*, and therefore any updates and deletes result in an exception as shown in Fig. 13.

5.2. CHECK Constraint

In relational databases, *CHECK* constraint can be used to restrict the values in a column of a table [6]. The constraint is checked whenever a value is inserted or updated in the column, and the operation is rolled back when the constraint is violated. Such a constraint can restrict a range for the numeric values or provide

```
CREATE VIEW valid_items AS
SELECT i.*,
(SELECT COUNT(*) FROM OrderItem WHERE orderID = o.orderID) items
FROM Order o
  LEFT OUTER JOIN OrderItem i ON (o.orderID = i.orderID)
WHERE (SELECT COUNT(*)
       FROM OrderItem WHERE orderID = o.orderID) > 0
WITH CHECK OPTION;
```

Fig. 12. SQL script creating the view on the orders and their items with a CHECK OPTION clause

```
Error report:
SQL Error: ORA-01779: cannot modify a column which maps to
  a non key-preserved table
01779. 00000 -  "cannot modify a column which maps to
  a non key-preserved table"
*Cause:    An attempt was made to insert or update columns
           of a join view which map to a non-key-preserved
           table.
*Action:   Modify the underlying base tables directly.
```

Fig. 13. Exception thrown by Oracle database when trying to insert a new record that is not accessible by the view used for the insertion

a list of valid values. By this approach, we can define a *CHECK* constraint to allow only the primary key values of the orders that are referred by the rows in the order items' table. According to the SQL:1999 specification [6], the constraint for the situation in Fig. 3 can be defined as shown in Fig. 14.

```
ALTER TABLE Order ADD CONSTRAINT order_check
CHECK (orderID IN (SELECT orderID FROM OrderItem))
```

Fig. 14. SQL script to create the CHECK constraint

As the *CHECK* constraint and the *FOREIGN KEY* constraint are mutually dependent, one of them must be defined as deferrable. Other ways, we would not be able to insert a new record to any of the two tables. We will suggest the deferrable FOREIGN KEY constraint as shown in Fig. 6.
By this realization, the data consistence is ensured, since it is impossible to insert invalid data. However, there are some problems with this implementation. One of the problems is as follows: if a violation is detected by the deferred constraint, the whole transaction is rolled back because it is not possible to determine which command caused the violation [6]. Another important problem

```
Error report:
SQL Error: ORA-02251: subquery not allowed here
02251. 00000 -  "subquery not allowed here"
*Cause:    Subquery is not allowed here in the statement.
*Action:   Remove the subquery from the statement.
```

Fig. 15. Exception thrown by the Oracle database when trying to create the CHECK constraint

of this realization is the fact that, although specified by the SQL:1999 specification [6], none of the current common database engines support this kind of the *CHECK* constraints because it contains a subquery. The Oracle database returns the error message (see Fig. 15) when trying to create the *CHECK* constraint.

Therefore we cannot use this realization until the database engines provide the support for this specification.

5.3. Triggers

Triggers are special procedures available in many relational databases [6] connected to some special events in the table. In Oracle database, each trigger can be defined to be executed *BEFORE* or *AFTER* such an event, while the event can be any statement to insert new rows, update rows or delete rows, including combinations. Furthermore, the triggers can be defined to be executed for each affected row or for all rows affected by the statement at once. During the execution of the trigger, the original row data and the new row data can be accessed by special keywords. In the following, we will use the syntax of the Oracle PL/SQL language to define triggers but similar approach can be used in other databases and database languages as well. The generic form of triggers for inverse referential integrity constraint can be seen in [1].

In the context of constraints checking, a trigger can be defined to check the validity of the inserted data. Such a trigger could throw an exception if the inserted data is invalid. For the situation in Fig. 3, the trigger would check the existence of order items for the inserted order. The insert trigger for Oracle 10g database can be defined in Oracle PL/SQL as shown in Fig. 16.

This trigger is executed before each insert statement, which is executed for the *Order* table. The order items referring to the inserted order by its PRIMARY KEY are being searched. If no items are found, the exception is thrown, which causes the statement to roll back. As this trigger is always executed in the time of an order insertion, the items must be inserted before this statements. To enable it, the FOREIGN KEY constraint on the *orderID* column of the *OrderItem* table must be defined as deferrable (see Fig. 6).

The trigger ensures the data inserted into the database is consistent, as it does not allow to insert the invalid data violating the multiplicity constraint. However, the check is executed for each order insertion or update searching

```
CREATE OR REPLACE TRIGGER check_existing_items_insert
BEFORE INSERT ON Order
FOR EACH ROW
DECLARE
  l_count NUMBER;
BEGIN
  SELECT COUNT(*) INTO l_count
  FROM OrderItem i
  WHERE i.orderID = :new.orderID;
  IF l_count = 0 THEN
    raise_application_error (-20910,
      'order item not found for the inserted order');
  END IF;
END
```

Fig. 16. SQL script for creating the trigger to check the constraint violation while inserting new orders

for the related items. This search takes the longer time the more records have already been stored in the table. However, this searching time can be decreased by defining an index on the FOREIGN KEY column *orderID* in the source table *OrderItem*. For the situation in Fig. 3, the index can be defined as shown in Fig. 17.

```
CREATE INDEX items_order_index ON OrderItem (orderID);
```

Fig. 17. SQL script for creating the index on orders of items

Moreover, this trigger does not prevent the violation of the source table optionality constraint by updating or deleting the items of an order. To prevent such violations, another trigger must be defined, see Fig. 18. This trigger checks, if there exists at least one order item for the currently referred order after updating or deleting the order item.

However, this trigger causes a mutating table exception, see Fig. 19, when trying to update or delete an item. This exception is caused because a query is executed on the table that is currently being updated and therefore the data cannot be reliable to resolve the query.

This problem can be solved by a trigger fired *AFTER* the event on the *STATEMENT* level as shown in Fig. 20. This trigger is fired after the operation of update or delete was executed and all the data was updated. Then, the trigger checks if there are any orders without the items. If it finds such orders, it throws an exception that causes the whole transaction to roll back. However, such a trigger can not detect which item caused the constraint violation.

```
CREATE OR REPLACE TRIGGER check_existing_items_up_del
BEFORE UPDATE OR DELETE ON OrderItem
FOR EACH ROW
DECLARE
  l_count NUMBER;
BEGIN
  SELECT COUNT(*) INTO l_count
  FROM OrderItem i
  WHERE i.orderID = :old.orderID
    and i.orderItemID <> :old.orderItemID;

  IF l_count = 0 THEN
    raise_application_error (-20910,
      'No item left for the order ' || :old.orderID || '!');
  END IF;
END;
```

Fig. 18. SQL script for creating the trigger to check the constraint violation while updating or deleting items

```
Error report:
SQL Error: ORA-04091: table ORDERITEM is mutating,
           trigger/function may not see it
ORA-06512: at "CHECK_EXISTING_ITEMS_UP_DEL", line 4
ORA-04088: error during execution of trigger
           'CHECK_EXISTING_ITEMS_UP_DEL'
04091. 00000 -  "table %s.%s is mutating, trigger/function
  may not see it"
*Cause:    A trigger (or a user defined plsql function that
           is referenced in this statement) attempted to look
           at (or modify) a table that was in the middle of
           being modified by the statement which fired it.
*Action:   Rewrite the trigger (or function) so it does not read
           that table.
```

Fig. 19. Exception thrown by the Oracle database when trying to update or delete a record from the OrderItem table

```
CREATE OR REPLACE TRIGGER check_existing_items_up_del
AFTER UPDATE OR DELETE ON item_table_trigger
DECLARE
  l_count NUMBER;
BEGIN
  SELECT COUNT(*) INTO l_count FROM (
    SELECT o.orderID
    FROM order_table_trigger o
    LEFT OUTER JOIN item_table_trigger i
      ON (o.orderID = i.orderID)
    GROUP BY o.orderID HAVING COUNT(i.orderItemID) = 0);

  IF l_count > 0 THEN
    raise_application_error (-20910,
      'No item left for an order!');
  END IF;
END;
```

Fig. 20. SQL script for creating the trigger to check the constraint violation after update or delete of items

6. Experiments

To compare our proposed implementations, we made some experiments. These experiments compare our proposed realizations with the commonly used realizations without the *source* table optionality constraint checking in three areas - in inserting new orders, in selecting existing orders, and in deleting order items.

The suggested implementation by the triggers requires the select operations being executed during the insertion of the new entries to the table. Similarly, the insert operations by the view with the *CHECK OPTION* require a condition evaluation. Therefore we made an experiment to compare our suggested implementations by the triggers and views with the *CHECK OPTION* with the commonly used realization omitting this constraint. In our experiment, the implementation by the check constraint should be also tested but it cannot be implemented in the Oracle database, because it does not support the queries in the *CHECK* constraints. The insertion experiment is described in Section 6.1.

The suggested realization by the triggers also requires additional select operations during the deletion of the order items to check whether there always remains at least one item for each order. Therefore, we made another experiment to compare the execution time of our proposed implementation by the triggers with the common implementation without such a validation. The experiment is described in Section 6.2.

On the other hand, the suggested realization by the views used to select only the valid data requires an additional condition evaluation during the selection. Therefore, we also made the experiments to compare the time of the selection

of the entries from the *Order* table directly, with the time of the selection using the view *valid_orders*. The experiment is described in Section 6.3.

Before each experiment and test variant, the database should contain from one to five order items for each of the already existing orders according to the following formula:

$$(orderID \ mod \ 5) + 1$$

items, where *OrderId* is the identifier of the order. To such a database, new orders and items are inserted, existing order items are deleted and existing orders are searched.

We used Oracle 10g XE database installed on Acer TravelMate 7730 (Intel(R) Core-(TM)-2 Duo CPU @ 2.00GHz with 2GB RAM, Windows 7 Professional 32-bit) for our experiments. The block size was set to 8kB and the database buffer was 52736 blocks.

6.1. The Insert Experiment

The experiment presents the time comparison of the process of inserting new entries for various implementations of a one-to-many relationship in a relational database. We developed several scripts for creating the database tables with the constraints and appropriate insert scripts for each of the implementation to simulate the process of inserting new entries into the database.

Table 1 presents the constraint implementation for each variant. The *Simple* variant is the standard implementation of one-to-many relationship with a primary key in both tables and a foreign key, which refers to the table *Order*, see Fig. 5. This variant does not restrict the minimal multiplicity for the items in the order. The *View* variant uses the view with the *CHECK OPTION* shown in Fig. 9 to insert new entries while checking whether there exist the items for this order. The *View with an index* variant uses the same view with the index defined in Fig. 17. The *Trigger* variant adds a trigger, as shown in Fig. 16, to check an existing item for each inserted order. In this variant, the trigger prevents inserting the orders without any items. Finally, the *Trigger with an index* variant adds the index on the orderID in the table *OrderItem*, as shown in Fig. 17, to speed up the search of items by their order.

Table 1. Variants of create scripts for various constraint realizations (+ implemented, * implemented deferrable, - not implemented)

Variant	primary keys	foreign key	index	trigger	view
Simple	+	+	-	-	-
View	+	*	-	-	+
View with index	+	*	+	-	+
Trigger	+	*	-	+	-
Trigger with index	+	*	+	+	-

The pseudo-SQL code of the insertion procedure for all the tested variants is given in Fig. 21. The script inserts several items with a reference to the inserted order. The number of the items of the same order differs in checking the options of inserting no, one or more items for the same order, respectively. While inserting and order, the number of its items is determined by the following formula:

$$(orderID \ mod \ 5).$$

The commit operation comes after each group of items of the same order to apply the constraint check. In the case of the *Simple* variant, the items are inserted after the order is inserted, because the FOREIGN KEY constraint is checked immediately, while in the case of other variants, the items are inserted before the order is inserted.
Fig. 22 presents the execution time of the insertion of 100 new orders for each of the variants in database already containing a various number of entries as described in the beginning of Section 6.

As we can see, the *Simple* variant proved that the execution time is nearly independent on the data already stored in the database since there are no constraints to check during the insertion. However, the optionality of the order items for each of the orders is not checked and even the orders without any items are inserted. The *Trigger* variant enforces only valid orders with at least one item to be inserted. However, the constraint check slows down the evaluation when more entries already exist in the tables. The *Trigger with the index* variant proved to be able to eliminate this problem and to be even faster than the *Simple* variant. Similar results were measured for the view implementations. The *View* variant became even slower than the *Trigger* variant because of checking the view condition after trying to insert new data. However, the *View with the index* variant eliminates the slowdown by the index and is almost equivalent to the *Trigger with the index* variant. All the measured data is summarized in Table 2.
The *strange* decrease of the time required for the insertion of data to the database containing 10000 and 100000 records in the *Simple* method is probably caused by the checkpoint processing. In the Oracle database, records are stored in data blocks in the buffer cache and the checkpoint process synchronizes the buffer cache with the data blocks in the persistent storage – usually data files. Also, for each experiment run, we delete the records inserted in the last run to insert the new data in the same database state. Therefore, some data blocks are loaded to the buffer cache just before the insert starts. Then, when inserting into a small database, there is only a few of data blocks is available to insert the data and the checkpoint process blocks the insertion when the blocks are locked for synchronization. On the other hand, in the large database, a lot of blocks is available in the buffer cache that are not locked by the checkpoint process and thus are available for insertion.
However, this applies only for the *Simple* variant as there are no special constraints aside the PRIMARY KEY that need to be checked and which cause the serialization of data access. Additionally, in all the variants except the *Sim-*

```
CREATE PROCEDURE insert_values (p_orders_count)
IS
BEGIN
  l_count := 0;
  SELECT COALESCE(MAX(orderItemID)+1,1)
   INTO l_items_count FROM OrderItem;
  SELECT COALESCE(MAX(orderID)+1,1)
   INTO l_orders_count FROM Order;
  l_starting_orders_count := l_orders_count - 1;

  FOR l_iter IN 1..p_orders_count
  LOOP
    INSERT INTO Order (orderID, order_date, paid)
    VALUES (l_orders_count, sysdate, 'N');
    COMMIT;

    FOR l_iter2 IN 1..l_count
    LOOP
      INSERT INTO OrderItem
      (orderItemID, orderID, name, price, quantity)
      VALUES (
        l_items_count, l_orders_count,
        'item' || l_iter2, mod(l_orders_count, 10)+1,
        mod(l_orders_count, 20)+1);

      l_items_count := l_items_count + 1;
    END LOOP; -- insert items
    COMMIT;

    l_orders_count := l_orders_count + 1;
    l_count := mod (l_count + 1, 5);
  END LOOP; -- insert order
END; -- insert_values
```

Fig. 21. Pseudo-SQL code of experimental insert scripts

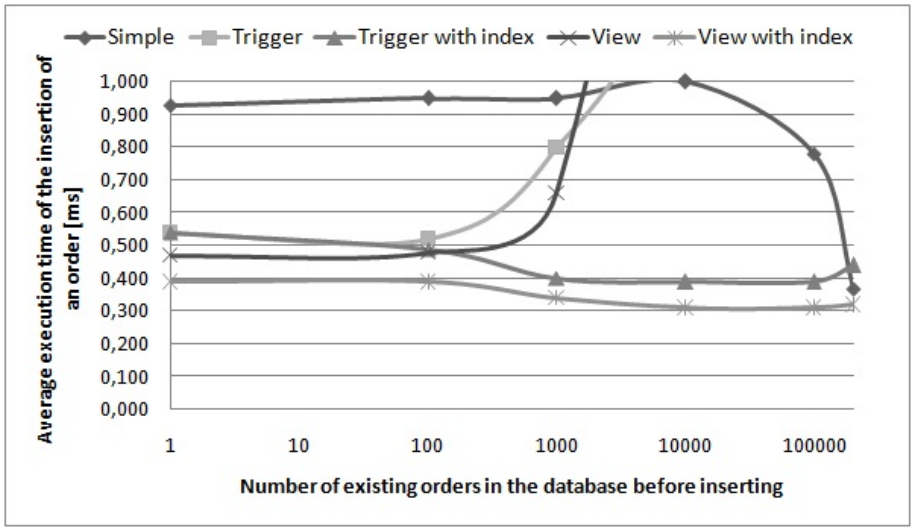

Fig. 22. Execution time of insertion of new entries for various implementation variants

ple variant the FOREIGN KEY constraint is deferrable. This causes the constraint to be checked at the end of the transaction and therefore it requires post-processing that eliminate the advantage of many available blocks in the buffer cache.

Table 2. The results of the insertion experiment - execution times of new entries insertion for various implementations in milliseconds.

Number of entries	Simple	Trigger	Trigger with index	View	View with index
0	0.930	0.540	0.540	0.470	0.390
100	0.950	0.520	0.490	0.480	0.390
1000	0.950	0.800	0.400	0.660	0.340
10000	1.000	1.590	0.390	3.150	0.310
100000	0.780	14.510	0.390	29.000	0.310
200000	0.370	28.980	0.440	57.870	0.320

Also note that the PRIMARY KEY value is generated in a sequence. If it is generated randomly, the insert would take more time as the correct data block would be needed to be loaded to the buffer cache to insert the record in the correct place according to the PRIMARY KEY value. It would especially affect the *Simple* variant in large databases where the execution time would not decrease.

6.2. The Delete Experiment

The delete experiment compares the execution time of the delete operations on the table *OrderItem*. If the source entity optionality constraint is realized by the trigger as defined in Fig. 18, the DELETE operation requires to select the orders and its items to check if the deleted item was not the last one, and thus making the order invalid. This constraint check slows down the DELETE operation as demonstrated by this experiment.

Table 3. Variants of deletes executed and measured (+ implemented, - not implemented)

Variant	Trigger	Index
Simple	-	-
Trigger	+	-
Trigger with index	+	+

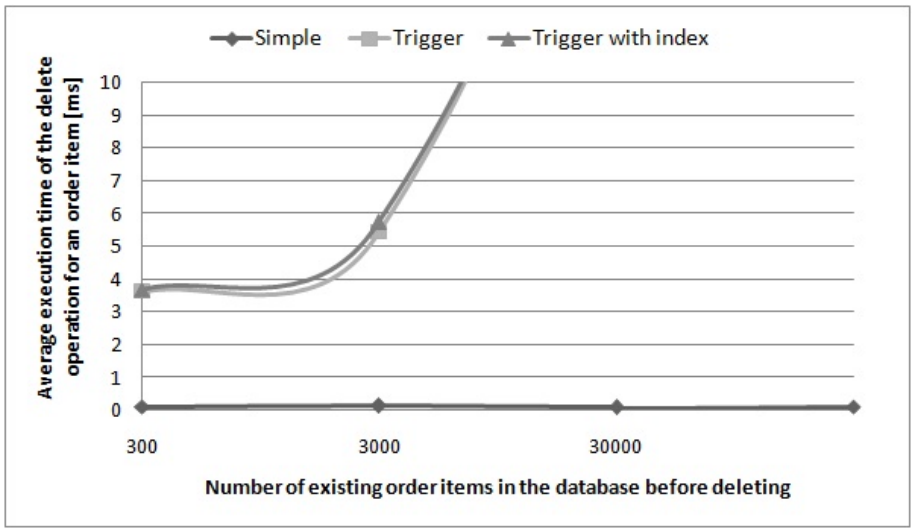

Fig. 23. Execution time of the deletion of the order items for various implementation variants

Three various implementations were tested to compare. The *Simple* variant represents the delete operations on the *OrderItem* table directly without a constraint realization. The *Trigger* variant represents the delete operations with the trigger defined on the *OrderItem* table as shown in Fig. 18. In this variant, only such items are deleted that do not violate the source entity optionality constraint

of the orders. The *Trigger with the index* variant uses the same trigger. However, in this variant, the index is defined as shown in Fig. 17 to speed up the search of the items by the order identifier. All the variants are summarized in Table 3.

Fig. 23 presents the execution time comparison for the deletion of the last 100 order items inserted to the database by its *OrderItemID* attribute executed for various number of orders and items existing in the database. Before such a test, the database contains the data as described in the beginning of Section 6.

As we can see, the *Simple* variant is the fastest, since there are no constraints to check when deleting the order items. However, the required optionality of the order items for each order is not checked and the orders without any item can appear in the database. It violates the *source* table optionality constraint. The *Trigger* variant prevents from deleting the last item of an order, however, the execution time is much slower, especially if more orders and order items are stored in the database. Even the index does not help because it is not used in the checking *SELECT* operation in the trigger while joining orders and its items. The measured data is summarized in the Table 4.

Table 4. The results of the deletion experiment - execution times of deletion of order items for various implementations in milliseconds.

Number of order items	Simple	Trigger	Trigger with index
300	0,11	3,63	3,69
3000	0,14	5,45	5,74
30000	0,09	26,45	26,33
300000	0,13	256,76	256,72

6.3. The Select Experiment

This experiment presents a comparison of the execution time of a *SELECT* operation from the table *Order* directly, and by the view for accessing the valid data only. The *SELECT* statement is shown in Fig. 24, where *X* is a random order identifier. It searches for an order by its *OrderID*. No other conditions were measured because we compared the effect of the source entity optionality constraint check for the selected data. Therefore the more data is stored in the database, the slower the *SELECT* statement is.

```
SELECT * FROM Order WHERE OrderID = X;
```

Fig. 24. The SELECT statement for the selection experiment

Three various implementations were measured for each selection. The *Simple* variant presents the selection from the table *Order* directly without checking the

constraint. The *View* variant presents the selection from the view *valid_orders* defined over the table *Order* to check the existing entries in the *Order* table and to select only from valid orders. The *View with the index* variant presents the selection from the view *valid_orders* defined over the table *Order* with the index defined on the order identifier in the table *OrderItem* to speed up the search of the items of the order while checking the existence. All select variants are summarized in Table 5.

Fig. 25 presents the results of the experiment. It shows the execution time of each variant for various number of orders stored in the *Order* table together with associated items in the *OrderItem* table as described in the beginning of Section 6.

The *Simple* variant proved to be the fastest variant – as expected – since there is no additional condition to check during the selection. However, the query returns back both valid and invalid data according to the source entity optionality constraint. The *View* variant results become much slower when more entries are stored in the tables, because of an additional constraint with a subquery for checking the valid orders. However, only valid orders according to the source entity optionality constraint are returned back. The index defined over the foreign key value in the *OrderItem* table speeds up the subquery execution rapidly as shown by the *View with index* variant results. Therefore, the *View with the index* variant seems to be nearly equivalent to the direct selection from the table in the execution time. However, the *View with the index* provides only the valid data. The measured data of the experiment is summarized in Table 6.

Table 5. Variants of selects executed and measured

Variant	Source	Index in the OrderItem table
Simple	table Order	not defined
View	view valid_orders	not defined
View with index	view valid_orders	defined

Table 6. The results of the selection experiment - execution times of *SELECT* operations for various implementations in seconds

Number of entries	Simple	View	View with index
100	0.002	0.002	0.003
1000	0.000	0.000	0.000
10000	0.000	0.002	0.000
100000	0.004	0.009	0.014
500000	0.039	0.061	0.031
1000000	0.050	0.519	0.039
2000000	0.054	3.091	0.041

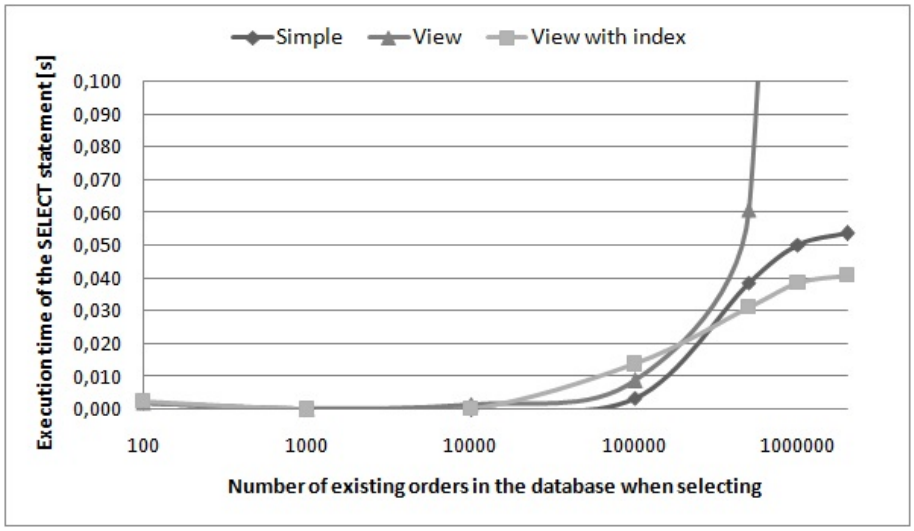

Fig. 25. Execution time of selection of entries for various implementation variants

7. Conclusions

In this paper, we summarized the currently used method for modeling binary associations in the data models using UML class diagrams. We showed the way to specify multiplicity constraints in the model. Furthermore, we showed a usual transformation of the model from PIM to PSM for the relational database and the usual transformations for multiplicity constraints using *FOREIGN KEY*, *NOT NULL* and *UNIQUE* constraints in SQL.

We pointed out the constraint for the source entity optionality. This constraint is often used in the model but not realized in the database because the foreign key is insufficient instrument for full implementation. Therefore, we defined this constraint in another formal way by an OCL invariant and suggested several methods how this constraint can realized in a relational database.

We also compared the suggested implementations to the currently used approaches in the context of the execution time while inserting new data to the tables, deleting data from the tables and selecting existing data from the tables. The experiments showed that the trigger realization and the view realization slow down the insertion of new data the more rapidly the more data has been stored in the tables. However, when the index is defined in the referring table, this slowdown is eliminated and the insertion is even faster. The results also showed that selecting the data using the view with the index on the FOREIGN KEY column is equivalent in the execution time to the direct access while providing only the valid data. However, when trying to check the *source* table optionality constraint by the trigger when deleting the data, the trigger implementation showed to be very slow even with the defined index.

According to the experiment results, we suggest the constraint should be realized in CASE tools' transformations of data models to relational databases either by the trigger or the view with the check option to prevent it from inserting invalid data or by the view to filter invalid data from the selection. However, the realization of the constraint check for the delete and update operations should be objectives of the future research to be able to fully prevent the invalid data being present in the database. We also believe that the integration of the suggested realizations in the transformation processes of CASE tools may save a lot of effort of analysts and database designers when trying to design a consistent database and even improve the database consistency as this effort is usually neglected.

Acknowledgments. We would like to thank for financial support of Student Grant Competition of CTU in Prague, grant number SGS13/099/OHK3/1T/18 and also to AVAST Foundation in Prague.

References

1. Aleksić, S., Ristić, S., Luković, I.: An approach to generating server implementation of the inverse referential integrity constraints. In: Proceedings. AL-Zaytoonah University of Jordan, Amman, Jordan (May 2011)
2. Arlow, J., Neustadt, I.: UML 2.0 and the Unified Process: Practical Object-Oriented Analysis and Design (2nd Edition). Addison-Wesley Professional (2005)
3. Cabot, J., Teniente, E.: Constraint support in MDA tools: A survey. In: Rensink, A., Warmer, J. (eds.) Model Driven Architecture Foundations and Applications. Lecture Notes in Computer Science, vol. 4066, pp. 256–267. Springer Berlin / Heidelberg (2006), http://www.springerlink.com/content/4902321654674181/abstract/
4. Demuth, B.: DresdenOCL. http://www.reuseware.org/index.php/ DresdenOCL (Jan 2011)
5. Luković, I., Mogin, P., Pavićević, J., Ristić, S.: An approach to developing complex database schemas using form types. Software: Practice and Experience 37(15), 16211656 (Dec 2007), http://dx.doi.org/10.1002/spe.v37:15
6. Melton, J.: Advanced SQL:1999. Morgan Kaufmann Publishers (2003)
7. OMG: Object constraint language, version 1.3. http://www.omg.org/spec/OCL/2.2/PDF (Feb 2010)
8. OMG: Object management group. http://www.omg.org/ (Dec 2011)
9. OMG: UML 2.4.1. http://www.omg.org/spec/UML/2.4.1/ (Aug 2011), http://www.omg.org/spec/UML/2.4.1/
10. OMG, Miller, J., Mukerji, J.: MDA guide version 1.0.1. http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf (Jun 2003)
11. Richta, K., Rybola, Z.: Transformation of relationships from UML/OCL to SQL. In: ITAT 2011: Zborník príspevkov prezentovaných na konferencii ITAT. vol. 11. University of P. J. Šafárik, Košice, Slovakia, Terchová, Slovakia (Sep 2011), http://itat.ics.upjs.sk/proceedings/itat2011-zbornik.pdf
12. Rob, P., Coronel, C.: Database Systems: Design, Implementation, and Management. Boyd & Fraser, 2nd edn. (1995)

13. Rybola, Z., Richta, K.: Transformation of binary relationship with particular multiplicity. In: DATESO 2011. vol. 11, pp. 25–38. Department of Computer Science, FEECS VSB - Technical University of Ostrava, Písek, Czech Republic (Apr 2011), http://www.informatik.uni-trier.de/~ley/db/conf/dateso/dateso2011.html
14. Rybola, Z., Richta, K.: Transformation of special multiplicity constraints - comparison of possible realizations. In: Proceedings of the Federated Conference on Computer Science and Information Systems. pp. 1357–1364. FedCSIS, Wroclaw, Poland (Sep 2012)
15. Softeam: Objecteering/UML. http://www.softeam.com/technologies_objecteering.php (Dec 2012)
16. Sparx Systems: Enterprise architect - UML design tools and UML CASE tools for software development. http://www.sparxsystems.com.au/products/ea/index.html (Mar 2011)
17. Wilke, C., Thiele, M., Freitag, B.: Dresden OCL: manual for installation, use and development (Oct 2010)

Zdeněk Rybola is a PhD. student and an assistant professor at the Department of Software Engineering at the Faculty of Information Technology, Czech Techical University in Prague. His area of interest includes Model Driven Development in context of relational databases and multiplicity constraints and the usage of OntoUML in software engineering.

Karel Richta is an associate professor at the Department of Software Engineering at the Faculty of Mathematics and Physics, Charles University in Prague, and also at the Department of Computer Science and Engineering at the Faculty of Electrical Engineering, Czech Technical University in Prague. His research is primarily focused on formal specifications and similar approaches usable in software engineering. He has published more than 100 publications, including 5 books. He is the president of Czech ACM Chapter.

Received: December 10, 2012; Accepted: September 2, 2013.