Communication Papers of the 20th Conference on Computer Science and Intelligence Systems (FedCSIS) pp. 85–93
DOI: 10.15439/2025F7149
ISSN 2300-5963 ACSIS, Vol. 45

Towards OntoUML for Software Engineering: Transformation of Constraints into Various Relational Databases

Jakub Jabůrek, Zdeněk Rybola and Petr Kroha
ORCID: 0009-0004-8212-2059, 0000-0001-9430-6921, 0000-0002-1658-3736
Faculty of Information Technology, Czech Technical University in Prague
Thákurova 9, 160 00 Praha 6, Czech Republic
Email: {jakub.jaburek, zdenek.rybola, petr.kroha}@fit.cvut.cz

***Abstract*—OntoUML is an ontologically well-founded conceptual modeling language that provides precise meaning to modeled elements. As a result, its usage is beneficial in the Model-Driven Development approach to software development. Relational databases are commonly used for storage of application data, and they offer support for the implementation of custom data constraints. In this paper, we discuss the realization of constraints that arise from OntoUML structural models in PostgreSQL, Microsoft SQL Server and MySQL, and provide a complete reference on how to implement these constraints so that only data conforming to the OntoUML model can be stored.**

***Index Terms*—Conceptual Modeling, Software Development, Relational Database, Model Transformation, Constraints.**

I. Introduction

SOFTWARE engineering is a demanding discipline that deals with complex systems. The goal of software engineering is to ensure high quality of the implementation of software systems. Various approaches to software development exist to achieve this.

In this paper, we focus on the Model-Driven Development (MDD) approach. It is based on the construction of models of the software and their transformations. An established practice within MDD is *forward engineering* — transformation of abstract models into more concrete ones [1]. One of the common use cases of such process is the transformation of a conceptual data model into application source code or database schema.

In MDD, the quality of the final implementation of the system depends on the accuracy of the transformation process as well as on the precision of the initial models. Therefore, a conceptual data model should capture the necessary constraints, and the transformation should preserve them in the transformed result.

For the conceptual data model, we use OntoUML — as it is based on cognitive science and modal logic [2], we consider it to be suitable for the development of highly expressive data models. In comparison with other modeling languages, such as the Entity–relationship model, OntoUML enforces stronger semantic constraints, therefore resulting in models that more closely follow reality.

For the implementation target, we focus on relational database management systems (RDBMS), as they are one of the most commonly used data persistence platforms [3]. As different RDBMS implementations have varying levels of support for the definition and enforcement of data constraints, the implementation is specific to the particular RDBMS product.

The existing literature describes the transformation process of an OntoUML model into Oracle RDBMS [4]. This transformation is divided into three successive steps:

1) First, the OntoUML model is transformed into a Unified Modeling Language (UML) class model.
2) Second, the UML model is transformed into a relational data model.
3) Finally, the relational data model is transformed to an implementation in the Structured Query Language (SQL) tailored to a specific RDBMS.

Our approach follows the aforementioned division and reuses transformation steps 1–2. The original contribution of this paper consists of novel implementation of the third transformation step, which is RDBMS-vendor-specific. While the transformation of a relational model to SQL is well-known, our approach focuses on preserving the integrity constraints defined by the original OntoUML model.

We elaborate implementations in MySQL, Microsoft SQL Server and PostgreSQL, while the realization in Oracle is described in existing literature [4].

This practically oriented paper is structured as follows: In Section II, OntoUML concepts relevant to the transformation are summarized, the features of SQL related to constraint checking are summarized, and the existing transformation of an OntoUML model to SQL is introduced. In Section III, we discuss constraint checking features supported in the considered RDBMS. In Section IV, we elaborate the implementation of constraints in SQL for each of the considered RDBMS. In Section V, we discuss the limitations of our approach. Finally, in Section VI, the results of this paper are summarized.

II. Background and Related Work

In this section, OntoUML, being the language of the input models of our transformation, is introduced. Then, SQL, the

Topical area: Software, System and Service Engineering

target language of the transformation, is discussed. Finally, the existing transformation of OntoUML to relational schema, which our transformation to SQL builds upon, is introduced.

A. OntoUML

OntoUML is a conceptual modeling language focused on building ontologically well-founded structural models [5]. It is a profile of the Unified Modeling Language (UML) Class Diagram that realizes the Unified Foundational Ontology (UFO) theory formulated in G. Guizzardi's Ph.D. thesis [6].

An OntoUML model defines *universals* — bundles of characteristics shared among their instances. Universals are instantiated by *individuals*. UFO differentiates between universals of *Sortal* and *Non-Sortal* type. Sortals bear an *identity principle* (either they provide it themselves or inherit it from a Sortal supertype), which defines how individual instances of the particular universal are distinguished [7].

A single individual may instantiate multiple universals, and the set of universals instantiated by a particular individual may change over time. Universal types that an individual must instantiate during its entire lifetime (or not instantiate at all) are classified as *Rigid*. Conversely, universal types that an individual may start to instantiate later or cease to instantiate prior to its own destruction are classified as *Anti-Rigid* [8].

UFO defines a taxonomy of universal types, with the concrete types then being applied as stereotypes to classes in an OntoUML model. In this section, we summarize a selection of universal types that significantly contribute to the set of constraints derived during the transformation of an OntoUML model to a relational schema, as introduced in Section II-C.

The backbone of an OntoUML model consists of *Kinds* and *SubKinds* — Rigid Sortals, with SubKinds inheriting the identity principle from Kinds through the generalization relationship.

An example is provided in Fig. 1. The model describes Kind `Document` with two SubKinds `IdCard` and `Passport`. The generalization set is covering and disjoint — an instance of `Document` must be at the same time an instance of exactly one of the subclasses. A `Person` Kind is also modeled. It can have any number of associated `IdCard` instances, but an `IdCard` must always be associated with exactly one `Person`. A `Person` must be associated with at least two but not more than four `Passport` instances, while a `Passport` must always be associated with exactly one `Person`. Also in the example, a `Brain` Kind is present, whose instance must always be associated with one `Person` instance (and vice versa). The particular instances in the association relationship may not change over time.

Next, an OntoUML model may define *Roles* and *Phases*, which are classified as Anti-Rigid Sortals [8]. Roles are relationally dependent and must be connected to at least one *Mediation* relationship (with a *Relator* universal at the other end) [9], while Phases are intrinsic and not relationally-dependent. A Phase must always be a part of a *phase partition* (a disjoint and covering generalization set) [10].

In the example OntoUML model in Fig. 1, the `Person` Kind has two Phases: `Alive` and `Deceased`, exactly one of them must always be instantiated.

More universal and relationship types are defined in UFO; their description is out of scope of this paper. For more detailed definitions, the reader is referenced to literature covering the UFO theory [6] [7] [8]. Thanks to the constraints placed by UFO on the types and relationships in an OntoUML model, OntoUML models are able to capture many domain constraints natively, as opposed to other modeling languages, such as plain UML, in which such constraints must be specified by other means.

The example OntoUML model in Fig. 1 is used throughout this paper to illustrate the realization of constraints in SQL. The model is used as the input to the transformation introduced in Section II-C, where the transformation result is also shown.

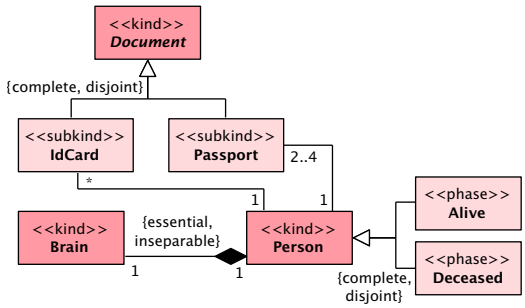

Fig. 1. OntoUML model of the running example

B. Structured Query Language

Structured Query Language (SQL) is a language used to manage data in RDBMS. In this paper, we deal with two commonly recognized parts of SQL [11]:

- Data Manipulation Language (DML) — querying, inserting, modifying and deleting data in the database,
- Data Definition Language (DDL) — definition of tables and integrity constraints.

In this paper, we focus on the realization of constraints, which pertains to the DDL part of SQL. Although SQL is defined by an ISO standard, not all SQL statements are universally portable across different RDBMS vendors. While SQL syntax is consistent across RDBMS, the set of supported features for integrity constraint checking varies; therefore not all DDL constructs defined in the SQL standard can be used in every RDBMS implementation.

As described in existing literature and in the SQL standard, the following mechanisms can be used to realize integrity constraints in a relational database (mechanisms not implemented in any of the mainstream RDBMS are omitted) [12]:

- PRIMARY KEY — column that contains a unique identifier for each row in a table,
- UNIQUE — column that must have a unique value across all rows in a table,
- FOREIGN KEY — column that references an existing record in another table,

- CHECK — expression that realizes a row-level constraint,
- TRIGGER — procedure that runs before or after a DML operation, and can change or block the statement.

Additionally, SQL defines two modes of constraint checking: *immediate* and *deferred*. In immediate mode, a constraint is checked at the end of each DML statement. In deferred mode, the constraint is checked at the end of the SQL transaction [12].

C. Transformation of OntoUML into SQL

The process of transforming an OntoUML into SQL has been elaborated in the Ph.D. thesis of Z. Rybola [4], and was further discussed in [13]. In this subsection, we offer a high-level overview of their approach, which is divided into three successive steps:

1) Transformation of OntoUML into UML. First, the OntoUML model is transformed into UML with added conditions written in the Object Constraint Language (OCL) to preserve the constraints defined by the original OntoUML model [4].

2) Transformation of UML into a Relational Model. Second, the UML model is transformed into a relational schema by using well-known algorithms. In addition, the OCL constraints from the previous step are transformed as well to preserve the meaning of the original OntoUML model.

Concerning the transformation of the generalization relationship between classes into the relational model, various approaches are described in the existing literature [14]. In this paper, we focus on the *related tables* approach, where each class in the hierarchy is transformed to a separate table.

To illustrate the realization of constraints in SQL in Section IV, a relational model shown in Fig. 2 is used. This model is the result of the aforementioned transformation applied to the OntoUML model in Fig. 1. Due to limited space, the derived OCL constraints are omitted.

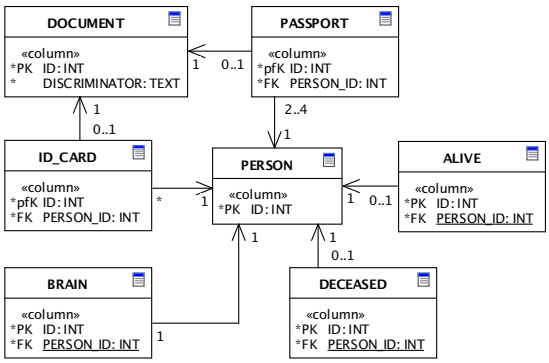

Fig. 2. Running example transformed into a relational model

3) Transformation of the Relational Model into SQL. Finally, the relational model is transformed into SQL. While the transformation of tables and columns alone is straightforward, the implementation of OCL constraints from the previous step is non-trivial. The existing literature describes the realization in Oracle [4]. However, due to differences in constraint checking mechanisms (as discussed in Section III), a realization in one RDBMS is generally not portable to another.

TABLE I
CONSTRAINT CHECKING SUPPORT MATRIX

Constraint	Oracle	MySQL	MSSQL	PostgreSQL
FOREIGN KEY				
deferrable	imm. / defer.	immediate	immediate	imm. / defer.
TRIGGER				
# of events	multiple	one	multiple	multiple
activation	before / after	before / after	after	before / after
granularity	row / statem.	row	statement	row / statem.

III. Relational Database Management Systems

According to the DB-Engines Ranking of Relational DBMS, the top four RDBMS (as of April 2025) are the following [15]:

1) Oracle
2) MySQL
3) Microsoft SQL Server (MSSQL)
4) PostgreSQL

As discussed in Section II-B, the support for integrity constraint checking is not uniform across various RDBMS. The four vendors listed above implement at least some forms of PRIMARY KEY, UNIQUE, FOREIGN KEY, CHECK and TRIGGER constraints.

We surveyed the features supported by the top four RDBMS important for the realization of constraints derived from an OntoUML model — the results are presented in table I.

Oracle 23ai and PostgreSQL 17 offer complete support for all surveyed constraint types. MySQL 9.3 supports only one activation event per trigger, the expressiveness is however equivalent to RDBMS that support any number of triggering events. Also, MySQL supports only row-level triggers. Conversely, MSSQL 2022 implements only statement-level triggers, and they can be executed only after the DML statement.

MySQL and MSSQL offer immediate constraint checking only. For the FOREIGN KEY constraint, both RDBMS offer mechanisms to temporarily disable integrity checks. MySQL however does not check existing data for consistency when the constraint is re-enabled, therefore referential integrity may be broken indefinitely.

IV. Implementation of Constraints in SQL

In this section, we present the main contributions of our paper: the implementation of constraints derived during the transformation of an OntoUML model to a relational schema in SQL for MySQL, Microsoft SQL Server and PostgreSQL.

In our approach, an OntoUML model is not transformed to SQL directly. Instead, we rely on the existing transformation of an OntoUML model to a relational data model, as discussed in Subsection II-C:

1) First, the OntoUML model is transformed into UML. Additional constraints are derived so the semantics of the original model are preserved where the UML class model alone is not sufficient.

2) Second, the UML model is transformed into a relational data model. The derived constraints are transformed as well to apply to the relational model.
3) Finally, the relational model is transformed into SQL, including the derived constraints. Based on the transformed constraints, the RDBMS prevents the creation of data that do not conform to the original OntoUML model.

The existing literature provides the implementation of constraints in Oracle [4]. As surveyed in Section III, the top four RDBMS implement varying features related to constraint checking. Therefore, the realization of constraints in SQL is dependent on the particular RDBMS product.

In this section, we thoroughly elaborate the realization of all constraints that can be derived from an OntoUML model in MySQL, Microsoft SQL Server (MSSQL) and PostgreSQL. The section is structured as follows: in Subsection IV-A, we first discuss a common approach to the realization of association-related constraints, then, in the remaining subsection, we discuss the realization of individual constraints. We illustrate the implementation by SQL listings applicable to the relational model in Fig. 2. The model was produced by the transformation introduced in Section II-C from the OntoUML model presented in Fig. 1.

A. Associations

In this subsection, we introduce the common approach for the implementation of constraints related to associations that is used throughout the entire Section IV.

For constraints involving associations, we often need to ensure that a referencing record exists. In OCL, this requirement is expressed as an invariant on the referenced table. The only invariant-like generic constraint implemented by RDBMS is the CHECK constraint, which cannot contain subqueries, therefore it cannot be used to implement this requirement in SQL. Instead, a set of triggers must be introduced, which are executed on all DML operations that may cause the invariant to not hold.

Notably, an INSERT operation on the referenced table without a corresponding INSERT operation on the referencing table would produce a non-referenced record. With triggers, this can be prevented only by checking the existence of the referencing record at the time of the insertion of the referenced record. Therefore, an additional restriction is placed on the order of DML operations: the referencing record must be inserted before the referenced record.

For PostgreSQL, the implication is that the FOREIGN KEY constraint in the referencing table must be checked in deferred mode. MySQL and MSSQL do not implement deferred constraint checking. In our approach, for MSSQL, we propose temporarily disabling the FOREIGN KEY constraint and re-enabling it after both records are inserted. MySQL however does not check the integrity of existing data when the constraint is re-enabled. To not break referential integrity, we present a different approach, where the client is required to manually execute a stored procedure that checks the existence of the referencing record once both sides are inserted.

In similar fashion, the order of DELETE operations is enforced to be either (a) the referenced record before the referencing record in PostgreSQL and MSSQL, or (b) the referencing record before the referenced record in MySQL.

B. Immutability

In OntoUML models, *immutability constraints* commonly emerge from relationships between a Sortal that provides an identity principle and its subtypes. In SQL, the implementation depends on whether an entire association or only a single column needs to be guarded by such constraint, as discussed in this subsection.

1) Immutable Column. To ensure that the value in a column remains unchanged, a BEFORE UPDATE row-level trigger must be defined in MySQL and PostgreSQL that blocks the operation if the old and new column values are different. An example trigger that prevent the change of the `PERSON_ID` column in the `BRAIN` table is implemented for MySQL in Listing 1, implementation in PostgreSQL is similar.

Listing 1 Realization of the immutability constraint in MySQL

```
CREATE TRIGGER `IM_BRAIN_PERSON_ID_UPD`
BEFORE UPDATE ON `BRAIN` FOR EACH ROW BEGIN
IF OLD.`PERSON_ID` <> NEW.`PERSON_ID` THEN
  SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = "...";
END IF; END;
```

As MSSQL does not support row-level triggers, a different approach needs to be employed. The statement-level trigger in Listing 2 uses the `UPDATE` function to short-circuit the condition in case the guarded column was not a part of the UPDATE operation. However, the function returns TRUE when updating to the same value. Therefore, the trigger needs to query `inserted` and `deleted` special tables, which contain the new and old values, respectively (w.r.t. the UPDATE operation). To determine whether the value of the guarded column actually changed, the two tables are joined on the PRIMARY KEY column and then the new and old values of the column are compared. Note that the lack of row-level triggers in MSSQL makes it impossible to detect a change in the PRIMARY KEY column, also such a change breaks the change detection for other columns in the table as well.

Listing 2 Realization of the immutability constraint in MSSQL

```
CREATE TRIGGER [IM_BRAIN_PERSON_ID_UPD]
ON [BRAIN] AFTER UPDATE AS BEGIN
IF UPDATE([PERSON_ID]) AND EXISTS (
    SELECT 1 FROM inserted i
      JOIN deleted d ON i.[BRAIN_ID] = d.[BRAIN_ID]
      WHERE i.[PERSON_ID] <> d.[PERSON_ID]) BEGIN
  THROW 50000, '...', 1; END; END;
```

2) Immutable Association. When the referenced side (i.e., opposite the table containing the FOREIGN KEY) is marked as immutable, the realization in SQL is the same as the previously discussed *immutable column* constraint. The constraint

is applied to the column in the referencing table that bears the FOREIGN KEY constraint. The deletion of the referenced record is prevented by the FOREIGN KEY constraint itself.

In case the referencing side is marked as immutable, the FOREIGN KEY column is immutable with the same realization as in the previous case. Additionally, the deletion of the referencing record must be prevented. In MySQL and PostgreSQL, this is realized by BEFORE DELETE row-level triggers that block the operation when an existing record is referenced by the referencing record (see Listing 3 for MySQL, implementation in PostgreSQL is similar). Again, in MSSQL, the trigger must be statement-level and it must query the `deleted` special table (see Listing 4).

Listing 3 Trigger preventing the deletion of a referencing record in MySQL

```
CREATE TRIGGER `IM_BRAIN_PERSON_ID_DEL`
BEFORE DELETE ON `BRAIN` FOR EACH ROW BEGIN
IF OLD.`PERSON_ID` IS NOT NULL AND EXISTS (
    SELECT 1 FROM `PERSON` `p`
      WHERE `p`.`ID` = OLD.`PERSON_ID`) THEN
  SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = "...";
END IF; END;
```

Listing 4 Trigger preventing the deletion of a referencing record in MSSQL

```
CREATE TRIGGER [IM_BRAIN_PERSON_ID_DEL]
ON [BRAIN] AFTER DELETE AS BEGIN
IF EXISTS (SELECT 1 FROM deleted d
    JOIN [PERSON] p ON d.[PERSON_ID] = p.[ID]) BEGIN
  THROW 50000, '...', 2;
END; END;
```

C. Exclusive Associations

The *exclusive associations constraint* is derived from OntoUML phase partitions [16]. It guards a set of referencing tables where a record in only one of the referencing tables can refer a record in the referenced table. It follows that the following five DML operations may violate the constraint:

1) INSERT or UPDATE on the referenced table — the number of referencing records is not exactly one,
2) INSERT to any of the referencing tables — the referenced record is already referenced,
3) UPDATE on any referencing table — the referencing record may get de-associated or attached to a record that already has an association,
4) DELETE from any referencing table — an orphan may be created in the referenced table.

To ensure exactly one referencing record exists when the referenced record is being inserted, a BEFORE INSERT/UPDATE trigger is employed. An example trigger that guards the `PERSON` referenced table and `ALIVE` and `DECEASED` referencing tables is provided in Listings 5 and 6 for PostgreSQL and MSSQL, respectively. As MSSQL does not support BEFORE nor row-level triggers, an AFTER trigger that queries the `inserted` special table is used instead.

Listing 5 INSERT/UPDATE trigger on the referenced table realizing an exclusive associations constraint in PostgreSQL

```
CREATE FUNCTION ex_person_phase()
RETURNS TRIGGER AS $$ BEGIN
IF NOT (
      (EXISTS (SELECT 1 FROM "ALIVE" a
          WHERE a."PERSON_ID" = NEW."ID")
       AND NOT EXISTS (SELECT 1 FROM "DECEASED" d
          WHERE d."PERSON_ID" = NEW."ID")) OR
      (NOT EXISTS (SELECT 1 FROM "ALIVE" a
          WHERE a."PERSON_ID" = NEW."ID")
       AND EXISTS (SELECT 1 FROM "DECEASED" d
          WHERE d."PERSON_ID" = NEW."ID"))) THEN
  RAISE EXCEPTION '...';
END IF; RETURN NEW; END; $$ LANGUAGE plpgsql;
CREATE TRIGGER ex_person_phase
BEFORE INSERT OR UPDATE ON "PERSON" FOR EACH ROW
EXECUTE FUNCTION ex_person_phase();
```

Listing 6 INSERT/UPDATE trigger on the referenced table realizing an exclusive associations constraint in MSSQL

```
CREATE TRIGGER [EX_PERSON_PHASE]
ON [PERSON] AFTER INSERT, UPDATE AS BEGIN
IF EXISTS (SELECT 1 FROM inserted i WHERE NOT (
      (EXISTS (SELECT 1 FROM [ALIVE] a
          WHERE a.[PERSON_ID] = i.[ID])
       AND NOT EXISTS (SELECT 1 FROM [DECEASED] d
          WHERE d.[PERSON_ID] = i.[ID])) OR
      (NOT EXISTS (SELECT 1 FROM [ALIVE] a
          WHERE a.[PERSON_ID] = i.[ID])
       AND EXISTS (SELECT 1 FROM [DECEASED] d
          WHERE d.[PERSON_ID] = i.[ID])))) BEGIN
  THROW 50000, '...', 1; END; END;
```

As discussed in Subsection IV-A, a manually invoked procedure must be used in MySQL. The procedure, as illustrated in Listing 7, needs to be executed by the client after inserting or updating a record in the referenced table, but only after the referencing record is inserted (if applicable). The `ID` input parameter shall contain the primary key of the inserted record.

Listing 7 Procedure on the referenced table realizing an exclusive associations constraint in MySQL

```
CREATE PROCEDURE `EX_PERSON_PHASE` (IN `ID` INT)
BEGIN
IF NOT (
      (EXISTS (SELECT 1 FROM `ALIVE` a
          WHERE a.`PERSON_ID` = `ID`)
       AND NOT EXISTS (SELECT 1 FROM `DECEASED` d
          WHERE d.`PERSON_ID` = `ID`)) OR
      (NOT EXISTS (SELECT 1 FROM `ALIVE` a
          WHERE a.`PERSON_ID` = `ID`)
       AND EXISTS (SELECT 1 FROM `DECEASED` d
          WHERE d.`PERSON_ID` = `ID`))) THEN
  SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = "...";
END IF; END;
```

Furthermore, the constraint may get violated when a record is inserted into a table that belongs to the mutually exclusive set. Therefore, a BEFORE INSERT trigger is in place for all tables in the set that prevents the operation if a matching referencing record already exists in any of the other tables. An example of such trigger is provided in Listing 8 for MySQL, implementation in PostgreSQL is similar.

Listing 8 INSERT trigger on a referencing table realizing an exclusive associations constraint in MySQL

```
CREATE TRIGGER `EX_PERSON_PHASE_ALIVE_INS`
BEFORE INSERT ON `ALIVE` FOR EACH ROW BEGIN
IF EXISTS (SELECT 1 FROM `DECEASED` d
    WHERE d.`PERSON_ID` = NEW.`ID`) THEN
  SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = "...";
END IF; END;
```

As MSSQL does not support BEFORE nor row-level triggers, an AFTER trigger that queries the `inserted` special table is used instead, as illustrated in Listing 9.

Listing 9 INSERT trigger on a referencing table realizing an exclusive associations constraint in MSSQL

```
CREATE TRIGGER [EX_PERSON_PHASE_ALIVE_INS]
ON [ALIVE] AFTER INSERT AS BEGIN
IF EXISTS (SELECT 1 FROM inserted i
    JOIN [DECEASED] d ON d.[PERSON_ID] = i.[ID])
BEGIN
  THROW 50000, '...', 2; END; END;
```

Finally, the exclusive association constraint may get violated when a record in the referencing tables is updated or deleted. In PostgreSQL and MSSQL, an AFTER UPDATE/DELETE trigger on all referencing tables is used. The trigger checks that after the operation finishes, all records in the referenced table have exactly one counterpart across all referencing tables; see Listing 10 for an example in MSSQL, implementation in PostgreSQL is similar.

Listing 10 UPDATE/DELETE trigger on a referencing table realizing an exclusive association constraint in MSSQL

```
CREATE TRIGGER [EX_PERSON_PHASE_ALIVE_UPD_DEL]
ON [ALIVE] AFTER UPDATE, DELETE AS BEGIN
IF EXISTS (SELECT 1 FROM [PERSON] p WHERE NOT (
      (EXISTS (SELECT 1 FROM [ALIVE] a
          WHERE a.[PERSON_ID] = p.[ID])
       AND NOT EXISTS (SELECT 1 FROM [DECEASED] d
          WHERE d.[PERSON_ID] = p.[ID])) OR
      (NOT EXISTS (SELECT 1 FROM [ALIVE] a
          WHERE a.[PERSON_ID] = p.[ID])
       AND EXISTS (SELECT 1 FROM [DECEASED] d
          WHERE d.[PERSON_ID] = p.[ID])))) BEGIN
  THROW 50000, '...', 3; END; END;
```

MySQL (i) does not support triggers with multiple triggering events, (ii) does not support statement-level triggers. Therefore, a different implementation must be used. To avoid the mutating table error in a row-level trigger, it must not access the table on which it was fired. Additionally, the trigger must be split up to an UPDATE and a separate DELETE trigger. The UPDATE trigger must check whether the reference has been changed to a different record, if so, it must check whether the sum of records referencing the previous record across all the other referencing tables is one and whether no record that references the currently referenced record exists among all the other referencing tables; see Listing 11. The DELETE trigger works in a similar fashion while omitting the check of the currently referenced record; see Listing 12.

Listing 11 UPDATE trigger on a referencing table realizing an exclusive association constraint in MySQL

```
CREATE TRIGGER `EX_PERSON_PHASE_ALIVE_UPD`
AFTER UPDATE ON `PERSON` FOR EACH ROW BEGIN
IF OLD.`PERSON_ID` <> NEW.`PERSON_ID` THEN
  SET @o_count := SELECT COUNT(*) FROM `DECEASED` d
      WHERE d.`PERSON_ID` = OLD.`PERSON_ID`;
  IF @o_count <> 1 OR EXISTS (
      SELECT 1 FROM `DECEASED` d
        WHERE d.`PERSON_ID` = NEW.`PERSON_ID`) THEN
    SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = "...";
END IF; END IF; END;
```

Listing 12 DELETE trigger on a referencing table realizing an exclusive association constraint in MySQL

```
CREATE TRIGGER `EX_PERSON_PHASE_ALIVE_DEL`
AFTER DELETE ON `ALIVE` FOR EACH ROW BEGIN
SET @o_count := SELECT COUNT(*) FROM `DECEASED` d
    WHERE d.`PERSON_ID` = OLD.`PERSON_ID`;
IF @o_count <> 1 THEN
  SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = "...";
END IF; END;
```

D. Generalization Set

For generalization sets, the constraint must guarantee a valid value in the discriminator column and a correct set of referencing records [14]. We argue there are five kinds of DML operations that may violate the constraint:

1) INSERT and UPDATE on the superclass table — may insert an invalid value in the discriminator column or the correct subclass records do not exist,
2) INSERT to any subclass table — an incorrect subclass record may be added,
3) DELETE from any subclass table — a required subclass record may be removed,
4) UPDATE on any subclass table — the reference to a superclass record may be changed incorrectly.

Thus, insertions and updates to the table representing the superclass must be checked. An example of such trigger for PostgreSQL is provided in Listing 13.

Listing 13 INSERT/UPDATE trigger on the superclass table realizing a generalization set constraint in PostgreSQL

```
CREATE FUNCTION gs_document_type()
RETURNS TRIGGER AS $$ BEGIN
IF NOT ((
      NEW."DISCRIMINATOR" = 'IdCard' AND
      NOT EXISTS (SELECT 1 FROM "PASSPORT" p
          WHERE p."ID" = NEW."ID") AND
      EXISTS (SELECT 1 FROM "ID_CARD" i
          WHERE i."ID" = NEW."ID")) OR
     (NEW."DISCRIMINATOR" = 'Passport' AND
      EXISTS (SELECT 1 FROM "PASSPORT" p
          WHERE p."ID" = NEW."ID") AND
      NOT EXISTS (SELECT 1 FROM "ID_CARD" i
          WHERE i."ID" = NEW."ID"))) THEN
  RAISE EXCEPTION '...';
END IF; RETURN NEW; END; $$ LANGUAGE plpgsql;
CREATE TRIGGER gs_document_type
BEFORE INSERT OR UPDATE ON "DOCUMENT" FOR EACH ROW
EXECUTE FUNCTION gs_document_type();
```

As MSSQL does not support row-level triggers, in this case the trigger must be adjusted to check all inserted or updated records by querying the `inserted` table; see Listing 14.

Listing 14 INSERT/UPDATE trigger on the superclass table realizing a generalization set constraint in MSSQL

```
CREATE TRIGGER [GS_DOCUMENT_TYPE]
ON [DOCUMENT] AFTER INSERT, UPDATE AS BEGIN
IF EXISTS (SELECT 1 FROM inserted i WHERE NOT ((
      i.[DISCRIMINATOR] = 'IdCard' AND
      NOT EXISTS (SELECT 1 FROM [PASSPORT] p
          WHERE p.[ID] = i.[ID]) AND
      EXISTS (SELECT 1 FROM [ID_CARD] i2
          WHERE i2.[ID] = i.[ID])) OR
     (i.[DISCRIMINATOR] = 'Passport' AND
      EXISTS (SELECT 1 FROM [PASSPORT] p
          WHERE p.[ID] = i.[ID]) AND
      NOT EXISTS (SELECT 1 FROM [ID_CARD] i2
          WHERE i2.[ID] = i.[ID])))) BEGIN
  THROW 50000, '...', 1; END; END;
```

As discussed in Subsection IV-A, the prevention of orphan records requires inserting the referencing record first. However, as MySQL does not support deferrable constraint checking, a stored procedure that the user must execute manually has to be used instead. The procedure is executed after the INSERT and UPDATE operations on the superclass table, but only after the referencing record is inserted (if applicable); see Listing 15 (where the `ID` input parameter shall contain the primary key of the inserted or updated record).

Listing 15 Procedure for checking the INSERT/UPDATE operation that realizes a generalization set constraint in MySQL

```
CREATE PROCEDURE `GS_DOCUMENT_TYPE` (IN `ID` INT)
BEGIN
SET @discr = (SELECT `DISCRIMINATOR`
    FROM `DOCUMENT` d WHERE d.`ID` = `ID`);
IF NOT ((
      @discr = 'IdCard' AND
      NOT EXISTS (SELECT 1 FROM `PASSPORT` p
          WHERE p.`ID` = `ID`) AND
      EXISTS (SELECT 1 FROM `ID_CARD` i
          WHERE i.`ID` = `ID`)) OR
     (@discr = 'Passport' AND
      EXISTS (SELECT 1 FROM `PASSPORT` p
          WHERE p.`ID` = `ID`) AND
      NOT EXISTS (SELECT 1 FROM `ID_CARD` i
          WHERE i.`ID` = `ID`))) THEN
  SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = "...";
END IF; END;
```

Next, we argue that for PostgreSQL and MSSQL, the insertion or deletion of a record that references an existing record in the superclass table in any subclass table breaks the generalization set constraint, since the subclass record is inserted before the superclass record and deleted after the superclass record. Therefore, INSERT and DELETE triggers must be employed that prevent the operation if the inserted or deleted record references an existing record in the superclass table. Examples of the INSERT and DELETE triggers for PostgreSQL are provided in Listings 16 and 17, respectively. Again, in MSSQL, due to the lack of support for row-level triggers, the `inserted` and `deleted` special tables must be queried.

Listing 16 INSERT trigger on a subclass table realizing a generalization set constraint in PostgreSQL

```
CREATE FUNCTION gs_document_type_passport_ins()
RETURNS TRIGGER AS $$ BEGIN
IF EXISTS (SELECT 1 FROM "DOCUMENT" d
      WHERE d."ID" = NEW."ID") THEN
  RAISE EXCEPTION '...';
END IF; RETURN NEW; END; $$ LANGUAGE plpgsql;
CREATE TRIGGER gs_document_type_passport_ins
BEFORE INSERT ON "PASSPORT" FOR EACH ROW
EXECUTE FUNCTION gs_document_type_passport_ins();
```

Listing 17 DELETE trigger on a subclass table realizing a generalization set constraint in PostgreSQL

```
CREATE FUNCTION gs_document_type_passport_del()
RETURNS TRIGGER AS $$ BEGIN
IF EXISTS (SELECT 1 FROM "DOCUMENT" d
      WHERE d."ID" = OLD."ID") THEN
  RAISE EXCEPTION '...';
END IF; RETURN OLD; END; $$ LANGUAGE plpgsql;
CREATE TRIGGER gs_document_type_passport_del
BEFORE DELETE ON "PASSPORT" FOR EACH ROW
EXECUTE FUNCTION gs_document_type_passport_del();
```

In MySQL, as discussed in Subsection IV-A, the order of insertion is reversed. Thus, the check for the existence of the correct subclass records is performed in the superclass INSERT trigger, and the subclass INSERT trigger needs to check only if the discriminator value corresponds to the subclass table that is being inserted to. It is not necessary to check that no other record in the same table references the same superclass record, as that is prevented by the PRIMARY KEY constraint on the referencing column.

Concerning the DELETE operation on subclass tables, due to the lack of support for deferrable constraints in MySQL, a subclass record must be deleted before the superclass one. The possibility of leaving a superclass record without a subclass record is impossible to prevent automatically via triggers in MySQL, and can only be checked manually by executing a stored procedure after the subclass and superclass (if applicable) records are deleted; see Listing 18 (where the `ID` column shall contain the primary key of the deleted record).

Listing 18 Procedure after DELETE on a subclass table realizing a generalization set constraint in MySQL

```
CREATE PROCEDURE `GS_DOCUMENT_TYPE_DEL`(IN `ID` INT)
BEGIN
IF EXISTS (SELECT 1 FROM `DOCUMENT` d
      WHERE d.`ID` = `ID`) THEN
  SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = "...";
END IF; END;
```

Finally, the UPDATE operation on the subclass table must not de-associate the record from an existing superclass record, nor associate it with another existing record. In MySQL and PostgreSQL, this can be ensured by an UPDATE trigger on the subclass tables; see Listing 19 for MySQL, implementation in PostgreSQL is similar.

MSSQL does not support row-level triggers, and changes to individual records may only be inferred by comparing

Listing 19 UPDATE trigger on a subclass table realizing a generalization set constraint in MySQL

```
CREATE TRIGGER `GS_DOCUMENT_TYPE_PASSPORT_UPD`
BEFORE UPDATE ON `PASSPORT` FOR EACH ROW BEGIN
IF OLD.`ID` <> NEW.`ID` AND EXISTS (
    SELECT 1 FROM `DOCUMENT` d
        WHERE d.`ID` = OLD.`ID`
        OR d.`ID` = NEW.`ID`) THEN
  SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = "...";
END IF; END;
```

the records in the `inserted` and `deleted` special tables. However, these tables do not contain a column that would make detecting the change of the primary key possible. As a result, in MSSQL, the UPDATE operation on subclass tables can only be checked manually by executing a stored procedure after updating the subclass record; see Listing 20 (where the `@OLD_ID` and `@NEW_ID` input parameters contain the previous and new values of the primary key, respectively).

Listing 20 Procedure after UPDATE on a subclass table realizing a generalization set constraint in MSSQL

```
CREATE PROCEDURE [GS_DOCUMENT_TYPE_PASSPORT_UPD]
       (@OLD_ID INT, @NEW_ID INT) AS BEGIN
IF @OLD_ID <> @NEW_ID AND EXISTS (
    SELECT 1 FROM [DOCUMENT] d
        WHERE d.[ID] = @OLD_ID
        OR d.[ID] = @NEW_ID) BEGIN
  THROW 50000, '...', 3;
END; END;
```

E. Mandatory and Special Multiplicity

The *mandatory multiplicity* and *special multiplicity* constraints impose a restriction on the number of referencing records for a particular referenced record. It follows that five DML operations that may violate the constraint exist:

1) INSERT to the referenced table — an orphaned record may be inserted,
2) UPDATE on the referenced table — an incorrect number of references may point to a record with changed primary key,
3) INSERT or DELETE on the referencing table — an incorrect number of records may point to the referenced record,
4) UPDATE on the referencing table — an incorrect number of record may reference the previous and current referenced record.

In MySQL, the referenced record must be inserted first. Therefore, there always exists a possibility of creating an orphan record that may violate the constraint. As a consequence, the constraint may only be checked manually by executing a stored procedure. Furthermore, as MySQL does not support statement-level triggers, all triggers on the referencing table would produce a mutating table error. Consequently, a stored procedure must be used in place of triggers as well. Collectively, a single stored procedure needs to be executed (i) after inserting to the referenced table, but only after the referencing record is inserted (if applicable), (ii) after updating or deleting from the referenced table, (iii) after inserting, updating or deleting from the referencing table; see Listing 21.

Listing 21 Procedure realizing a special multiplicity constraint in MySQL

```
CREATE PROCEDURE `MUL_PASSP_PERSON_ID` () BEGIN
IF EXISTS (SELECT 1 FROM `PERSON` p WHERE NOT (
      SELECT COUNT(1) FROM `PASSPORT` p2
          WHERE p2.`PERSON_ID` = p.`ID`
    ) BETWEEN 2 AND 4) THEN
  SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = "...";
END IF; END;
```

In the case of PostgreSQL and MSSQL, automatic checking via triggers can be implemented. A trigger on the INSERT and UPDATE operations on the referenced table prevents the operation if an incorrect number of referencing records is detected for the affected records; see Listings 22 and 23 for PostgreSQL and MSSQL, respectively.

Listing 22 INSERT/UPDATE trigger on the referenced table realizing a special multiplicity constraint in PostgreSQL

```
CREATE FUNCTION mul_passp_person_id()
RETURNS TRIGGER AS $$ BEGIN
IF NOT ((
      SELECT COUNT(1) FROM "PASSPORT" p
          WHERE p."ID" = NEW."ID"
    ) BETWEEN 2 AND 4) THEN
  RAISE EXCEPTION '...';
END IF; RETURN NEW; END; $$ LANGUAGE plpgsql;
CREATE TRIGGER mul_passp_person_id
BEFORE INSERT OR UPDATE ON "PERSON" FOR EACH ROW
EXECUTE FUNCTION mul_passp_person_id();
```

Listing 23 INSERT/UPDATE trigger on the referenced table realizing a special multiplicity constraint in MSSQL

```
CREATE TRIGGER [MUL_PASSP_PERSON_ID]
ON [PERSON] AFTER INSERT, UPDATE AS BEGIN
IF EXISTS (SELECT 1 FROM inserted i WHERE NOT ((
      SELECT COUNT(1) FROM [PASSPORT] p
          WHERE p.[ID] = i.[ID]
    ) BETWEEN 2 AND 4)) BEGIN
  THROW 50000, '...', 1; END; END;
```

The INSERT, UPDATE and DELETE operations are checked by a second trigger that prevents the operation if after the operation is executed a record with an incorrect number of referencing records exists. See Listing 24 for an example in MSSQL, implementation in PostgreSQL is similar.

Listing 24 INSERT/UPDATE/DELETE trigger on the referencing table realizing a special multiplicity constraint in MSSQL

```
CREATE TRIGGER [MUL_PASSP_PERSON_ID_REL]
ON [PASSPORT] AFTER INSERT, UPDATE, DELETE AS BEGIN
IF EXISTS (SELECT 1 FROM [PERSON] p WHERE NOT ((
      SELECT COUNT(1) FROM [PASSPORT] p2
          WHERE p2.[ID] = p.[ID]
    ) BETWEEN 2 AND 4)) BEGIN
  THROW 50000, '...', 2; END; END;
```

In Listings 21–24, we presented the realization of a special multiplicity of `[2..4]`. For constraints where the lower or upper bound is unrestricted, a <= or >= operator is used instead of the `BETWEEN` operator.

The mandatory multiplicity constraint is a special case of the special multiplicity constraint, where the lower bound is `1` and the upper bound is unrestricted. Therefore, its realization is the same as of the special multiplicity constraint, possibly with the `EXISTS` predicate instead of >= and the `COUNT` function for improved performance.

V. Discussion

In this paper, we presented our approach to the enforcement of integrity constraints of an OntoUML model in relational databases, in particular PostgreSQL, Microsoft SQL Server and MySQL.

Our research appears to suggest that among the three considered RDBMS, PostgreSQL is the most suitable for the implementation of constraints of an OntoUML model. Due to the lack of support for certain features concerning triggers and the absence of deferrable constraint checking, the realization in MSSQL and MySQL at times falls back to user-executed validation procedures.

Furthermore, due to triggers being processed after each DML operation, the implementation of constraints that validate data across multiple tables is difficult. In our approach, we work around these shortcomings by (i) requiring all FOREIGN KEY constraints to be deferred in PostgreSQL and inserting the referencing record prior to the referenced record, (ii) in MSSQL, due to lack of support for deferrable constraints, temporarily disabling FOREIGN KEY constraints and re-enabling them after both records are inserted, (iii) as MySQL does not re-check referential integrity when constraints are re-enabled, we opted to prioritize referential integrity and implemented many of the constraints as user-executed validation procedures instead of automatically executed triggers. As the implementation in PostgreSQL and MSSQL requires a referencing record to be inserted before the referenced record, automatically generated primary keys cannot be used in the referenced table. On the other hand, as the implementation in MySQL relies heavily on manually-executed procedures, the enforcement of constraints is not guaranteed.

VI. Conclusions

In this paper, we introduced our approach to the transformation of an OntoUML model to a relational database, with focus on the preservation of integrity constraints defined by the OntoUML model. We discussed the realization in the following RDBMS:

- MySQL
- Microsoft SQL Server
- PostgreSQL

We drew on existing research that divides the transformation into three separate steps, as listed below. The original contribution of our paper lies in the third step, which elaborates the implementation in the aforementioned RDBMS.

1) First, the source OntoUML model is transformed to an UML model.
2) Then, the intermediate UML model is transformed to a platform-independent model of a relational database.
3) Finally, the relational model is transformed to a vendor-specific realization in SQL.

We showed the transformation of all possible constraints defined by the constructs used in OntoUML models. We demonstrated the differences between the three considered RDBMS and discussed the limitations of each one w.r.t. the implementation of constraints. The results indicate that PostgreSQL allows implementing complete and automatic constraint enforcement, while MySQL lacks certain needed features and many constraints need to be checked manually.

References

[1] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3rd ed. Boston: Addison-Wesley, Sep. 2003. ISBN 978-0-321-19368-1

[2] G. Guizzardi, A. Botti Benevides, C. Morais Fonseca, D. Porello, J. P. A. Almeida, and T. Prince Sales, "UFO: Unified Foundational Ontology," *Applied Ontology*, vol. 17, no. 1, pp. 167–210, Mar. 2022. doi: 10.3233/AO-210256

[3] DB-Engines. Ranking per database model category. Accessed Apr. 2025. [Online]. Available: https://db-engines.com/en/ranking_categories

[4] Z. Rybola, "Towards OntoUML for Software Engineering: Transformation of OntoUML into Relational Databases," Ph.D. dissertation, Czech Technical University in Prague, Prague, Aug. 2017.

[5] G. Guizzardi, G. Wagner, J. P. A. Almeida, and R. S. Guizzardi, "Towards Ontological Foundations for Conceptual Modeling: The Unified Foundational Ontology (UFO) Story," *Applied Ontology*, vol. 10, no. 3-4, pp. 259–271, Dec. 2015. doi: 10.3233/AO-150157

[6] G. Guizzardi, "Ontological Foundations for Structural Conceptual Models," Ph.D. dissertation, University of Twente, Enschede, 2005.

[7] G. Guizzardi, C. Morais Fonseca, A. B. Benevides, J. P. A. Almeida, D. Porello, and T. P. Sales, "Endurant Types in Ontology-Driven Conceptual Modeling: Towards OntoUML 2.0," in *Conceptual Modeling*. Cham: Springer, 2018, vol. 11157, pp. 136–150.

[8] G. Guizzardi, C. Morais Fonseca, J. P. A. Almeida, T. P. Sales, A. B. Benevides, and D. Porello, "Types and Taxonomic Structures in Conceptual Modeling: A Novel Ontological Theory and Engineering Support," *Data & Knowledge Engineering*, vol. 134, p. 101891, Jul. 2021. doi: 10.1016/j.datak.2021.101891

[9] G. Guizzardi and G. Wagner, "What's in a Relationship: An Ontological Analysis," in *Conceptual Modeling - ER 2008*. Berlin: Springer, 2008, vol. 5231, pp. 83–97. ISBN 978-3-540-87877-3

[10] G. Guizzardi, T. P. Sales, J. P. A. Almeida, and G. Poels, "Automated Conceptual Model Clustering: A Relator-Centric Approach," *Software and Systems Modeling*, vol. 21, no. 4, pp. 1363–1387, Aug. 2022. doi: 10.1007/s10270-021-00919-5

[11] L. Liu and M. T. Özsu, *Encyclopedia of Database Systems*. New York: Springer, 2018. ISBN 978-1-4614-8265-9

[12] I. O. for Standardization, "Database Language SQL — Part 1: Framework," Geneva, Jun. 2023.

[13] J. Jabůrek, "Implementation of the Transformation of an OntoUML Model in OpenPonk into Its Realization in a Relational Database," Master's thesis, Czech Technical University in Prague, Prague, May 2024.

[14] Z. Rybola and R. Pergl, "Towards OntoUML for Software Engineering: Transformation of Kinds and Subkinds into Relational Databases," *Computer Science and Information Systems*, vol. 14, no. 3, pp. 913–937, 2017. doi: 10.2298/CSIS170109035R

[15] DB-Engines. Ranking of Relational DBMS. Accessed Apr. 2025. [Online]. Available: https://db-engines.com/en/ranking/relational+dbms

[16] Z. Rybola and R. Pergl, "Towards OntoUML for Software Engineering: Transformation of Anti-Rigid Sortal Types into Relational Databases," in *Model and Data Engineering*. Cham: Springer, 2016, vol. 9893, pp. 1–15. ISBN 978-3-319-45547-1