Noname manuscript No.
(will be inserted by the editor)

Implementing a Graph Transformation Engine in Relational Databases[*]

Gergely Varró[1], Katalin Friedl[1], Dániel Varró[2]

[1] Department of Computer Science and Information Theory,
Budapest University of Technology and Economics
e-mail: {gervarro|friedl}@cs.bme.hu

[2] Department of Measurement and Information Systems,
Budapest University of Technology and Economics
e-mail: varro@mit.bme.hu

The date of receipt and acceptance will be inserted by the editor

Abstract We present a novel approach to implement a graph transformation engine based on standard relational database management systems (RDBMSs). The essence of the approach is to create database views for each rule and to handle pattern matching by inner join operations while handling negative application conditions by left outer join operations. Furthermore, the model manipulation prescribed by the application of a graph transformation rule is also implemented using elementary data manipulation statements (such as insert, delete). As a result, we obtain a robust and fast transformation engine especially suitable for (i) extending modeling tools with an underlying RDBMS repository and (ii) embedding model transformations into large distributed applications where models are frequently persisted in a relational database and transaction handling is required to handle large models consistently.

Key words Tool support – Graph transformation – Pattern matching – Relational databases

1 Introduction

While nowadays model-driven systems development is more and more being supported by a wide range of conceptually different *model transformation tools*, nearly all of these tools have to solve a common problem: the efficient query and manipulation of complex graph-based model structures. The importance of these issues from an MDA perspective has been identified by issuing the QVT RFP [21] to establish an OMG standard for capturing Queries, Views and Transformations within and between different domains.

While the QVT is a relatively new initiative for specifying model transformations, *graph transformation (GT)* [8] already integrates valuable research results of several decades both from conceptual and implementation side.

From a conceptual point of view, graph transformation provides a visual, rule and pattern-based formal paradigm. Informally, a graph transformation rule performs local manipulation on graph models by finding a matching of the pattern prescribed by its left-hand side (LHS) graph in the model, and changing it according to the right-hand side (RHS) graph.

Graph transformation has proved its maturity for precisely defining (i) the operational semantics of various visual modeling languages (ii) as well as model transformations within and between such languages on a very high level of abstraction. Furthermore, there is already a wide range of available tools for simulation or verification purposes like AGG [9], ATOM3 [31], Diagen [17], Fujaba [10], GReAT [1], Groove [23], Progres [27], Viatra [32] and many more.

Surprisingly, all these tools are identical from a specific aspect, namely, their underlying implementation technology is related to a *programming language* (e.g. Java in case of AGG, Fujaba, Groove, and Viatra2).

Relational database management systems (RDBMSs) that serve as the storage medium for business critical data for large companies are probably the most successful products of software engineering. A crucial factor in this success is the close synergy between theory and practice: SQL, the standard data definition, manipulation and query language is built upon precise mathematical foundations.

In the current paper, we investigate how to exploit powerful RDBMSs to serve as an underlying implementation technology for model transformations. More precisely, we provide a mapping from graph transformation systems into relational databases. The essence of the approach is to create database views for each rule and to handle graph pattern matching by inner join operations while negative application conditions by left outer join operations. Furthermore,

[*] The first and the third author were partially supported by the SEGRAVIS Research Training Network. The first and the second author were also partially supported by the grant OTKA T42559. The János Bolyai Scholarship provided additional support for the third author.
Correspondence to: gervarro@cs.bme.hu

the model manipulation prescribed by the application of a graph transformation rule is also implemented using elementary data manipulation statements (such as INSERT, DELETE). We extend previous results in [33] by the full formalization of this mapping with proofs of correctness.

Furthermore, we implemented a prototype graph transformation engine, which uses open, off-the-shelf relational databases (namely, PostgreSQL [18] or MySQL [28]) as a backend to demonstrate the practical feasibility of our approach. For a detailed experimental evaluation, we assess how the performance of a graph transformation engine based upon a relational database is influenced by (i) parallel rule applications (ii) RDBMS-specific query optimization techniques and (iii) the choice of the underlying RDBMS.

Structure of the paper. Our main intention in Sec. 2 is to briefly and informally summarize the essence of our approach on an example prior to going into deep mathematical details. For that purpose, we assume the reader's familiarity with the basics of relational databases.[1]

Readers also interested in the precise mathematical treatment of our approach should continue with Sec. 3, which provides formal definitions for modeling languages and graph transformation to capture model transformations between these languages. In Sec. 4, an overview is provided to the main concepts of relational databases together with formal definitions. Sec. 5 presents the formalization of our approach to encode graph transformation rules into relational databases (with formal proofs of correctness listed in Appendix A).

Then Sec. 6 discusses implementation issues of our experimental graph transformation engine built upon an off-the-shelf RDBMS. We also investigate how the performance of graph transformation over a RDBMS is dependent on different design decisions, and tool or graph transformation-specific heuristics. Finally, an overview of related work is presented in Sec. 7, while Sec. 8 concludes our paper.

2 Overview of the approach

We first demonstrate how model transformations between modeling languages (metamodels) can be specified by graph transformation. An informal overview is provided on how graph transformation rules can be implemented by using traditional relational database techniques. Our concepts are presented on a widely used benchmark model transformation problem: the object-relational mapping [30], which serves as a running example for the paper.

2.1 Metamodels, models and graph transformation: An informal overview

Metamodels and models. The *metamodel* describes the abstract syntax of a modeling language (or domain). The metamodels of UML class diagrams and relational database schemas (following the CWM standard [20]) are depicted in Fig. 1. In order to avoid complex figures, only the relevant parts of the metamodel is presented.

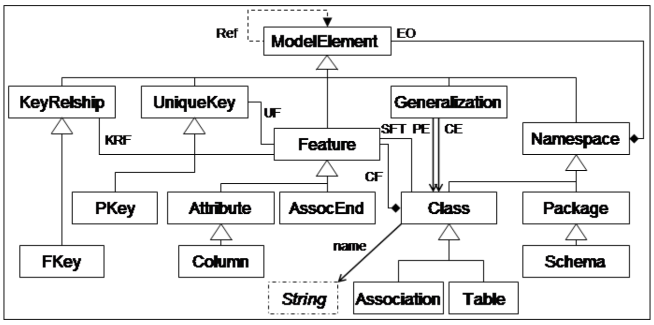


Fig. 1 Metamodel of the problem domain

Nodes (e.g. Schema, Table) of the metamodel are called *classes*. A class may have *attributes* (e.g. the edge labelled by name) that define some kind of properties of the specific class. *Inheritance* may be defined between classes, which means that the inherited class has all the properties its parent has, but it may further contain some extra attributes. Note that the CWM standard derives database notions like tables, columns, etc. from UML notions by inheritance (see Fig. 1).

Associations like EO, CF, SFT, CE, PE, KRF and UF define connections between classes. Both ends of an association may have a *multiplicity* constraint attached to them, which declares the number of objects that, at run-time, may participate in an association. We consider the most typical multiplicity constraints, which are (i) the at most one (denoted by arrows or diamonds), and (ii) the arbitrary (denoted by line ends without arrows and diamonds). Furthermore, we use reference edges (denoted by dashed lines in instance models) connecting source and target model nodes.

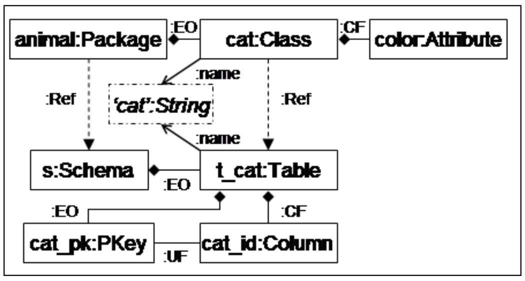
The *instance model* describes concrete systems defined in a modeling language and it is a well-formed instance of the metamodel. Nodes and edges are called *objects* and *links*, respectively. Objects and links are the instances of metamodel level classes and associations, respectively. Attributes in the metamodel appear as *slots* in the instance model. Inheritance in the instance model imposes that instances of the subclass can be used in every situation where instances of the superclass are required.

Example 1 A well-formed instance model of this domain (shown in Fig. 2(a)) has a UML package animal, [2] which contains a UML class cat named as 'cat'. A UML class cat has a UML attribute color. We assume for the paper that UML package animal and UML class cat have already been transformed by the object-relational mapping algorithm, which means that schema s and table t_cat with a name 'cat' are attached to UML package animal and UML class cat, respec-

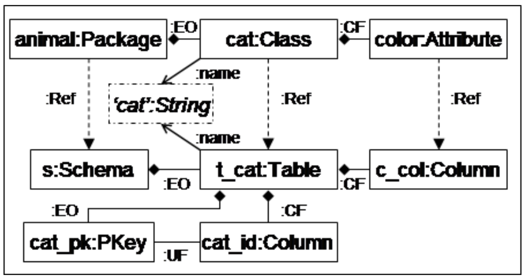
[1] If this is not the case, we recommend to read Sec. 4 for an overview on relational database concepts

[2] To prevent confusion between metamodeling terms and class diagram notions we use the UML prefix for the latter.

tively, via edges of type Ref. Table t_cat has a single column cat_id and a primary key constraint cat_pk referring to the column cat_id.



(a) Model1



(b) Model2

Fig. 2 Sample instance models

Since our technique is applicable for models appearing on any MOF metamodel layer, we use terms *model* and *metamodel* in a generalized sense by not restricting them only to M1 and M2 layer, respectively. According to the terminology of this paper, the *metamodel* is always one MOF level above the *model* independently of the position of *model* in the MOF metamodel hierarchy.

Graph transformation. Graph transformation [8, 24] provides a pattern and rule based manipulation of graph models. Each rule application transforms a graph by replacing a part of it by another graph.

A *graph transformation rule* contains a left–hand side graph LHS, a right–hand side graph RHS, and negative application condition graphs NAC (depicted by crosses). The LHS and the NAC graphs are together called the precondition of the rule.

Rule application. The application of a rule to an instance model replaces a matching of the LHS in the model by an image of the RHS. Informally, this is performed by (i) finding a matching of LHS in the model, (ii) checking the negative application conditions (which prohibit the presence of certain objects and links) (iii) removing a part of the model that can be mapped to LHS but not to RHS yielding the context model, and (iv) gluing the context model with an image of the RHS by adding new objects and links (that can be mapped to the RHS but not to the LHS) obtaining the derived model.

Example 2 The object relational mapping can be described by 6 graph transformation rules as presented in Fig. 3.

(a) SchemaRule (Fig. 3(a)) simply generates a database schema for a UML package.
(b) ClassRule (Fig. 3(b)) searches for a UML class in the UML package, for which there does not exist a corresponding table in the database schema, and creates the corresponding table that has a single column tid, for which a primary key tp is defined.
(c) The inheritance relation in the UML model is handled by appropriate foreign key constraints in the database schema. This is expressed by the GeneralizationRule (Fig. 3(c)), which creates a foreign key constraint on the identifier column cb of the subclass table tb for any unhandled generalization node. The constraint will refer to the column cp of the superclass table tp that has a primary key pp.
(d) A new column is created in the table assigned to the UML class that includes the unhandled UML attribute. This is performed by the AttributeRule (Fig. 3(d)). The application of this rule is restricted to tables not having a name table. A further negative application condition prohibits the attribute to have a UML class as its type.
(e) AssociationRule (Fig. 3(e)) creates a new table in the database, if there has not been any table assigned yet. This new table has again a single column crel with a primary key prel.
(f) The AssocEndRule (Fig. 3(f)) selects an unhandled UML association end, and generates an additional column crel and a corresponding primary key prel in the table trel that has been created for the UML association itself. Moreover, a foreign key constraint is added to the trel table, which refers to the column cc of the table tc that is associated with the UML class c.

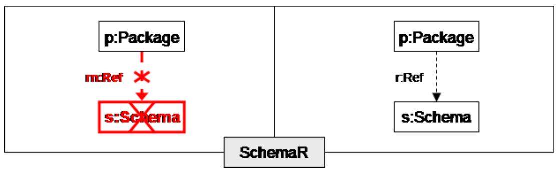
To continue our previous example, one can notice that the attributeRule (Fig. 3(d)) is applicable if there is a UML class in the model that has been transformed to a table with a name not equal to 'table' and this UML class has an untransformed UML attribute of a type not being a UML class. Model 1 of Fig. 2(a) presents a situation where this rule is applicable, since UML class cat, table t_cat and UML attribute color fulfil all the necessary criteria.

In this specific case, rule application means that a new column is added to the selected table and a reference edge is set from the UML attribute to the new column. The derived instance model Model 2 is presented in Fig. 2(b).

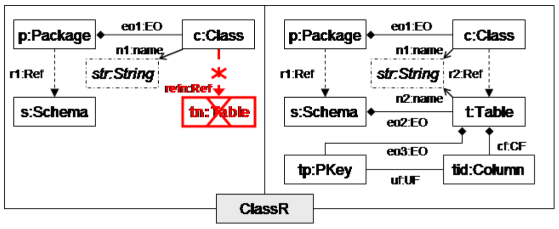
2.2 Graph transformation in relational databases: An informal overview

Mapping metamodels to database tables. We use a standard mapping (for more details see [22, 30]) to generate the schema of the database from the metamodel.

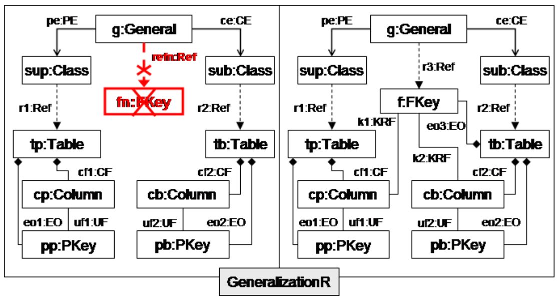
– Each class with k outgoing many-to-one associations and attributes is mapped to a table with $k + 1$ columns. Col-



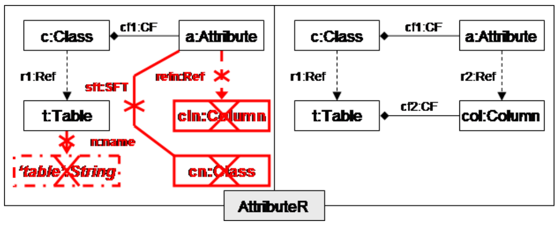
(a) SchemaRule



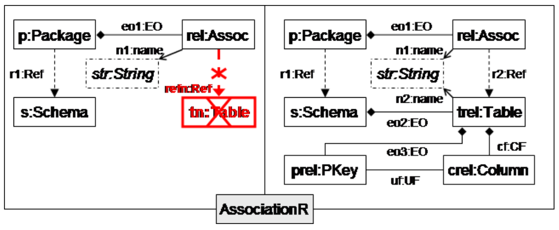
(b) ClassRule



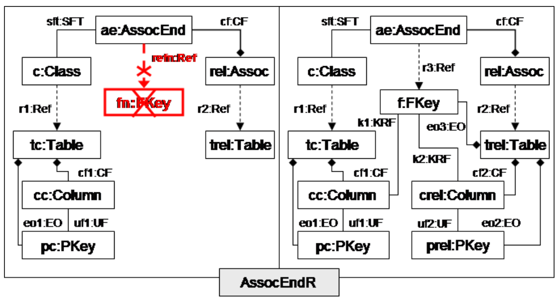
(c) GeneralizationRule



(d) AttributeRule



(e) AssociationRule



(f) AssocEndRule

Fig. 3 Rules describing the object relational mapping

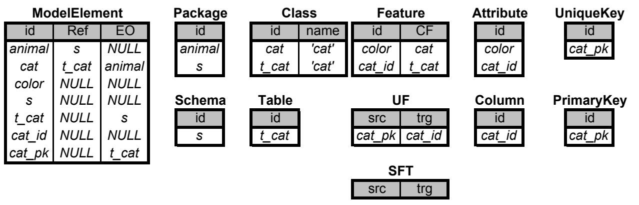
umn *id* will store the identifiers of objects of the specific class. All other columns will contain the identifiers of target objects of such outgoing many-to-one links that have the corresponding association as their direct type. If no such outgoing link exists in the model, the undefined (NULL) value is used in the corresponding column. Additional foreign key constraints, whose role is to guarantee the consistency of the database have to be defined for columns representing many-to-many associations referring to the table assigned to the corresponding target class.

– A table with 2 columns storing the identifiers of source and target objects is assigned to each many-to-many association.
– Inheritance is handled by a foreign key constraint defined for the identifier column *id* of the table assigned to the subclass. This foreign key constraint maintains reference to the identifier column *id* of the superclass table.

Database representation of instance models. Instance models representing the system under design are stored in these database tables.

– A unique identifier is assigned to each object of the instance model.
– The identifier of each object has to appear in the column *id* of all tables that correspond to ancestors of the object's direct type.
– The database representation of a many-to-one link is a row in the table that corresponds to the source class of the link's type. This row should contain the identifiers of source and target objects in the identifier column *id* and the column representing the many-to-one association, respectively.
– Each many-to-many link is represented in the database by a pair of source and target object identifiers appearing in the table that corresponds to the direct type of the link.

Example 3 The database representation of the instance model Model 1 is depicted in Fig. 4.



ModelElement

id	Ref	EO
animal	*s*	*NULL*
cat	*t_cat*	*animal*
color	*NULL*	*NULL*
s	*NULL*	*NULL*
t_cat	*NULL*	*s*
cat_id	*NULL*	*NULL*
cat_pk	*NULL*	*t_cat*

Package

id
animal
s

Class

id	name
cat	*'cat'*
t_cat	*'cat'*

Feature

id	CF
color	*cat*
cat_id	*t_cat*

Attribute

id
color
cat_id

UniqueKey

id
cat_pk

Schema

id
s

Table

id
t_cat

UF

src	trg
cat_pk	*cat_id*

Column

id
cat_id

PrimaryKey

id
cat_pk

SFT

src	trg

Fig. 4 Database representation of the instance model

(i) Model 1 contains a UML class cat, which is identified by the key cat in the database. As ModelElement, Namespace and Class are ancestors of Class, all their corresponding tables should have the key cat in their identifier column *id*. (ii) UML class cat is contained by UML package animal. This containment is a many-to-one link of type EO going from

UML class cat to UML package animal. The database representation of this link is a row in the ModelElement table, which has values cat and animal in columns id and EO, respectively. (iii) Model 1 has a single many-to-many link of type UF connecting objects cat_pk and cat_id, which is represented by a corresponding row in table UF.

Views for LHS and NAC. The matching patterns of a graph transformation rule are calculated by using views, which contain all matchings of the rule. More specifically, we introduce a separate view for each LHS and NAC graph.

1. The view generated for rule graphs (LHS and NAC) executes an *inner join operation* on tables that represent either a node or an edge of the rule graph.
2. The joined table is *filtered by injectivity and edge constraints*. Injectivity constraints express the injective mapping of rule graph nodes and edges on the database level. Edge constraints define restrictions imposed by the graph structure, which means that the source (target) node identifier of the given edge should be found in tables representing the type of the edge and the type of the source (target) node.
3. Finally, a *projection* selects only those columns of the filtered joined table that represent node identifiers. Information about the source and target nodes of edges is discarded during projection. This information is unnecessary in the sequel, since requirements imposed by the graph structure have already been checked and fulfilled.

Example 4 We introduce the essence of this approach by an example listing the view generated for the LHS and NAC graph of the rule AttributeR (see Fig. 3(d)).

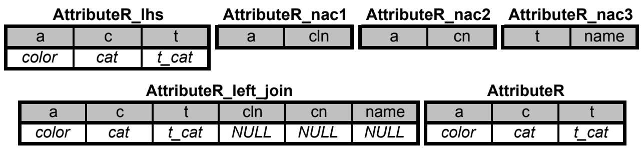
```
CREATE VIEW AttributeR_lhs AS          -- an LHS view
SELECT a.id AS a, t.id AS t, c.id AS c -- with 3 columns
FROM Attribute AS a, Feature AS a_anc, Class AS c,
     ModelElement AS c_anc, Table AS t
WHERE a.id = a_anc.id AND a_anc.CF = c.id  -- CF edge cf1
  AND c.id = c_anc.id AND c_anc.Ref = t.id -- Ref edge r1
  AND c.id <> t.id  -- injectivity constraint
                    -- for nodes c and t

CREATE VIEW AttributeR_nac1 AS
SELECT a.id AS a, cln.id AS cln
FROM Attribute AS a, ModelElement AS a_anc2, Column AS cln
WHERE a.id = a_anc2.id AND a_anc2.Ref = cln.id
                      -- Ref edge refn
  AND a.id <> cln.id  -- injectivity constraint
                      -- for nodes a and cln

CREATE VIEW AttributeR_nac2 AS
SELECT a.id AS a, cn.id AS cn
FROM Attribute AS a, SFT as sft, Class AS cn
WHERE a.id = sft.src AND sft.trg = cn.id   -- SFT edge sft

CREATE VIEW AttributeR_nac3 AS
SELECT t.id AS t
FROM Table AS t, Class AS t_anc
WHERE t.id = t_anc.id AND t_anc.name = 'table'
                      -- for name edge n
```

The LHS of rule AttributeR requires the presence of a CF edge that connects a UML attribute to a UML class. Since CF edges are stored in the Feature table, it must also be included in the inner join operation in addition to tables Attribute and Class. Since the source node of cf1 has to be a UML attribute, only such source object identifiers of the column id of table Feature can participate in a matching that can also be found in table Attribute as expressed by the edge constraint `a.id=a_anc.id`. A similar edge constraint `a_anc.CF=c.id` requires possible target object identifiers of column CF in table Feature to be equal to a value from the identifier column of table Class. A similar pair of equalities express the edge constraints for the reference edge r1. Due to inheritance relations defined in the metamodel, every table is a UML class at the same time. Thus, mappings of t and c to the same object has to be avoided. On the database level, this (injectivity) constraint is expressed by the inequality `c.id <> t.id`.



AttributeR_lhs

a	c	t
color	*cat*	*t_cat*

AttributeR_nac1

a	cln

AttributeR_nac2

a	cn

AttributeR_nac3

t	name

AttributeR_left_join

a	c	t	cln	cn	name
color	*cat*	*t_cat*	*NULL*	*NULL*	*NULL*

AttributeR

a	c	t
color	*cat*	*t_cat*

Fig. 5 Database representation of matchings

The upper part of Fig. 5 shows the contents of views that have been defined for the LHS and the NAC parts of rule AttributeR.

As color is a UML attribute of the UML class cat and this UML class is connected to table t_cat by a reference edge in Model 1, a matching for the LHS of rule AttributeR is found, which is represented by a row in the corresponding (i.e., the leftmost) view of Fig. 5. The view generated for the first NAC is empty, since there is no matching for this NAC as no reference edges leave any UML attributes of Model 1. Since table SFT is empty, the view representing the second NAC has no rows. The last view is again empty, since there are no UML classes with name 'table'.

Left joins for preconditions of rules. When the view for the precondition graph is calculated, views of all its positive and negative application conditions are available. If the precondition has no negative application conditions then the view defined for the LHS contains the database representation of all matchings of the precondition graph.

1. Each NAC view is *left outer joined* to the LHS view one by one. The *join condition* of this operation expresses that columns representing the same shared node in the LHS and the NAC graphs should be equal.
2. For a matching of the precondition graph, we require (in the *null condition*) that columns of NAC(s), which are shared with the LHS part, are filled with undefined values. This means that there are no possible extensions of a matching of the LHS that is also a matching of (any) NAC graph.
3. Then a *projection* is performed, which displays only those columns that originate from LHS.

Example 5 To continue our running example, we present the view definition for the precondition graph of rule AttributeR.

```
CREATE VIEW AttributeR AS
SELECT lhs.*
FROM AttributeR_lhs AS lhs
   LEFT JOIN AttributeR_nac1 AS nac1 ON lhs.a = nac1.a
   LEFT JOIN AttributeR_nac2 AS nac2 ON lhs.a = nac2.a
   LEFT JOIN AttributeR_nac3 AS nac3 ON lhs.t = nac3.t
WHERE nac1.a IS NULL AND nac2.a IS NULL AND nac3.t IS NULL
```

The upper part of Fig. 5 shows the contents of views that have been defined for the LHS, the NACs of rule AttributeR, respectively. The first table in the bottom of Fig. 5 presents the result of the left outer join operation, while the last table corresponds to the precondition of rule AttributeR. Note that columns representing UML attribute a are shared between LHS, NAC_1 and NAC_2 graphs, and columns showing table t are shared between LHS and NAC_3, so these columns appear both in the join and in the filtering condition.

Since views generated for NAC graphs are empty, such column sets of the left outer joined table that originate from the NAC views are filled with NULL values (meaning unsuccessful matchings for the NAC graphs), while column sets from the LHS view contain the database representation of the single matching of the LHS. As this row is not filtered out by the null conditions, it can also be found in the view generated for the whole precondition graph, which means that a matching has been found for the rule AttributeR, and as a consequence the rule is applicable on that matching.

Model manipulation in relational databases. Operations in the graph manipulation phase can be implemented by issuing several data manipulation commands (INSERT, DELETE, and UPDATE) in a single transaction block. The transaction block is needed to ensure that a graph transformation step is atomic, i.e., either all commands or none of them are executed to result in a consistent model after rule application.

In the graph manipulation phase, deletions are followed by insertions.

– We further restrict the order of delete operations in such a way that edge deletions precede node deletions.
  – If a many-to-one link has to be deleted from the model, then the table that represents the source class of the direct type association of the given link has to be updated. Specifically, the value of the column corresponding to the many-to-one association has to be set to NULL in the row that contains the source node identifier of the link in its column id.
  – In case of a deletion of a many-to-many link, the row consisting of the source and the target node identifiers of the link has to be removed from the table that corresponds to the direct type of the given link.
– As the node identifier to be deleted can be found in tables representing the ancestors of the object's direct type, the deletion should proceed in a bottom-up order (to respect foreign key constraints) by starting at the class, which is the direct type of the object.
  During this iteration, additional attention is needed to consistently handle the removal of dangling edges from the database. As a first step, all associations have to be determined, whose source or target is the class, which is just being traversed by the iteration. Then we should perform the above mentioned edge deletion procedure on all links that (i) have the object to be deleted as their source or target node and that (ii) are instances of associations collected in the previous step. The final step of the iteration is the deletion of the object itself from the table that corresponds to the class being traversed. This is performed by deleting the row of this table, which contains the identifier of the given object in its column id.

For handling node and edge insertions on the database level in the graph manipulation phase, we can use exactly the same procedures as for the initial table filling phase.

We state that the new content of database tables always corresponds to the derived model, thus it can be proven that our approach performs graph transformation over an underlying relational database.

Example 6 We continue our sample graph transformation rule AttributeR with the model manipulation parts. This rule prescribes the insertion of a new column, which is contained by the table being selected in the pattern matching phase. Moreover, the origin of this column has to be marked by inserting a new reference link.

On the database level, the same effect can be achieved by generating a new identifier `c_col` for this new column and by inserting this identifier into all tables that represent the ancestors of Column. In order to respect foreign key constraints, insertions are executed in a top-down order starting at the table corresponding to the most general ancestor. Insertion of the 2 new many-to-one links appears as the 2 update operations presented in the listing below.

```
INSERT INTO ModelElement (id) VALUES (c_col);
INSERT INTO Feature (id) VALUES (c_col);
INSERT INTO Attribute (id) VALUES (c_col);
INSERT INTO Column (id) VALUES (c_col);
UPDATE ModelElement SET Ref = c_col WHERE id = color;
UPDATE Feature SET CF = t_cat WHERE id = c_col;
```

When the execution of these graph manipulation commands terminates, the new content of database tables corresponds to the derived model Model 2.

3 Metamodels, models and graph transformation

Now the formalization of concepts related to metamodels, models and graph transformation is presented.

3.1 Metamodels and models

The *metamodel* describes the abstract syntax of a modeling language.

Definition 1 *A* ***directed graph*** *(denoted by* $G = (V_G, E_G, src_G, trg_G)$*) is a 4-tuple, where* V_G *and* E_G *denote nodes and edges of the graph, respectively. Functions* $src_G : E_G \to V_G$ *and* $trg_G : E_G \to V_G$ *map edges to their source and target node, respectively.*

Definition 2 *A **metamodel** (denoted by MM) is a directed graph, where*

- *V_{MM} and E_{MM} denote nodes and edges of the metamodel;*
- *classes (Cls) and datatypes ($DTypes$) form a (distinct and complete) partition of nodes, formally, $V_{MM} = Cls \cup DTypes$, $Cls \cap DTypes = \emptyset$;*
  - *a **class** C is a node of the type graph that represents a user-defined and domain-specific type, formally, $C \in Cls$;*
  - *a **datatype** D is a node of the type graph that represents a built-in type of a programming language (e.g. `int`, `String`), formally, $D \in DTypes$;*
- *associations ($Assoc$) and generalization (inheritance) edges ($Inher$) constitute a partition of edges, formally, $E_{MM} = Assoc \cup Inher$, $Assoc \cap Inher = \emptyset$;*
- *associations can be further partitioned into attributes $Attr$, 'many-to-many' ($Assoc_{M2M}$) and 'many-to-one' ($Assoc_{M2O}$) associations, formally, $Assoc = Assoc_{M2M} \cup Assoc_{M2O} \cup Attr$, $Assoc_{M2M} \cap Assoc_{M2O} = \emptyset, Attr \cap Assoc_{M2M} = \emptyset, Attr \cap Assoc_{M2O} = \emptyset$;*
  - *a **many-to-many association** A **from source class** C_s **to target class** C_t (denoted by $C_s \xrightarrow{A} C_t$) is an edge from the set $Assoc_{M2M}$, where $src_{MM}(A) = C_s \in Cls$, $trg_{MM}(A) = C_t \in Cls$;*
  - *a **many-to-one association** A **from source class** C_s **to target class** C_t (denoted by $C_s \overset{A}{\mapsto} C_t$) is an edge from the set $Assoc_{M2O}$, where $src_{MM}(A) = C_s \in Cls$, $trg_{MM}(A) = C_t \in Cls$;*
  - *an **attribute** A **in a class** C **of a datatype** D (denoted by $C \overset{A}{\mapsto} D$) is an edge from the set $Attr$, formally, $src_{MM}(A) = C \in Cls$, $trg_{MM}(A) = D \in DTypes$, and $C \overset{A}{\mapsto} D \in Attr$;*
- *a **generalization (inheritance) edge** I **leading from class** C_t **to class** C_s (denoted as in UML by $C_s \leftarrowtriangle C_t$) is an edge from the set $Inher$, formally, $src_{MM}(I) = C_t \in Cls$, $trg_{MM}(I) = C_s \in Cls$, and $C_s \leftarrowtriangle C_t \in Inher$.*

In the above definition, associations define binary relations between classes. In the current paper, we do not handle association classes. Note that we use the same notation for many-to-one associations and attributes as they differ only in the categorization of their target nodes. In the following, the notation $C_s \xrightarrow{A} C_t$ is used for a general association of any kind that is $A \in (Assoc_{M2M} \cup Assoc_{M2O} \cup Attr)$.

Inheritance graph. The inheritance hierarchy forms a lattice, which implies that the inheritance graph is a directed acyclic graph (DAG), and there is a common root ancestor class for all classes.

Definition 3 *The **inheritance graph** $MM_{Inher} = (Cls, Inher, src_{MM}, trg_{MM})$ is the type graph restricted to generalization (inheritance) edges, which forms a lattice.*

Definition 4 *Given a metamodel MM, class C_1 is a (direct) **superclass** of class C_2 (or, equivalently, class C_2 is a (direct) **subclass** of class C_1) as denoted by $C_1 \leftarrowtriangle C_2$, if and only if*

- *there is a generalization edge $C_1 \leftarrowtriangle C_2 \in Inher$;*
- *there are no other classes in the inheritance hierarchy between C_1 and C_2, formally, $\nexists C \in V_{MM}$ such that $C_1 \leftarrowtriangle C \leftarrowtriangle C_2$.*

Note that this definition does not imply that a class C_2 has a single superclass C_1, as multiple inheritance is allowed in the inheritance graph. Since the superclass of a class may also have its own superclass, it is useful to define the transitive closure of the superclass relation.

Definition 5 *Given a metamodel MM, class C_1 is an **ancestor (class)** of class C_2 (or, equivalently, class C_2 is a **descendant** of class C_1) (denoted by $C_1 \overset{*}{\leftarrowtriangle} C_2$), if either $C_1 = C_2$, or $\exists C \in V_{MM}$ such that $C_1 \leftarrowtriangle C \overset{*}{\leftarrowtriangle} C_2$.*

Capital letters from the beginning of the alphabet (e.g., $C, D_s \xrightarrow{A} D_t$) will be used for meta-level graph elements (classes, associations).

The instance model describes concrete systems defined in a modeling language and it is always a well-formed instance of the metamodel, which means that typing morphisms for nodes and edges of the model can be defined. Type functions map model nodes and edges to metamodel level classes and associations, respectively.

Definition 6 *Given a metamodel MM, a **well-formed instance model (graph)** M **of the metamodel** MM is a directed graph together with a **direct type function (graph morphism)** $t : M \to MM$, which maps model M to metamodel MM according to the following rules*

- ***Unambiguous mapping of objects and values:*** *model nodes are mapped to metamodel nodes, formally, $\forall c \in V_M : t(c) \in V_{MM}$;*
  - *a model node is called as an **object**, if its direct type is a class;*
  - *a model node is called as a **value**, if its direct type is a datatype;*
- ***Unambiguous mapping of links:*** *model edges (called as **links**) are mapped to associations, formally, $\forall e \in E_M : t(e) \in E_{MM}$;*
- ***Type conformance of source objects:*** *the direct type of the source object of a link is a descendant of the source of the direct type of the same link, formally, $\forall e \in E_M : src_{MM}(t(e)) \overset{*}{\leftarrowtriangle} t(src_M(e))$;*
- ***Type conformance of target objects:*** *the direct type of the target object of a link is a descendant of the target of the direct type of the same link, formally, $\forall e \in E_M : trg_{MM}(t(e)) \overset{*}{\leftarrowtriangle} t(trg_M(e))$;*
- ***Multiplicity criterion for many-to-one associations and attributes:*** *each object can have at most one link of a given direct type originating from the same many-to-one association. Formally, $\forall A \in Assoc_{M2O}$, $\forall e_1, e_2 \in$*

$E_M : src_M(e_1) = a \wedge trg_M(e_1) = b \wedge src_M(e_2) = a \wedge trg_M(e_2) = c \wedge A = t(e_1) = t(e_2) \Longrightarrow e_1 = e_2$; and

- ***Non-existence of parallel edges:*** *No parallel edges are allowed, which means that there cannot be any pair of links of the same type leading between the same pair of objects in a given direction. Formally, $\forall e_1, e_2 \in E_M : src_M(e_1) = src_M(e_2) \wedge trg_M(e_1) = src_M(e_2) \wedge t(e_1) = t(e_2) \Longrightarrow e_1 = e_2$.*

Small letters from the beginning of the alphabet (e.g. $c, a \xrightarrow{e} b$) will be used for objects and links of the instance model.

In the following, we use terms **many-to-many link** (denoted by $a \overset{e}{\rightharpoondown} b$), **many-to-one link** (denoted by $a \overset{e}{\mapsto} b$) and **slot** (denoted again by $a \overset{e}{\mapsto} b$), if the direct type of the given link is a many-to-many association, a many-to-one association and an attribute, respectively.

Type definition can be generalized in such way that all ancestors of a direct type are also implied.

Definition 7 *Given a metamodel MM, a well-formed instance model M with a direct type function t, the **type of an object** c (denoted by $t^*(c)$) consists of all ancestors of $t(c)$. Formally, $t^*(c) = \left\{ C \mid C \in V_{MM} \wedge C \overset{*}{\leftarrow} t(c) \right\}$.*

Alternatives for handling inheritance in graph-based models can be found in [15] (graph schema) and in [29] (typed graphs).

3.2 Graph transformation

Graph transformation [8] provides a pattern and rule based manipulation of graph models. Each rule application transforms a graph by replacing a part of it by another graph.

Definition 8 *Given a metamodel MM, a **basic rule** r_b consists of a left–hand side graph LHS and a right–hand side graph RHS and an injective partial morphism $p : LHS \to RHS$ where LHS and RHS are well-formed instances of the metamodel MM. One further criteria has to be fulfilled, namely,*

- ***Preservation of values:*** *If a value appears on one side of a basic rule, then it must also exist on the other side. Formally, $\forall x \in V_{LHS} : t(x) = D \Longrightarrow D \in DTypes \Longrightarrow x \in V_{RHS} \wedge p(x) = x$, and $\forall x \in V_{RHS} : t(x) = D \Longrightarrow D \in DTypes \Longrightarrow x \in V_{LHS} \wedge p(x) = x$.*

Definition 9 *Given a metamodel MM and a basic rule r_b, a **negative application condition** [13] consists of the LHS graph of r_b, a directed graph NAC (depicted by crosses in figures) and an injective partial morphism $p_{NAC} : LHS \to NAC$. The NAC graph also has to be a well-formed instance of the metamodel MM.*

In the following, we use the term **attribute constraint** for a slot appearing in an LHS or a NAC graph, and we restrict ourselves to equalities in case of attribute constraints. A slot in an RHS graph is called an **attribute assignment** in the sequel.

Definition 10 *Given a metamodel MM, a **graph transformation rule** r consists of a basic rule r_b, and a set of negative application conditions $\{NAC_i\}$.*

Definition 11 *The **precondition of rule** r (denoted by r_{PRE}) is the LHS graph together with the set of negative application conditions.*

The LHS graph and the ith negative application condition graph NAC_i of a rule r are denoted by r_{LHS} and r_{NAC_i}, respectively. For the graph objects (nodes and edges) of rules we always use small letters from the end of the alphabet (e.g. $x, u \xrightarrow{z} v$).

In this paper we use notation $V_{LHS} \setminus V_{RHS}$ for the nodes of LHS that do not have images in RHS according to morphism p. The notational shorthand $V_{LHS} \cap V_{RHS}$ will denote those nodes of LHS that are mapped by the morphism p. Finally, $V_{RHS} \setminus V_{LHS}$ marks those nodes of RHS that do not have an origin in LHS. We will use the same notation for negative application conditions (e.g. $V_{LHS} \setminus V_{NAC_i}$) and edges (e.g. $E_{LHS} \setminus E_{RHS}$).

For the application of a rule we follow the single pushout approach [24] with injective morphisms. However, the definitions are slightly adapted to our proof technique.

Definition 12 *A **matching** m **for a graph** G **in a model** M (denoted by m_G) is a type conformant total morphism $m_G : G \to M$, which means that*

- *$\forall x \in V_G, \exists c \in V_M : t(x) \overset{*}{\leftarrow} t(c) \wedge m_G(x) = c$, and*
- *$\forall u \xrightarrow{z} v \in E_G, \exists a \xrightarrow{e} b \in E_M : t(u) \overset{*}{\leftarrow} t(a) \wedge t(v) \overset{*}{\leftarrow} t(b) \wedge t(z) = t(e) \wedge m_G(u \xrightarrow{z} v) = a \xrightarrow{e} b$.*

Definition 13 *A **matching** m **for a rule** r **in a model** M (denoted by m_r) is*

- *a matching m for the LHS in model M, provided that*
- *no matching exists for any NAC graph, formally, $\forall NAC_i \nexists m' : NAC_i \to M$, for which $\forall x \in V_{LHS} \cap V_{NAC_i} : m'(x) = m(x)$ and $\forall u \xrightarrow{z} v \in E_{LHS} \cap E_{NAC_i} : m'(u \xrightarrow{z} v) = m(u \xrightarrow{z} v)$.*

Definition 14 *Given a matching m for a rule r in model M, the **deletion phase** of a rule application of the rule r is executed on a matching m in the model M yielding the context model M_c, when*

- *we delete all objects, to which nodes appearing only in the LHS are mapped by m, formally, $V_{M_c} = V_M \setminus \{ c \mid \exists x \in V_{LHS} \setminus V_{RHS} \wedge m(x) = c \}$; and*
- *we delete all links, to which edges appearing only in the LHS are mapped by m, formally,*

$$E_{M_1} = E_M \setminus \left\{ a \xrightarrow{e} b \mid \exists u \xrightarrow{z} v \in E_{LHS} \setminus E_{RHS} \wedge m(u \xrightarrow{z} v) = a \xrightarrow{e} b \right\};$$

– *all dangling (i.e., incident) edges are deleted as well, formally,*

$$E_{M_c} = E_{M_1} \setminus \left\{ a \xrightarrow{e} b \mid \exists x \in V_{LHS} \setminus V_{RHS} \wedge (m(x) = a \vee m(x) = b) \right\}$$

Definition 15 *Given a matching m for a rule r in model M, the* ***insertion phase*** *of a rule application of the rule r is executed on a matching m in the context model M_c yielding the model M', if*

– *a new object $m(x)$ is added to model M for each node of RHS that is not contained by LHS, formally, $V_{M'} = V_{M_c} \cup \{ m(x) \mid x \in V_{RHS} \setminus V_{LHS} \}$, and*
– *a new link $m(u \xrightarrow{z} v)$ is added to model M for each edge of RHS that cannot be found in the LHS graph. Note that in this case source and target objects $m(u)$ and $m(v)$ already exist in the model M. Formally, $E_{M'} = E_{M_c} \cup \left\{ m(u \xrightarrow{z} v) \mid u \xrightarrow{z} v \in E_{RHS} \setminus E_{LHS} \right\}$*

Definition 16 *Given a matching m for a rule r in model M,* ***rule r is applied to the matching m in the model M yielding the derived model*** *M', (denoted by $M \overset{r,m}{\Longrightarrow} M'$) if deletion and insertion phases are executed in this order.*

When modeling complex systems, naturally, more than a single graph transformation rule is required. A graph transformation system encapsulates a set of rules, which can be applied during the evolution of the system model.

Definition 17 *A* ***graph transformation system*** *$\mathfrak{S}_{GT} = (MM, R)$ is a tuple that consists of the metamodel MM, and the set of graph transformation rules R.*

Definition 18 *Given a graph transformation system $\mathfrak{S}_{GT} = (MM, R)$, a* ***graph transformation run*** *is a sequence of rule applications (denoted as $M_I \overset{*}{\Longrightarrow} M_n$), which starts from an initial model M_I and which applies rules from the set R.*

4 Database operations

In our graph transformation engine a relational DBMS is used to represent metamodels as database schemas, to store instance models and to perform modifications on such models. Now we summarize the database terminology used throughout this paper.

4.1 Tables and views

The most basic entities of a database are tables that may have several columns and their role is to store data in its rows.

Definition 19 *A* ***database table with n columns*** *(denoted by $\mathcal{T}^{(n)}(A_1, \ldots, A_n)$) is an n-ary relation over sets $(C_1 \cup \{ \varepsilon \}), \ldots, (C_n \cup \{ \varepsilon \})$. $\mathcal{T}$ and A_i denote names of the table and of the ith column, respectively. Column names definitely have to be unique in the scope of a single table, thus a table cannot have columns sharing the same name. The ith column of the table may contain values from the set C_i. Undefined (or null) values (denoted by ε) are also allowed in any columns. Formally, $\mathcal{T}(A_1, \ldots, A_n) \subseteq (C_1 \cup \{ \varepsilon \}) \times \ldots \times (C_n \cup \{ \varepsilon \})$.*

Definition 20 *Since database tables are n-ary relations, their elements are n-tuples $\mathbf{x} = (x_1, \ldots, x_n)$, which are called* ***rows*** *in database terminology.*

While the traditional relational DBMSs use multi-set semantics, we can simplify to set semantics in the paper, since uniqueness of rows can be guaranteed by the algorithm that will be presented in Sec. 5.

Definition 21 *A* ***direct column reference for a table*** *$\mathcal{T}$ (denoted by $\mathcal{T}.A_i$ or simply by A_i (if the table to which it refers can unambiguously be determined)) identifies the column of $\mathcal{T}$ that has a name A_i.*

Definition 22 *Given a table $\mathcal{T}$ with a column called A_i, a* ***direct column reference for a row*** *$\mathbf{t} \in \mathcal{T}$ (denoted by $\mathbf{t}[A_i]$) identifies the element of $\mathbf{t}$ that can be found in the column $\mathcal{T}.A_i$.*

Definition 23 *A* ***primary key constraint for columns*** *$A_1, \ldots, A_j$* ***of table*** *$\mathcal{T}(A_1, \ldots, A_n)$ guarantees the uniqueness of values in the selected set of columns. Formally, $\forall \mathbf{r}, \mathbf{s} \in \mathcal{T} : (\mathbf{r} = \mathbf{s} \iff \forall i, 1 \leq i \leq j : \mathbf{r}[A_i] = \mathbf{s}[A_i])$.*

Foreign key constraints are integrity constraints provided by the most RDBMSs. Their role is to ensure that columns in different tables never contain inconsistent data. In our approach, these constraints are (mainly) used to guarantee that the database representation of an edge can never appear in the database without its source and target nodes being already present.

Definition 24 *A* ***foreign key constraint for column*** *$\mathcal{R}.A$* ***referring to column*** *$\mathcal{S}.B$ (denoted by $\mathcal{R}.A \xrightarrow{FK} \mathcal{S}.B$) declares that all values of column $\mathcal{R}.A$ should also be found in column $\mathcal{S}.B$, or formally $\mathcal{R}.A \subseteq \mathcal{S}.B$.*

Definition 25 *A* ***view*** *$\mathcal{V}$ is a derived (calculated) table with a separate name.*

Definition 26 *The* ***database schema*** *(denoted by $\mathfrak{S}_{DB}$) consists of the set of tables and views appearing in the database.*

4.2 Query operations

After introducing the basic entities (i.e., tables), query operations are discussed, which can be used to define derived tables (i.e., views).

Definition 27 *Given an ordered sequence of column references $\mathcal{T}.A_1, \ldots, \mathcal{T}.A_k$ for $\mathcal{T}$, the* ***projection of a table*** *$\mathcal{T}$* ***to columns*** *$A_1, \ldots, A_k$ (denoted by $\pi_{A_1,\ldots,A_k}(\mathcal{T})$) is a k-ary* *relation, which consists of only the enumerated columns of $\mathcal{T}$. Its formal definition is as follows*

$$(x_1, \ldots, x_k) \in \pi_{A_1,\ldots,A_k}(\mathcal{T}) \iff \exists(y_1, \ldots, y_n) \in \mathcal{T} : \bigwedge_{i=1}^{k} x_i = y_{A_i},$$

where $\bigwedge_{i=1}^{k} x_i = y_{A_i}$ denotes the conjunction (logical AND*) of equalities.*

In SQL terms projection is implemented in the select statement as follows:

```
SELECT $A_1,\ldots,A_k$ FROM R;
```

Definition 28 *An* ***atomic expression*** *has a form $\alpha\theta\beta$, where α and β can be either a column of $\mathcal{T}$ or a constant c. θ is a comparison operator, so $\theta \in \{ =, <, >, \leq, \geq, \neq \}$. A* ***formula*** *$F$ is either an atom or it is constructed from atoms using the logical and ($\wedge$), logical or ($\vee$), and negation ($\neg$) operators.*

Definition 29 *Given a formula F,* ***selection*** *(denoted by $\sigma_F(\mathcal{T})$) operates on a single table $\mathcal{T}$ and collects the rows of $\mathcal{T}$ where $F(y_1, \ldots, y_n)$ holds. The formal definition of selection is*

$$\sigma_F(\mathcal{T}) = \{ (y_1, \ldots, y_n) \mid (y_1, \ldots, y_n) \in \mathcal{T} \wedge F(y_1, \ldots, y_n) = \textit{true} \} .$$

An obvious corollary is that $\sigma_F(\mathcal{T}) \subseteq \mathcal{T}$.

Selection operation can also be expressed in SQL, using a WHERE condition with F as its parameter.

Definition 30 *The* ***cross join of tables*** *$\mathcal{R}^{(m)}$* ***and*** *$\mathcal{S}^{(n)}$ (denoted by $\mathcal{R} \times \mathcal{S}$) is a table with $m + n$ columns and it is the Cartesian product of the two tables. A row is in the result table, if its first m values correspond to a row in $\mathcal{R}$ and its last n values corresponds to a row in $\mathcal{S}$. Its formal definition is:*

$$\mathcal{R} \times \mathcal{S} = \{ (x_1, \ldots, x_m, y_1, \ldots, y_n) \mid (x_1, \ldots, x_m) \in \mathcal{R} \wedge (y_1, \ldots, y_n) \in \mathcal{S} \} .$$

Cross join operation also exists in SQL, which can be formulated as:

```
SELECT * FROM R,S;
```

Column name uniqueness has only a table scope, so name clashes may occur in joint tables. In order to avoid this uncomfortable consequence caused by join operations, we should be able to differentiate between columns that originate from different base tables.

In RDBMSs name clashes are resolved by some renaming mechanisms. The SQL notation for renaming depends on the actual RDBMS software that is being used. In this paper, we use the PostgreSQL notation, namely the AS keyword for this purpose in SQL queries (e.g. `T.id AS T`). In our mathematical formalism, column sets implement the table renaming functionality, while column renaming is performed implicitly by defining a new name for a column in the view definition.

Definition 31 *Given two tables $\mathcal{R}^{(m)}$ and $\mathcal{S}^{(n)}$, a* ***column set of a joint table*** *$\mathcal{R} \times \mathcal{S}$* ***referring to the base table*** *$\mathcal{R}$ (denoted by R^{cs}) is the largest possible set of columns that originate from table $\mathcal{R}$, which is the first m columns of $\mathcal{R} \times \mathcal{S}$ in this case.*

Definition 32 *Given two tables $\mathcal{R}$ and $\mathcal{S}$, an* ***indirect column reference for the joint table*** *$\mathcal{T} = \mathcal{R} \times \mathcal{S}$ (denoted by $\mathcal{T}.\mathcal{R}^{cs}.A_i$, or simply by $\mathcal{R}^{cs}.A_i$) identifies a column of $\mathcal{T}$ by selecting a column set first and then by using the direct column reference A_i on the column set.*

An indirect column reference for a row of the joint table can be similarly defined.

Definition 33 *Given a formula F, the* ***inner join of tables*** *$\mathcal{R}$* ***and*** *$\mathcal{S}$ (denoted by $\mathcal{R} \overset{F}{\bowtie} \mathcal{S}$) is a selection from the Cartesian product filtered by formula F. Formally,*

$$\mathcal{R} \overset{F}{\bowtie} \mathcal{S} = \sigma_F(\mathcal{R} \times \mathcal{S}).$$

In this paper, only atoms of type $A = B$ (two column names in equality relation) and the logical and operator will be used for basic atoms and for constructing formulae, respectively. Typically, A and B are taken from different tables. It is useful from a practical point of view, if column names on the different sides of the equality relation are from different tables. However, the general definition does not require any such restrictions. SQL notation of the inner join operation is as follows.

```
SELECT * FROM R INNER JOIN S ON R.A=S.B;
```

Definition 34 *Given a formula F, the* ***left outer join of tables*** *$\mathcal{R}$* ***and*** *$\mathcal{S}$ (denoted by $\mathcal{R} \overset{F}{\ltimes} \mathcal{S}$) (i) contains all the rows of $\mathcal{R} \overset{F}{\bowtie} \mathcal{S}$, (ii) additionally contains all such rows of $\mathcal{R}$, for which there does not exist any row in $\mathcal{S}$, where $F(\mathbf{x}|\mathbf{y})$ holds, and (iii) the latter rows are filled with undefined values in columns originating from $\mathcal{S}$.*

The formal definition of left outer join is

$$\mathcal{R} \overset{F}{\ltimes} \mathcal{S} = (\mathcal{R} \overset{F}{\bowtie} \mathcal{S}) \cup \{ (\mathbf{x}, \varepsilon, \ldots, \varepsilon) \mid \mathbf{x} \in \mathcal{R} \wedge \nexists \mathbf{y} \in \mathcal{S} \text{ for which } F(\mathbf{x}|\mathbf{y}) = \textit{true} \} .$$

where $F(\mathbf{x}|\mathbf{y})$ denotes whether formula F is satisfiable if its unbound variables are replaced by the corresponding values of rows $\mathbf{x}$ and $\mathbf{y}$.

A sample query presenting the left outer join operation is

```
SELECT * FROM R LEFT JOIN S ON R.A=S.B;
```

4.3 Data manipulation operations

Finally, we define three data manipulation operations. $\mathcal{T}'$ will mark the content of table $\mathcal{T}$, after the database operation has completed.

Definition 35 *The **delete operation***

$$\texttt{DELETE FROM } \mathfrak{T} \texttt{ WHERE } A_1 = y_1 \texttt{ AND } \ldots \texttt{ AND } A_k = y_k$$

removes those rows of table $\mathfrak{T}$, which contain values y_i in their column A_i, respectively. Formally, $\mathfrak{T}' = \mathfrak{T} \setminus \left\{ \mathbf{x} \in \mathfrak{T} \mid \bigwedge_{i=1}^{k} \mathbf{x}[A_i] = y_i \right\}$, where $\bigwedge_{i=1}^{k} \mathbf{x}[A_i] = y_i$ denotes the conjunction (logical AND*) of equalities.*

Definition 36 *The **update operation***

$$\texttt{UPDATE } \mathfrak{T} \texttt{ SET } A_j = y \texttt{ WHERE } A_i = x$$

sets the value of column A_j to y in all rows of table $\mathfrak{T}(A_1, \ldots, A_n)$ where column A_i has value x. Formally, $\mathfrak{T}' = (\mathfrak{T} \setminus Minus) \cup Plus$, where

$$Minus = \{ \mathbf{z} \in \mathfrak{T} \mid \mathbf{z}[A_i] = x \}$$

and

$$Plus = \left\{ \mathbf{z}' \mid \exists \mathbf{z} \in Minus, \forall k \in \mathbb{Z}_n^+ : \mathbf{z}'[A_j] = y \wedge \bigwedge_{j \neq k} \mathbf{z}'[A_k] = \mathbf{z}[A_k] \right\},$$

where $\mathbb{Z}_n^+$ denotes the set of positive integers up to n (i.e., $1 \leq k \leq n$).

Definition 37 *The **insert operation***

$$\texttt{INSERT INTO } \mathfrak{T}\ (A_1, \ldots, A_k) \texttt{ VALUES } (y_1, \ldots, y_k)$$

adds an n-tuple $\mathbf{y}$ to table $\mathfrak{T}$, if $\mathbf{y}$ is not yet contained. The tuple $\mathbf{y}$ has value y_i in column A_i, respectively, and it contains undefined values in all other columns. In other words, $\mathfrak{T}' = \mathfrak{T} \cup \{ \mathbf{y} \}$, where $\mathbf{y}[A_i] = y_i$, if $1 \leq i \leq k$, and $\mathbf{y}[C] = \varepsilon$, if $C \notin \{ A_1, \ldots, A_k \}$.

Definition 38 *Given a sequence of database operations TA, a **transaction is executed on a representation $\mathfrak{M}$ resulting in an other representation $\mathfrak{M}'$** (denoted by $\mathfrak{M} \overset{TA}{\Longrightarrow} \mathfrak{M}'$), if either all operations of TA or none of them are executed.*

5 Graph transformation in relational databases

We present how a graph transformation engine (following the single pushout [24] approach with injective matchings) can be implemented using a relational database. First, we present how an appropriate database schema can be created based on the metamodel, and how the database representation of the model can be generated (Sec. 5.1). Afterwards, the pattern matching phase of rule application is implemented using database queries (Sec. 5.2–5.3), finally data manipulation is handled (in Sec. 5.4).

5.1 Mapping metamodels and models to database tables

Mapping of metamodels to database tables. Instance models representing the system under design are stored in database tables. We use the standard bi-directional mapping (for more details see [22, 30]) to generate the schema of the database with BCNF property [6] from the metamodel.

– Let us first introduce a set called *universe* (denoted by $\mathfrak{U}$), which denotes the set of all identifiers that are (or will be) ever stored in the database.
– Each datatype D is mapped to a table with a single column $D^d(id)$. Column id contains all the possible values of the datatype that can be ever used. Note that this table is introduced only for making definitions, notations and proofs simpler and more understandable. As relational databases support some built-in types (e.g. `DECIMAL`, `VARCHAR`), the implementation omits these tables as the same type restrictions can be achieved by defining appropriate built-in types for columns. Formally, $D^d \subseteq \mathfrak{D}$, where $\mathfrak{D}$ denotes the built-in database type that is assigned to datatype D of the metamodel.
– Each class C with k outgoing many-to-one associations (and attributes) ($C \overset{A_1}{\mapsto} C_1, \ldots, C \overset{A_k}{\mapsto} C_k$) is mapped to a table with $k+1$ columns $C^d(id, A_1^d, \ldots, A_k^d)$. Column id will store the identifiers of objects of the specific class. Column A_i^d will contain the identifiers of target objects of such outgoing many-to-one links that have association $C \overset{A_i}{\mapsto} C_i$ as their direct type. If no such outgoing link exists in the model, the undefined value ε is used in column A_i^d. Additionally, we should define foreign keys $\forall i \in [1..k] : C^d.A_i^d \overset{FK}{\to} C_i^d.id$ to respect the graph structure in the database. Formally, $C^d \subseteq \mathfrak{U} \times \left(C_1^d \cup \varepsilon\right) \times \ldots \times \left(C_k^d \cup \varepsilon\right)$.
– We assign a table $A^d(src, trg)$ for each many-to-many association $C_s \overset{A}{\rightharpoonup} C_t$ connecting classes C_s and C_t in the metamodel. Columns src and trg contain identifiers of source and target objects, respectively. Foreign keys $A^d.src \overset{FK}{\to} C_s^d.id$ and $A^d.trg \overset{FK}{\to} C_t^d.id$ should additionally be defined to respect the graph structure (preserve the source and the target of edges) in the database. In a more formal way, $A^d \subseteq C_s^d \times C_t^d$.
– If a class C is inherited from a superclass D, then table C^d should be extended by a foreign key constraint $C^d.id \overset{FK}{\to} D^d.id$.

We introduced the superscript d to uniformly denote database representations of all kinds of graph transformation related entities. For instance, C^d, r_{LHS}^d, and c^d mark the entities that represent a class C, a rule graph r_{LHS}, and an object c in the database, respectively. This notation is always used as a bi-directional mapping meaning that, e.g. C^d unambiguously identifies the database table that was assigned to class C, and vice versa.

Mapping of instance models into rows. Now we define a bijective mapping, which assigns an identifier to each object of the instance model. The image of the mapping c^d will be used as a primary key that identifies an object c in the database.

In order to appropriately represent an object in the database, its key has to be contained by all tables that are assigned to an ancestor of the object's type. Since inheritance relation in the metamodel (i.e., the type hierarchy) poses restriction (in the form of foreign key constraints) on exactly the same set of tables, additional care has to be taken when inserting (or deleting) even a single key (identifier). The order that handles insertion correctly is being defined now.

Definition 39 *Given a metamodel MM with inheritance relations that are acyclic, a* ***topological order of a type*** *t (denoted by $TopologicalOrder(t)$) is such a sequence of the ancestors of t in which a class D cannot appear before an ancestor C in the order, if $C \overset{*}{\leftharpoondown} D$.*

A natural consequence of the definition is that type t is the last element in its topological order.

Definition 40 *Given a metamodel MM with inheritance relations that are acyclic, an* ***inverse topological order of a type*** *t (denoted by $InverseTopologicalOrder(t)$) is a topological order of t traversed in the opposite order.*

A natural consequence of the definition is that type t is the first element in its inverse topological order.

After fixing a certain topological and inverse topological order of a type to be used in the sequel, Algorithm 1 derives the database representation of the initial model as follows.

– We suppose that all the tables are initially empty.
– A new identifier c^d is generated for each object c of the instance model M. Then ancestors of the type $t(c)$ of the object c are determined and furthermore they are ordered topologically according to the inheritance relation. The ordering is done in a top-down manner, meaning that the "most general" class is enumerated first. (The role of topological ordering is to avoid the violation of foreign key constraints that have already been imposed on database tables.) The final step is to insert the new identifier to all the tables that have been assigned to the enumerated ancestor classes. (Note that this algorithm performs exactly the same steps in the cases, when $t(c)$ is a class or a datatype.)
– For each many-to-one link $a \overset{e}{\mapsto} b$ of the instance model, the row in the table $src(t(e))^d$, which represents the source object a, is updated by replacing the value in column $t(e)^d$ by the identifier b^d of the target object b.
– For each many-to-many link $a \overset{e}{\rightharpoonup} b$ of the instance model, the identifiers of the source and target nodes (a^d and b^d) are inserted to the table $t(e)^d$ that has been assigned to the edge type (association) $t(e)$ of the link.

We introduce a new term that formalizes the consistent database representation of an instance model.

Definition 41 *Let a metamodel MM, and a database schema $\mathfrak{S}_{DB}$ be given together with the bidirectional mapping d from MM to the tables of $\mathfrak{S}_{DB}$.*

Algorithm 1 From instance models to its database representation

```
1:  for all $c \in V_M$ {For all objects in model $M$} do
2:      $c^d := GenerateNewIdentifier()$
3:      for all $C \in TopologicalOrder(t(c))$ do
4:          INSERT INTO $C^d$ $(id)$ VALUES $(c^d)$ {Inserts the new identifier to all ancestor tables}
5:      end for
6:  end for
7:  for all $a \overset{e}{\mapsto} b \in E_M$ {For all many-to-one links (and slots) in model $M$} do
8:      UPDATE $src(t(e))^d$ SET $t(e)^d = b^d$ WHERE $id = a^d$ {Updates the value in column $t(e)^d$ to $b^d$ in the row with identifier $a^d$}
9:  end for
10: for all $a \overset{e}{\rightharpoonup} b \in E_M$ {For all many-to-many links in model $M$} do
11:     INSERT INTO $t(e)^d$ $(src, trg)$ VALUES $(a^d, b^d)$ {Inserts identifiers of end points $a$ and $b$ into the table that corresponds to many-to-many association $t(e)$}
12: end for
```

A model M and a database representation $\mathfrak{M}$ are consistent *($M \cong \mathfrak{M}$), if*

– *each object (and value) of the instance model is represented in the database by one row in all the tables that have been assigned to ancestors of the node type. Moreover, these rows must contain the identifier of the object in their identifier column id. Formally, $\forall C \in V_{MM}, \forall c \in V_M : \left(C \overset{*}{\leftharpoondown} t(c) \iff \exists \mathbf{c} \in C^d : \mathbf{c}[id] = c^d\right)$,*
– *each many-to-one link (and slot) of the instance model is represented in the database by exactly one row in the table that corresponds to the source class of the type of the edge. This single row must contain identifiers of source objects in the identifier column id and target objects in the column corresponding to the direct type of the edge. Formally, $a \overset{e}{\mapsto} b \in E_M \iff \left(\exists \mathbf{a} \in src(t(e))^d : \mathbf{a}[id] = a^d \wedge \mathbf{a}[t(e)^d] = b^d\right)$, and*
– *the identifiers of source and target nodes of each many-to-many link (edge) of the instance model can be found exactly in the table that corresponds to the type of the edge. Formally, $a \overset{e}{\rightharpoonup} b \in E_M \iff \left(a^d, b^d\right) \in t(e)^d$.*

Finally, we formulate a theorem, which states that the database representation that has been created by the abovementioned initialization algorithm is consistent with the initial instance model.

Theorem 1 *The initial instance model M and its database representation $\mathfrak{M}$ are consistent. Formally, $M \cong \mathfrak{M}$.*

Proof Proofs of all theorems can be found in Appendix A.

5.2 Views for rule graphs (LHS and NAC).

As it is described in Sec. 2.2, the view generated for rule graphs (LHS and NAC) executes an inner join operation on tables that have been assigned to types of nodes and edges appearing in the rule graph. Then the joined table is filtered by injectivity and edge constraints. Finally, a projection selects only those columns of the filtered joined table that represent node identifiers.

Formalization. In order to define pattern matching calculation for an LHS precisely, let us suppose that $n_V = |V_{LHS}|$ and $n_E = |E_{LHS}|$. Let us define a total order on the node and edge sets in which nodes precede edges, and let x_i and z_{n_V+j} be the ith node and the jth edge according to this order, respectively.

Now the view ${r^d_{LHS}}^{(n_V)}$ for the LHS can be calculated as follows:

$$r^d_{LHS}(ResCols) = \pi_{ProjColRefs}\left(\sigma_{Inj \wedge Edge}\left(\mathcal{T}\right)\right)$$

– First the *Cartesian product of tables* $\mathcal{T}_i$ is calculated. $\mathcal{T}_i$ denotes the table that was assigned to the type of the ith graph object of r_{LHS}. Formally, $\mathcal{T} = \mathcal{T}_1 \times \cdots \times \mathcal{T}_{n_V+n_E}$, where

$$\mathcal{T}_i = \begin{cases} t(x_i)^d, \text{ when } i \le n_V \text{ and } x_i \in V_{LHS} \\ src(t(z_i))^d, \text{ when } n_V < i \le n_V + n_E \\ \qquad \text{and } u_i \overset{z_i}{\mapsto} v_i \in E_{LHS} \\ t(z_i)^d, \text{ when } n_V < i \le n_V + n_E \\ \qquad \text{and } u_i \overset{z_i}{\rightharpoonup} v_i \in E_{LHS} \end{cases}$$

– *Edge constraints* A pair of equations is defined for each edge of LHS. One such pair expresses that the edge is incident to its source and its target node, respectively. (As the database representation of many-to-many and many-to-one links differ from each other, the corresponding pairs of edge constraints have to be obviously different in their structure.) The conjunction of these equations constitute edge constraints $Edge$. Formally,

$$Edge_{one} = \bigwedge \left\{\, z^{cs}.id = u^{cs}.id \wedge z^{cs}.t(z)^d = v^{cs}.id \mid u \overset{z}{\mapsto} v \in E_{LHS} \,\right\}$$

$$Edge_{many} = \bigwedge \left\{\, z^{cs}.src = u^{cs}.id \wedge z^{cs}.trg = v^{cs}.id \mid u \overset{z}{\rightharpoonup} v \in E_{LHS} \,\right\}$$

The edge constraint of the view can be expressed as $Edge = Edge_{one} \wedge Edge_{many}$.

– *Injectivity constraints* Inj are defined for all pairs of LHS nodes, for which the type of one node is an ancestor of the type of the other. The role of injectivity constraints is always to ensure the injective mapping of graph objects.

$$Inj_{=} \bigwedge \left\{\, x^{cs}_j.id \ne x^{cs}_k.id \mid x_j, x_k \in V_{LHS} \wedge t(x_j) \overset{*}{\leftarrow} t(x_k) \,\right\}$$

– Projection selects all the node identifier columns. Formally,

$$ProjColRefs = x^{cs}_1.id, \ldots, x^{cs}_{n_V}.id$$

– Finally, a renaming is executed. In the result view, the name of each column corresponds to the node from which it originates. Moreover, it stores the identifiers of those objects that were assigned to the original rule graph node by matchings. Note that the result view has as many columns as many nodes its origin rule graph had.

$$ResCols = x^d_1, \ldots, x^d_{n_V}$$

The view for the NACs can be calculated in exactly the same way, but using the NAC graphs in the process. Now we define when a matching is consistent with its database representation.

Definition 42 *Given a model M together with a database representation $\mathfrak{M}$,* ***a matching m for a pattern r_G in model M is consistent with a row $\mathbf{m}^d$ of a view r^d_G in database representation*** *$\mathfrak{M}$ — denoted by $(m|r_G) \cong (\mathbf{m}^d|r^d_G)$ — (i) if the identifiers of all objects of instance model M that have been selected by matching m for pattern r_G can be found as an element in the corresponding position of row $\mathbf{m}^d$, and (ii) for each element of a row $\mathbf{m}^d$ in r^d_G there is a node in pattern r_G that is mapped to the object that corresponds to the given element of the selected row by the matching m. Formally,*

– *there exists matching m for pattern G in model $M \implies \exists \mathbf{m}^d \in r^d_G, \forall x \in V_G : \mathbf{m}^d[x^d] = m(x)^d$*
– *$\exists \mathbf{m}^d \in r^d_G, \forall x \in V_G : \mathbf{m}^d[x^d] = m(x)^d \implies$ there exists matching m for pattern G in model M.*

Note that the above definition is asymmetric as pattern matching requires matching model elements both for nodes and edges of the pattern, while the corresponding row in the view contains only the identifiers of matching objects.

Definition 43 *Given a model M together with a database representation $\mathfrak{M}$,* ***a pattern r_G is consistent with a view r^d_G*** *(denoted by $r_G \cong r^d_G$) if (i) for each matching m of a pattern r_G in instance model M there exists a row $\mathbf{m}^d$ in r^d_G where matching m is consistent with row $\mathbf{m}^d$ and (ii) for each row $\mathbf{m}^d$ in r^d_G there exists a matching m of a pattern r_G where matching m is consistent with row $\mathbf{m}^d$. Formally,*

– $\forall m : G \to M, \exists \mathbf{m}^d \in r^d_G : (m|r_G) \cong (\mathbf{m}^d|r^d_G)$
– $\forall \mathbf{m}^d \in r^d_G, \exists m : G \to M : (m|r_G) \cong (\mathbf{m}^d|r^d_G)$

Finally, we formulate a theorem that states that each possible matching of a LHS (or NAC) rule graph corresponds to exactly one row in the r^d_{LHS} (or r^d_{NAC}) view. Furthermore, the row in the view contains the identifiers of objects and links that participate in the matching.

Theorem 2 *Let d be a bidirectional mapping between $\mathfrak{S}_{GT}$ and $\mathfrak{S}_{DB}$. If model M is consistent with the database representation $\mathfrak{M}$, then a pattern r_G (*without *negative application condition) in $\mathfrak{S}_{GT}$ is consistent with view r^d_G in $\mathfrak{S}_{DB}$. Formally, $M \cong \mathfrak{M} \implies r_G \cong r^d_G$.*

5.3 *Left joins for preconditions of rules.*

As it has been introduced in Sec. 2.2, the calculation of a view for the precondition of a rule proceeds as follows. Each NAC is left outer joined to the LHS graph one by one by using join conditions, which express that columns representing the same shared node in different rule graphs should be equal. Additional filtering conditions require that columns of NAC(s), which are shared with the LHS part, have to be filled with undefined values. Then a *projection* displays only those columns that originate from LHS. Finally, a column renaming procedure performs an identical redefinition of column names.

Formalization. We suppose that the rule r consists of a LHS and k negative application conditions. Furthermore, the notational shorthand n_V is used for denoting the cardinality of V_{LHS}.

The view generated for the precondition r_{PRE} consists of n_V columns and it can be calculated as follows.

$$r^d_{PRE}(ResCols) = \pi_{ProjColRefs}\left(\sigma_{Null}(\mathcal{S}_k)\right).$$

- *Left outer join.* Each r_{NAC_i} is left outer joined to r_{LHS} one by one using a join condition F_i. Formally, $\mathcal{S}_k = r^d_{LHS} \overset{F_1}{\ltimes} r^d_{NAC_1} \overset{F_2}{\ltimes} \ldots \overset{F_k}{\ltimes} r^d_{NAC_k}$
- *Join conditions* F_i express that shared nodes cannot be mapped to different objects in the model M by matching functions m of r_{LHS} and m' of r_{NAC_i}. Formally,

$$F_i = \bigwedge \{\, r^{cs}_{LHS}.x^d = r^{cs}_{NAC_i}.x^d \mid x \in V_{LHS} \cap V_{NAC_i} \,\}$$

  Note that the fact that column name x^d appearing in several tables only denotes that those columns represent the same (shared) node of the rule graph in tables r^d_{LHS} and $r^d_{NAC_i}$.
- *Null conditions* $Null$ express that it is not allowed to have matchings for any r_{NAC_i} in order to have a matching for r_{PRE}. Formally,

$$Null = \bigwedge \left\{ r^{cs}_{NAC_i}.x^d = \varepsilon \mid i \in \mathbb{Z}^+_k \wedge x \in V_{LHS} \cap V_{NAC_i} \right\}.$$

  In this expression, $\mathbb{Z}^+_k$ denotes positive integers up to k.
- *Projection* selects all columns that originate from view r^d_{LHS}. Formally,

$$ProjColRefs = r^{cs}_{LHS}.x^d_1, \ldots, r^{cs}_{LHS}.x^d_{n_V}$$

- Finally, *identical renaming* is implemented. In the result view, the name of each column is the same as the node of the LHS graph from which it originates. Moreover, it stores the identifiers of those objects that were assigned to the LHS graph node by matchings. Note that the result view has as many columns as many nodes r_{LHS} had. Formally,

$$ResCols = x^d_1, \ldots, x^d_{n_V}$$

As a result, each matching for precondition graph r_{PRE} appears as exactly one row in the corresponding view r^d_{PRE}. A row consists of the identifiers of objects that are selected by the matching. In a more formal way, the following theorem can be formulated.

Theorem 3 *Let us suppose that there exists a bijective mapping from $\mathfrak{S}_{GT}$ to $\mathfrak{S}_{DB}$. If model M is consistent with the database representation $\mathfrak{M}$, then a pattern r_{PRE} in $\mathfrak{S}_{GT}$ that* has *negative application condition is consistent with view r^d_{PRE} in $\mathfrak{S}_{DB}$. Formally, $M \cong \mathfrak{M} \Longrightarrow r_{PRE} \cong r^d_{PRE}$.*

5.4 *Graph manipulation in relational databases*

Operations in the graph manipulation phase can be implemented by issuing several data manipulation commands in a single transaction block as it has been explained informally in Sec. 2.2. Note that the database updating algorithm parts should be executed in exactly the same order as it appears in the current section.

Deletions. For each $u_{del} \overset{z_{del}}{\rightarrow} v_{del} \in E_{LHS} \setminus E_{RHS}$, the matched edge $m(u_{del}) \overset{m(z_{del})}{\rightarrow} m(v_{del})$ has to be deleted from the model M. In the database the corresponding edge deletion is performed as follows.

- For each many-to-one edge $u_{del} \overset{z_{del}}{\mapsto} v_{del}$ of the $E_{LHS} \setminus E_{RHS}$ set (line 1), an UPDATE$(src(t(m(z_{del})))^d, id, m(u_{del})^d, t(m(z_{del}))^d, \varepsilon)$ operation (line 2) is executed.
- For each many-to-many edge $u_{del} \overset{z_{del}}{\leftrightarrows} v_{del}$ of the $E_{LHS} \setminus E_{RHS}$ set (line 4), a DELETE$(t(m(z_{del}))^d, src, m(u_{del})^d, trg, m(v_{del})^d)$ operation (line 5) is executed.

Algorithm 2 Edge deletion

```
Require: $\exists \mathbf{r} \in r^d \wedge \exists m_r \wedge (m_r|r) \cong (\mathbf{r}|r^d)$
1: for all $u_{del} \overset{z_{del}}{\mapsto} v_{del} \in E_{LHS} \setminus E_{RHS}$ do
2:   UPDATE $src(t(m(z_{del})))^d$
     SET $t(m(z_{del}))^d = \varepsilon$ WHERE $id = m(u_{del})^d$
3: end for
4: for all $u_{del} \overset{z_{del}}{\leftrightarrows} v_{del} \in E_{LHS} \setminus E_{RHS}$ do
5:   DELETE FROM $t(m(z_{del}))^d$
     WHERE $src = m(u_{del})^d$ AND $trg = m(v_{del})^d$
6: end for
```

If $x_{del} \in V_{LHS} \setminus V_{RHS}$, then its image $m(x_{del})$ and all the dangling edges (i.e., all incident edges) should be removed from the model M. On the database level even the deletion of a single node is performed by issuing a sequence of DELETE operations. One reason why a single DELETE is insufficient is that a node identifier can appear in several node tables because of inheritance in the metamodel. Moreover, node identifiers may appear in tables that represent edges. These latter types of rows should also be deleted in order to ensure that the instance model still remains a graph.

The node deletion algorithm (see Alg. 3) proceeds as follows.

– It iterates through all the nodes of $V_{LHS} \setminus V_{RHS}$ (line 1).
– All types of each node belonging to the difference set are determined, and they get ordered according to the inverse topological order (line 2) to prevent violating foreign key constraints during deletion. (The inverse topological order is a bottom-up style enumeration of the ancestors of a specific type.)
– All the outgoing many-to-many associations A_{out} that have class C as their source node have to be determined. (line 3–5)
  ◦ The appropriate DELETE command can be executed on the tables that correspond to the above-mentioned association. (line 4)
– All the incoming many-to-many associations A_{in} that have class C as their target node have to be determined. (line 6–8)
  ◦ A similar DELETE command has to be executed on the tables that correspond to the above-mentioned association. (line 7)
– All the incoming many-to-one associations A_{in} that have class C as their target node have to be determined. (line 9–11)
  ◦ An UPDATE command has to be executed on the tables that correspond to the source nodes of the above-mentioned associations. (line 10)
– Finally, the node itself can be deleted from class C (line 12), and the iteration should be continued on the ancestors of C. Note that this step automatically deletes all outgoing many-to-one links, which have been stored in table C^d.

Algorithm 3 Node and dangling edge deletion

```
Require: $\exists \mathbf{r} \in r^d \wedge \exists m_r \wedge (m_r|r) \cong (\mathbf{r}|r^d)$
 1: for all $x_{del} \in V_{LHS} \setminus V_{RHS}$ do
 2:   for all $C \in InverseTopologicalOrder(t(m(x_{del})))$ {List ancestors of $t(m(x_{del}))$ in a bottom-up order} do
 3:     for all $C \overset{A_{out}}{\twoheadrightarrow} D_1 \in Assoc_{M2M}$ {For all outgoing many-to-many associations $A_{out}$ having source class $C$} do
 4:       DELETE FROM $A^d_{out}$ WHERE $src = m(x_{del})^d$
 5:     end for
 6:     for all $D_2 \overset{A_{in}}{\twoheadrightarrow} C \in Assoc_{M2O}$ {For all incoming many-to-many associations $A_{in}$ having target class $C$} do
 7:       DELETE FROM $A^d_{in}$ WHERE $trg = m(x_{del})^d$
 8:     end for
 9:     for all $D_3 \overset{A_{in}}{\mapsto} C \in Assoc_{M2M}$ {For all incoming many-to-one associations $A_{in}$ having target class $C$} do
10:       UPDATE $D^d_3$ SET $A^d_{in} = \varepsilon$ WHERE $A^d_{in} = m(x_{del})^d$
11:     end for
12:     DELETE FROM $C^d$ WHERE $id = m(x_{del})^d$ {Deletes the object itself from $C^d$ and all outgoing many-to-one links, which have been stored in $C^d$}
13:   end for
14: end for
```

Insertions. If a node x_{ins} appears only in RHS, but not in LHS, then a new node (denoted by $m(x_{ins})$) of type $t(x_{ins})$ should be added to the model M.

– The algorithm iterates over each node x_{ins} that appears only in RHS, but not in LHS (line 1–6),
– A new identifier $m(x_{ins})^d$ is generated. (line 2)
– On each ancestor of $t(x_{ins})$ (line 3–5) an INSERT operation is executed. (line 4)

Algorithm 4 Node insertion

```
Require: $\exists \mathbf{r} \in r^d \wedge \exists m_r \wedge (m_r|r) \cong (\mathbf{r}|r^d)$
 1: for all $x_{ins} \in V_{RHS} \setminus V_{LHS}$ do
 2:   $m(x_{ins})^d$ := $GenerateNewIdentifier()$ {Generates identifier for the new node}
 3:   for all $C \in TopologicalOrder(t(x_{ins}))$ {Top-down traversal of class hierarchy ending in $t(x_{ins})$} do
 4:     INSERT INTO $C^d$ $(id)$ VALUES $(m(x_{ins})^d)$
 5:   end for
 6: end for
```

If $u_{ins} \xrightarrow{z_{ins}} v_{ins} \in E_{RHS} \setminus E_{LHS}$, then a new edge $(m(u_{ins}) \xrightarrow{m(z_{ins})} m(v_{ins}))$ of type $t(z_{ins})$ should be added to the model M.

– For each many-to-one edge $u_{ins} \overset{z_{ins}}{\mapsto} v_{ins}$ that can be found in $E_{RHS} \setminus E_{LHS}$ (line 1–3), an UPDATE command should be executed on the table that corresponds to the source node $src(t(z_{ins}))$ of the direct type of the edge.
– For each many-to-many edge $u_{ins} \overset{z_{ins}}{\twoheadrightarrow} v_{ins}$ of $E_{RHS} \setminus E_{LHS}$ (line 4–6), an INSERT command should be executed on the table that corresponds to the type of the edge $t(z_{ins})$. (line 5)

Algorithm 5 Edge insertion

```
Require: $\exists \mathbf{r} \in r^d \wedge \exists m_r \wedge (m_r|r) \cong (\mathbf{r}|r^d)$
 1: for all $u_{ins} \overset{z_{ins}}{\mapsto} v_{ins} \in E_{RHS} \setminus E_{LHS}$ do
 2:   UPDATE $src(t(z_{ins}))^d$
      SET $t(z_{ins})^d = m(v_{ins})^d$ WHERE $id = m(u_{ins})^d$
 3: end for
 4: for all $u_{ins} \overset{z_{ins}}{\twoheadrightarrow} v_{ins} \in E_{RHS} \setminus E_{LHS}$ do
 5:   INSERT INTO $t(z_{ins})^d$ $(src, trg)$
      VALUES $(m(u_{ins})^d, m(v_{ins})^d)$
 6: end for
```

Now we can formulate the final statement that expresses the correct behaviour of our algorithm. This states that if a model M was consistent with its database representation $\mathfrak{M}$, and if we perform modifications on the model by a graph transformation rule and we execute the corresponding updating algorithm in the database, then the resulting model M' and the database representation $\mathfrak{M}'$ will still be consistent, yielding that our algorithm built on top of a relational database correctly performs graph transformation.

Theorem 4 *Let us suppose that there exists a bijective mapping d from $\mathfrak{S}_{GT}$ to $\mathfrak{S}_{DB}$. If (i) model M is consistent with the database representation $\mathfrak{M}$, (ii) we have a matching m_r for rule r, together with a corresponding row $\mathbf{m}^d$ in view r^d, and m is consistent with $\mathbf{m}^d$, (iii) rule r is applied on matching m_r resulting in M', and (iv) Algorithms 2–5 are executed in the database for $\mathbf{m}^d \in r^d$ resulting in a database representation $\mathfrak{M}'$, then $M' \cong \mathfrak{M}'$.*

Formally, if

– *(i)* $M \cong \mathfrak{M}$,
– *(ii)* $(m_r|r) \cong (\mathbf{m}^d|r^d)$ *for a pair* $(m_r, \mathbf{m}^d)$,
– *(iii)* $M \overset{r,m_r}{\Longrightarrow} M'$,
– *(iv)* $\mathfrak{M} \overset{Alg.\ 2-5}{\Longrightarrow} \mathfrak{M}'$,

then $M' \cong \mathfrak{M}'$.

6 Implementation issues and experimental evaluation

Now implementation issues of our experimental graph transformation engine are discussed.

Implementation issues. We have already implemented a prototype version of our graph transformation engine. The engine is written entirely in Java and it uses the standard JDBC interface to communicate with the underlying relational database, which was MySQL version 4.1.7. and PostgreSQL 8.0.3 in our case.

The initial phase of a standard application scenario is as follows.

1. Our engine connects to the database and automatically builds the database schema from the metamodel by issuing the appropriate data definition commands to create tables and foreign key constraints as discussed in Sec. 5.1.
2. Then for each rule, queries and data manipulation commands are automatically generated from the rule descriptions for representing graph transformation activities in the pattern matching and the updating phase, respectively. These SQL commands are stored as `PreparedStatement`s as their structure does not change during their application.
3. Finally, during a traversal of the initial instance model, the tables are filled by using the data manipulation commands.

As the current version of the engine is a prototype, the decision on selecting a standard interface (e.g. JMI, MDR) for representing the input (i.e., metamodels, instance models, LHS, RHS and NAC graphs) has been postponed to a later phase of development. As a consequence, their current representation uses an own graph structure implemented in Java.

The initialization phase is followed by the normal operation phase that performs graph transformation. During this phase, the user can call the following methods of the rule to be applied.

1. The `match()` method executes the prepared queries and it collects (and returns) the actual matching of LHS nodes to objects. (A repeated invocation of the `match()` method provides the next matching, if such exists.)
2. Method `match(Map m)` allows the user to define a partial matching `m`, which is extended by our engine to yield a complete matching as a result. In this case, the queries have to be constructed at run-time to be able to express the additional constraints posed by matching `m`. (Note that this specific operation is not discussed in details in Sec. 5, however, its handling is obvious by adding some equations to the WHERE clause of the SELECT query.)
3. The `update(Map m)` method gets the actual matching `m` as its parameter, and it executes the prepared data manipulation statements that reflect activities of the updating phase of graph transformation.
4. The `apply()` method performs a standard rule application step, which consists of a pattern matching phase (i.e., a call of method `match()`) followed by an updating phase (i.e., execution of `update(m)`) on the selected matching `m`.
5. The `applyParallel()` method applies the given rule in parallel, which means that model updates are performed in a transaction block in order to avoid re-evaluation of matchings during the transaction. Parallel execution is implemented by iterative calls of `match()` and `update(m)` methods which are placed inside a single database transaction.
The transaction handling subsystem of the underlying database engine makes a snapshot of the possible matchings before any modification is performed. The iterative calls of `match()` and `update(m)` methods always use this snapshot without recalculating the database content before the successive pattern matching, which means that modifications cannot influence the set of matchings. As a consequence, modifications can be applied in any order always yielding the same derived model, which means that the current implementation achieves real parallelism. As methods `match()` and `update(m)` are applied once on each matching, termination is guaranteed by the finiteness of the snapshot.
Note that parallel independence is not checked by our approach. If the method `applyParallel()` is executed in a conflicting situation, the parallel rule application may not have any serial equivalent sequential rule applications.

The implementation of the approach follows the example presented in Sec. 2 and not the mathematical descriptions as the set of possible values of an attribute can be constrained by simply defining an appropriate built-in type for the column that represents the given attribute.

Experimental results. Since graph transformation can be used for different scenarios in several fields, a detailed quantitative performance comparison of graph transformation tools requires extensive examinations to determine, in which situation a tool has a good performance. As our aim is to present a new technique to implement a graph transformation engine built on top of a relational database, the performance analysis of GT tools is out of scope of this paper. However, a comprehensive study on such performance analysis can be found in [34]. Instead of such a wide-range comparison of graph transformation tools, we focus on such properties of our approach that are expected to have a significant impact on runtime performance or that are specific to our database related solution.

By using the terminology defined in [34], we selected the object-relational mapping as a benchmark example for our current measurements, which can be considered as an incarnation of a typical model transformation scenario. In order to fix a test set, which is a complete, deterministic, but parametric specification, the structure of the initial model and the transformation sequence have to be fixed up to numerical parameters. In our case, the number of Classes in the initial instance model (denoted by N) is selected as the single numerical parameter.

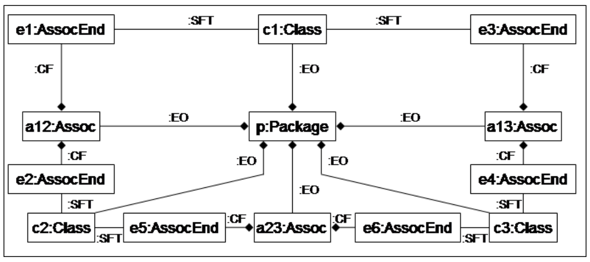


Fig. 6 Initial model of the test case for the $N = 3$ case

The structure of the initial model is presented in Fig. 6 for the $N = 3$ case. The model has a single Package that contains N classes. An Association and 2 AssociationEnds are added to the model for each pair of Classes, thus initially, we have $N(N-1)/2$ Associations and $N(N-1)$ AssociationEnds. Associations are also contained by the single Package as expressed by the corresponding links of type EO. Each AssociationEnd is connected to a corresponding Association and Class by a CF and SFT link, respectively.

The transformation sequence consists of 4 macro steps that are executed in this specific order. The first macro step is a single application of the SchemaR (Fig. 3(a). This is followed by a macro step that consists $N(N-1)/2$ applications of rule AssociationR (Fig. 3(e)). Then Classes are transformed by the execution of rule ClassR (Fig. 3(b)) for N times. Finally, a macro step of length $N(N-1)$ follows, which prescribes the application of rule AssocEndR (Fig. 3(f)).

This test set can be characterized by large patterns and a large number of possible matchings for a rule. The remaining two paradigm features (i.e., the maximum degree of nodes (fan-out) and the length of the transformation sequence) depend on parameter N.

According to our earlier analysis reported in [34], the most significant speed-up could be observed in case of a database related approach when *parallel rule execution* is used as an optimization strategy. As a consequence, only this tool feature is included into our current experiments. In case of parallel rule execution all matchings of a rule are calculated in the pattern matching phase, and then updates are performed as a transaction block on the collected matchings without re-evaluating valid matchings during the transaction.

We identified an additional optimization possibility that is specific to a graph transformation approach that is based on top of a relational database. This database specific feature is the *application of the built-in query optimizer* of the underlying RDBMS. Note that the query built for the precondition of a graph transformation rule has a special structure, for which the built-in query plan generator, which is optimized for handling general queries, may not provide an optimal solution as it lacks the additional information about the structure of GT rules or models. Since some relational databases allow the definition of such queries, for which the generated plan can be influenced from outside the RDBMS, the examination of this optimization possibility has been included into our measurements.

As two orthogonal features have been identified, we performed our measurements on all the four possible combinations of these features, which means that four test cases have been analyzed. The parameter N was fixed to 10 and 30 in test cases where rules were executed sequentially, and N was set to 10, 30, 50 and 100 for test cases with parallel rule application feature having been switched on.

Two popular RDBMSs (namely MySQL version 4.1.7 and PostgreSQL version 8.0.3) took part in our measurements, which were performed on a 1500 MHz Pentium machine with 768 MB RAM. A Linux kernel of version 2.6.7 served as an underlying operating system. The execution time results are shown in Table 1.

	ObjRel	Class	Model	TS	MySQL				PostgreSQL			
			size	length	db		own		db		own	
					match	update	match	update	match	update	match	update
		#	#	#	msec	msec	msec	msec	msec	msec	msec	msec
assocEndRule	parallel OFF	10	1342	146	24.23	2.91	29.45	3.50	27.63	4.40	53.40	4.46
		30	12422	1336	543.41	2.74	549.97	2.73	127.22	6.39	679.81	5.15
	parallel ON	10	1342	146	0.23	3.28	0.23	3.39	2.60	6.23	1.00	4.07
		30	12422	1336	0.13	2.83	0.40	2.40	0.40	5.97	0.80	6.14
		50	34702	3726	0.37	3.93	0.14	5.22	0.26	4.77	1.53	5.34
		100	139402	14951	0.12	4.24	0.12	4.68	0.58	7.69		
associationRule	parallel OFF	10	1342	146	12.20	4.82	13.60	5.18	5.57	5.60	4.29	6.72
		30	12422	1336	160.20	2.94	159.41	2.96	37.20	4.90	48.62	5.62
	parallel ON	10	1342	146	0.38	4.43	0.26	6.13	0.22	6.05	0.26	5.61
		30	12422	1336	0.12	2.91	0.11	2.98	0.08	5.90	0.09	3.77
		50	34702	3726	0.10	2.71	0.10	3.24	0.08	8.19	0.08	8.03
		100	139402	14952	0.08	4.43	0.07	4.88	0.06	6.39		
classRule	parallel OFF	10	1342	146	13.17	2.68	14.28	3.14	7.29	5.31	5.86	5.41
		30	12422	1336	249.38	3.04	247.82	2.68	32.95	5.08	32.91	5.01
	parallel ON	10	1342	146	1.33	2.94	1.35	2.94	0.82	4.81	0.81	4.86
		30	12422	1336	7.41	2.38	7.44	2.35	1.25	4.07	1.09	4.12
		50	34702	3726	39.78	1.99	38.32	2.04	1.99	3.80	2.00	3.74
		100	139402	14951	262.40	2.00	268.99	1.95	8.37	3.62		

Table 1 Experimental results

The head of a row (i.e., the first two columns) shows the name of the rule and the optimization strategy settings for the single tool feature (i.e., parallel rule execution) on which the average is calculated. (Note that a rule is executed several times in a run.) The third column (Class) depicts the number of classes in the run, which is, in turn, the runtime parameter

N for the test case. The fourth and fifth columns show the concrete values for the model size and the transformation sequence length, respectively. Heads of the remaining columns unambiguously identify the RDBMS used and the status denoting whether the built-in query optimizer was used (db) or not (own). Values in match and update columns depict the average times needed for a single execution of a rule in the pattern matching and updating phase, respectively. Execution times were measured on a microsecond scale, but a millisecond scale is used in Table 1 for presentation purposes. Light grey areas denote run-time failures due to exceeding the default memory allocation limits of the operating system.

Our initial experiments can be summarized as follows.

- In accordance with our assumptions, parallel rule execution has a dramatic effect on pattern matching. The time increase for rule ClassR can be explained by having a constant initialization and resource allocation time, which is distributed over a relatively small number of rule applications.
- We have been forced into using temporary tables instead of views in case of MySQL version 4.1.7 as it does not support the concept of views. This obligate choice has a strong negative impact in case of sequential rule execution on the performance of the graph transformation engine as temporary tables are always stored on disks in contrast to views (of PostgreSQL), which are calculated in the memory in general.
- The update phase is slightly longer for PostgreSQL, but the difference cannot be considered significant as the execution times for both databases are of the same order of magnitude.
- The results for query plans own being generated and injected by the GT engine may deviate in both directions from the results of plans db that have been created by the query optimizer. This observation indicates that it is possible to create queries with better performance than the ones that are produced by RDBMS, which is an argument for doing further research on generating special queries optimized for GT rules.
- In contrast to our assumptions, MySQL does not allow manual influence on query plan generation, which is indicated by the similar values in its db and own columns.

7 Related work

Related work can be grouped into two main categories depending on the topics that are also covered in this paper. One category concerns the integration of graph transformation and relational database techniques. The other category focuses on different pattern matching techniques.

Graph transformation and databases. During the past years, intensive research has been focusing on how graph transformation could be adapted as a visual query and data manipulation language for databases. The following list is a brief selection of some main results in the field.

- GRAS [15] is a graph-oriented database management system developed at the University of Aachen, which served as the underlying database for the PROGRES [27] graph transformation tool. It uses a different underlying data model (based on attributed graphs), instead of the relational data model used in our approach.
  However, a recent version of the GRAS database (namely GRAS/GXL [5]) aims to define an interface that provides access to RDBMSs for graph based tools (e.g., PROGRES).
- Andries and Engels propose in [2] a hybrid (visual and textual) query language together with a method, which translates hybrid queries into traditional textual queries by graph transformation.
  In their approach, (i) the graphical part of hybrid queries was based on an E/R diagram notation, (ii) while the target (textual) language was an object-relational extension of SQL. (iii) For graph transformation they employed the above-mentioned PROGRES tool. (iv) Generated SQL queries used the concept of subqueries for expressing restrictions posed by the graph structure.
- In [14], Jahnke and Zündorf propose the use of triple graph grammars [25] for database re-engineering of legacy systems in their Varlet framework. In their approach, again PROGRES was used as a graph transformation engine, and it translated the database schema described by an E/R diagram to an object-oriented conceptual model.

It is common in all these approaches that they investigate how graph transformation can contribute to object-relational database design or to other database related tasks, such as translating hybrid queries to textual ones. Another common feature is that they all use a graph-oriented underlying database (namely GRAS).

In contrast, our proposal is to examine how the mature theory and practice of RDBMSs can potentially contribute to the paradigm of graph transformation. In our approach, a plain relational DBMS was used as an underlying database.

Graph pattern matching approaches. Typically, the most critical phase of a graph transformation step is graph pattern matching, i.e., to find a single (or all) occurrence(s) of a given LHS graph in a host model. Pattern matching techniques of existing graph transformation tools can be grouped into two main categories. For further comparison of graph transformation approaches see [26].

- Algorithms based on *constraint satisfaction* (such as [16] in AGG [9], VIATRA [32]) interpret the graph elements in the LHS pattern of a rule as variables which should be instantiated by fulfilling the constraints imposed by the elements of the instance model and the pattern itself. Our implementation also falls into this category.
- Algorithms based on *local searches* start from matching a single node and then extending the matching step-by-step by neighboring nodes and edges. Several optimizations can be carried out to derive good search plans from

graph transformation rules. The graph pattern matching algorithm of PROGRES (with sophisticated search plans [35]), Dörr's approach [7], and the object-oriented solution in FUJABA [19] fall in this category.

8 Conclusion and Future Work

In the paper, we proposed a new graph transformation engine based on off-the-shelf relational databases to support model transformations between modeling languages. Complex graph queries were implemented as database views defined by join operations constructed according the patterns of the graph transformation rules. Model manipulation statements were translated into elementary insert, delete and update database operations.

We carried out several benchmark test cases to evaluate the performance of our approach based on relational databases in itself. We assessed the overall impact of (i) parallel rule applications (ii) RDBMS-specific query optimization techniques and (iii) the choice of the underlying RDBMS.

Further benchmarks were carried out in [34] to compare the performance of different graph transformation tools based on fundamentally different implementation strategies. These experiments also demonstrated that that relational databases provide a feasible candidate as an implementation framework for graph transformation engines with promising performance results.

However, performance is not the only aspect one needs to consider from a practical point of view when implementing model transformations. Our relational database approach automatically provides persistence and transaction services without further programming effort.

Persistence is very important in the case of MDA tools storing their UML models in relational databases as e.g. AMEOS of Aonix [3]. This tool offers a powerful built-in means to capture model-to-code transformations, but model-to-model transformations (including model manipulations) are not supported, which could be complemented by our technique to provide a general solution.

While model transformations served as the focal application field for the current paper, another interesting future field of our technique is EJB-based solutions. Enterprise Java Beans (EJB) is one of the most fundamental parts of the Java 2 Enterprise Edition (J2EE) platform, which defines a layered architecture for scalable, distributed application development. EJB contains an object-oriented data query language (called EJB-QL), which shows close resemblance with SQL. Therefore, business queries and operations using EJB-QL and accessed via a Java interface could be generated automatically with minor changes to our current approach. First experiments in this direction have been carried out in [4].

Finally, further optimizations are required if we aim at incremental transformations in the future. Despite the fact that incremental updating techniques are subject to research in many fields (e.g. database view recalculation [12], expert systems [11]), there are still only a few RDBMSs that implement incremental view updating even with strong restrictions. MySQL and PostgreSQL do not support this feature at all, which was the main reason for recalculating the views from scratch in each step.

References

1. Aditya Agrawal, Gabor Karsai, and Feng Shi. Graph transformations on domain-specific models. Technical Report ISIS-03-403, Institute for Software Integrated Systems, Vanderbilt University, November 2003.
2. M. Andries and G. Engels. A hybrid query language for the extended entity relationship model. *Journal of Visual Languages and Computing*, 8(1), 1997.
3. Aonix. Ameos framework. http://www.aonix.com/ameos.html.
4. András Balogh, Gergely Varró, Dániel Varró, and András Pataricza. Generation of platform-specific model transformation plugins for EJB 3.0. Accepted to SAC 2006.
5. B. Böhlen. Specific graph models and their mappings to a common model. In *Proc of the 2nd International Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE)*, volume 3062 of *LNCS*, pages 45–60. Springer Verlag, 2003.
6. E. F. Codd. A relational model for large shared data bank. *Communications of the ACM*, 13(6):377–387, June 1970.
7. Heiko Dörr. *Efficient Graph Rewriting and Its Implementation*, volume 922 of *LNCS*. Springer-Verlag, 1995.
8. Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Handbook on Graph Grammars and Computing by Graph Transformation, volume 2: Applications, Languages and Tools*. World Scientific, 1999.
9. C. Ermel, M. Rudolf, and G. Taentzer. *In [8], chapter The AGG-Approach: Language and Tool Environment*, pages 551–603. World Scientific, 1999.
10. T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the Unified Modeling Language. In G. Rozenberg G. Engels, editor, *Proc. of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT)*, volume 1764 of *LNCS*, pages 296–309. Springer Verlag, 1998.
11. C. L. Forgy. RETE: A fast algorithm for the many pattern/many object match problem. *Artificial Intelligence*, 19:17–37, 1982.
12. Ashish Gupta and Inderpal Singh Mumick, editors. *Materialized Views: Techniques, Implementations, and Applications*. MIT Press, June 1999.
13. Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3/4):287–313, 1996.
14. Jens H. Jahnke, Wilhelm Schäfer, Jörg P. Wadsack, and Albert Zündorf. Supporting iterations in exploratory database reengineering processes. *Science of Computer Programming*, 45(2-3):99–136, 2002.
15. Norbert Kiesel, Andy Schürr, and Bernhard Westfechtel. GRAS, a graph-oriented database system for (software) engineering applications. In Jarzabek Lee, Reid, editor, *Proc. CASE '93, 6th Int. Conf. on Computer-Aided Software Engineering*, pages 272–286. IEEE Computer Society Press, 1993.
16. J. Larrosa and G. Valiente. Constraint satisfaction algorithms for graph pattern matching. *Mathematical Structures in Computer Science*, 12(4):403–422, 2002.
17. M. Minas. Concepts and realization of a diagram editor generator based on hypergraph transformation. *Science of Computer Programming*, 44(2):157–180, 2002.

18. Bruce Momjian. *PostgreSQL: Introduction and Concepts*. Addison-Wesley, 2000.
19. U. Nickel, J. Niere, and A. Zündorf. The FUJABA environment. In *The 22nd International Conference on Software Engineering (ICSE)*, pages 742–745, Limerick, Ireland, June 2000. ACM Press.
20. John Poole, Dan Chang, Douglas Tolbert, and David Mellor. *Common Warehouse Metamodel*. John Wiley & Sons, Inc., 2002.
21. QVT Partners. *Revised submission for MOF 2.0 Query/Views/Transformations RFP*, August 2003. `http://qvtp.org/`.
22. R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 3rd edition, 2002.
23. Arend Rensink. The GROOVE simulator: A tool for state space generation. In J. Pfalz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485. Springer-Verlag, 2004.
24. Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, volume 1: Foundations*. World Scientific, 1997.
25. Andy Schürr. Specification of graph translators with triple graph grammars. In *Proc. of the 20th Intl. Workshop on Graph-Theoretic Concepts in Computer Science (WG 1994)*, volume 903 of *LNCS*, pages 151–163. Springer, 1995.
26. Andy Schürr. *In [24]*, chapter Programmed Graph Replacement Systems, pages 479–546. World Scientific, 1997.
27. Andy Schürr, Andreas J. Winter, and Albert Zündorf. *In [8], chapter The PROGRES Approach: Language and Environment*, chapter 13, pages 487–550. World Scientific, 1999.
28. Jon Stephens and Chad Russell. *Beginning MySQL Database Design and Optimization: From Novice to Professional*. Apress, October 2004.
29. Gabriele Taentzer and Arend Rensink. Ensuring structural constraints in graph-based models with type inheritance. In Maura Cerioli, editor, *Proc. 8th Int. Conf on Fundamental Approaches to Software Engineering (FASE 2005)*, volume 3442 of *LNCS*, pages 64–79. Springer Verlag, 2005.
30. J. D. Ullman, J. Widom, and H. Garcia-Molina. *Database Systems: The Complete Book*. Prentice Hall, 2001.
31. Hans Vangheluwe, Juan de Lara, and Pieter J. Mosterman. An introduction to multi-paradigm modelling and simulation. In F. Barros and N. Giambiasi, editors, *Proc. of the AIS'2002 Conference (AI, Simulation and Planning in High Autonomy Systems)*, pages 9–20, April 2002.
32. Dániel Varró, Gergely Varró, and András Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, 44(2):205–227, August 2002.
33. Gergely Varró, Katalin Friedl, and Dániel Varró. Graph transformation in relational databases. In *Int. Workshop on Graph-Based Tools (GraBaTs)*, October 2004. `http://tfs.cs.tu-berlin.de/grabats/`.
34. Gergely Varró, Andy Schürr, and Dániel Varró. Benchmarking for graph transformation. In *Proc. of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 79–88, Dallas, Texas, USA, September 2005.
35. Albert Zündorf. Graph pattern-matching in PROGRES. In *Proc. 5th Int. Workshop on Graph Grammars and their Application to Computer Science*, volume 1073 of *LNCS*, pages 454–468. Springer-Verlag, 1996.

A Proofs of Theorems

Theorem 1 *The initial instance model M and its database representation $\mathfrak{M}$ are consistent. Formally, $M \cong \mathfrak{M}$.*

Proof In order to prove the consistency of M and $\mathfrak{M}$, we have to check whether statements in its definition hold in both directions for all classes and associations.

Nodes. $\Longrightarrow$ First we check the property that should be hold for the classes. Let us select an arbitrary class $C \in V_{MM}$.

According to the left part of Def. 41, $\exists c \in V_M$ such that $C \overset{*}{\leftharpoondown} t(c)$. Since topological order (Def. 39) enumerates all the ancestors of $t(c)$, C will surely appear in the topological order of $t(c)$. But Alg. 1 iterates over all objects (lines 1–6), then over all classes appearing in the topological order (lines 3–5), line 4 is also executed for the object c, class C pair, which means that the identifier c^d generated for c in line 2 should be contained by table C^d in column id after the termination of Alg. 1. The same statement is valid for any arbitrary class of the metamodel.

Many-to-one edges. $\Longrightarrow$ Now we have a many-to-one link $a \overset{e}{\mapsto} b \in E_M$. When Alg. 1 reaches line 7, the source object a of this link has already a database representation, which means that there exists a row $\mathbf{a}$ with $\mathbf{a}[id] = a^d$ in all tables that correspond to ancestors of class $t(a)$. As $src(t(e)) \overset{*}{\leftharpoondown} t(a)$ holds according to the type conformance requirements of Def. 6 for source objects, there exists a row $\mathbf{a}$ with $\mathbf{a}[id] = a^d$ in table $src(t(e))^d$. But the update operation in line 8 of Alg. 1 is executed for our selected many-to-one link, which sets $\mathbf{a}[t(e)^d]$ to b^d, thus we have found an appropriate row $\mathbf{a}$ required by Def. 41.

Many-to-many edges. $\Longrightarrow$ It can be assumed that we have a many-to-many link $a \overset{e}{\rightharpoondown} b \in E_M$. Since lines 10–12 are executed for all many-to-many links of the instance model, it should also be executed for $a \overset{e}{\rightharpoondown} b$ as well, which includes the insertion of tuple (a^d, b^d) to table $t(e)^d$ in line 11. But we are ready now, since (a^d, b^d) got into the table $t(e)^d$ as it is required in the right side of Def. 41. $\square$

Nodes. $\Longleftarrow$ Let us select an arbitrary class $C \in V_{MM}$ again. By using the statement of consistency definition (Def. 41) for a class C, it may be assumed that $\exists \mathbf{c} \in C^d$ such that $\mathbf{c}[id] = c^d$, thus there is a row $\mathbf{c}$ in table C^d that contains the value c^d in column id. Since table C^d was empty in the beginning, the only possibility for c^d to appear in the table is that it should be inserted during the execution of lines 1–6 of Alg. 1. But this could only happen, if object c and class C have been enumerated in line 1 and in line 3, respectively. Since class C has to be in the topological order of $t(c)$, this means that $C \overset{*}{\leftharpoondown} t(c)$. But in this case we have found an object c for which $C \overset{*}{\leftharpoondown} t(c)$ holds, so it fulfils the requirements appearing in the left part of Def. 41. Since in the beginning an arbitrary class was selected, our proof is valid for all other classes as well.

Many-to-one edges. $\Longleftarrow$ It can be assumed that table $\mathcal{T}$, which corresponds to a class in the metamodel, has a row $\mathbf{a}$ for which $\mathbf{a}[id] = a^d$ and $\mathbf{a}[t(e)^d] = b^d$ hold. Since all tables were initially empty and only line 8 of Alg. 1 is able to modify such table $\mathfrak{T}$ in columns other than id, this part of the algorithm has to be executed. But this can only happen, if there exists a many-to-one link $a \overset{e}{\mapsto} b$ in model M.

Many-to-many edges. $\Longleftarrow$ We know that there exists a row $\mathbf{e} = (a^d, b^d)$ in a table $t(e)^d$. Since tables were empty initially, $\mathbf{e}$ had to be inserted during one execution of lines 10–12 of Alg. 1, which means that there should exist a many-to-many link $a \overset{e}{\rightharpoonup} b$ in the original instance model M for which the corresponding `INSERT` operation could be executed in line 11. $\square$

Theorem 2 *Let d be a bidirectional mapping between $\mathfrak{S}_{GT}$ and $\mathfrak{S}_{DB}$. If model M is consistent with the database representation $\mathfrak{M}$, then a pattern r_G (*without *negative application condition) in $\mathfrak{S}_{GT}$ is consistent with view r_G^d in $\mathfrak{S}_{DB}$. Formally, $M \cong \mathfrak{M} \Longrightarrow r_G \cong r_G^d$.*

Proof ($\Longrightarrow$) When proving in this direction, we may assume that we have a matching m for rule graph r_G in model M, and we want to prove that there exists a corresponding row in view r_G^d.

Since $M \equiv \mathfrak{M}$ we know that the instance model has a correct representation in the database. During the proof we first examine what the contents of database tables are, and then we apply operations defined in the query for r_G^d step-by-step, and our aim is to prove that the result (namely the r_G^d view) will contain a row $\mathbf{r}$ with object identifiers defined by matching m.

Consequences of $M \equiv \mathfrak{M}$. Having a matching m means that for all nodes and edges of the G graph have a type conform image in the model M.

(i) Let us use the consistency definition (Def. 41) in left to right direction for any object $m(x) \in V_M$ that participates in the matching m. We get that a corresponding row $\mathbf{m}_x$ with $\mathbf{m}_x[id] = m(x)^d$ should be contained not only by table assigned to its own direct type $t(m(x))^d$ but also by all its ancestor tables, and as such $\mathbf{m}_x \in t(x)^d$ as well. (ii) By applying the consistency definition for many-to-one link $a \overset{e}{\mapsto} b$ assigned to an edge $u \overset{z}{\mapsto} v$ of rule graph G by matching m, we get that table $src(t(e))^d$ has a row $\mathbf{m}_z$ for which $\mathbf{m}_z[id] = a^d$ and $\mathbf{m}_z[t(e)^d] = b^d$ hold. Since $t(e) = t(z)$, $\mathbf{m}_z$ appears in $src(t(z))^d$ as well. (iii) By using the consistency definition for many-to-many link $a \overset{e}{\rightharpoonup} b$ assigned to an edge $u \overset{z}{\rightharpoonup} v$ of rule graph G by matching m, we get that table $t(e)^d$ has a row $\mathbf{m}_z = (a^d, b^d)$. It is worth to emphasize that at this point we already know the contents of all database tables that are used in the query of r_G^d.

Construction of the joined table. Now, if we enumerate nodes and edges of G in their natural order (and also take care of nodes being ahead of edges in the enumeration), and we select exactly the same rows from the tables that were mentioned above, then a row $\mathbf{s} = \Big(\mathbf{m}_{x_1}, \dots, \mathbf{m}_{x_{n_V}}, \mathbf{m}_{z_1}, \dots, \mathbf{m}_{z_{n_E}}\Big)$ will appear in the joined table $\mathfrak{T} = t(x_1)^d \times \cdots \times t(x_{n_V})^d \times t(z_1)^d \times \cdots \times t(z_{n_E})^d$. In the following, it is examined why row $\mathbf{s}$ is not filtered out by injectivity and edge constraints of the selection operation.

Checking injectivity constraints. Let us suppose by contradiction that $\mathbf{s}$ has been filtered out because of violating an injectivity constraint in the query (e.g. $x_j^{cs}.id \neq x_k^{cs}.id$ for some different $x_j, x_k \in V_G$ where $t(x_j) \overset{*}{\leftharpoondown} t(x_k)$ holds). Violating the constraint means that values should be equal in columns $x_j^{cs}.id$ and $x_k^{cs}.id$ for all rows the joined table contains, and as such this equation must also hold for the corresponding elements of $\mathbf{s}$. By taking care of construction rules of $\mathbf{s}$ it yields to $m(x_j)^d = \mathbf{m}_{x_j}[id] = \mathbf{m}_{x_k}[id] = m(x_k)^d$. Since d is bijective, the equation could hold only if, the origins in model M were the same ($m(x_j) = m(x_k)$). But in this case we have different rule graph nodes that have been mapped to the same object of the model by m, which is an immediate violation of injective mapping requirements for m. As a consequence, we may state that if m takes care of injective mapping, then the injectivity filtering condition will also take care of this requirement for the database representation.

Checking edge constraints. Let us select an arbitrary many-to-one edge $u \overset{z}{\mapsto} v \in E_G$ and let us further suppose that it is mapped to link $a \overset{e}{\mapsto} b$ by matching m. As a consequence of the query construction algorithm, we know that $\mathbf{s}[z^{cs}.id] = \mathbf{m}_z[id] = a^d$, and similarly, $\mathbf{s}[z^{cs}.t(z)^d] = \mathbf{m}_z[t(z)^d] = b^d$. Since u and v are rule graph nodes in G, there should exist columns $\mathbf{s}[u^{cs}.id]$ and $\mathbf{s}[v^{cs}.id]$ originating from $\mathbf{m}_u[id]$ and $\mathbf{m}_v[id]$ with values a^d and b^d, respectively. Summarizing our experience results in $\mathbf{s}[u^{cs}.id] = a^d = \mathbf{s}[z^{cs}.id]$ and $\mathbf{s}[v^{cs}.id] = b^d = \mathbf{s}[z^{cs}.t(z)^d]$. Recall the edge constraint that has been defined for edge $u \overset{z}{\mapsto} v$. Note that this specific edge constraint prescribes the equation of exactly the same columns, whose equation has just been proved for $\mathbf{s}$.

Let us select an arbitrary many-to-many edge $u \overset{z}{\rightharpoonup} v \in E_G$ and let us further suppose that it has been mapped to $a \overset{e}{\rightharpoonup} b$ by matching m. By using a similar reasoning, we get equalities $\mathbf{s}[u^{cs}.id] = a^d = \mathbf{s}[z^{cs}.src]$ and $\mathbf{s}[v^{cs}.id] = b^d = \mathbf{s}[z^{cs}.trg]$, which means that $\mathbf{s}$ fulfils the edge constraints defined for edge $u \overset{z}{\rightarrow} v$.

Since $\mathbf{s}$ satisfies all the injectivity and edge constraints we may state that $\mathbf{s} \in \sigma_{Inj \wedge Edge}(\mathfrak{T})$.

Performing projection. By using the definition of projection to columns being defined in Sec. 5.2, we get $\mathbf{r} = \big(m(x_1)^d, \dots, m(x_{n_V})^d\big) \in r_G^d$, which means that we have found a row in r_G^d that contains all the identifiers of nodes that have been selected by the specific matching. $\square$

Proof ($\Longleftarrow$) When proving in this direction, we may assume that table r_G^d having n_V columns contains a row $\mathbf{r}$, for which $\forall x \in V_G : \mathbf{r}[x^d] = c^d$. Now our goal is to define an appropriate matching m for rule r_G in model M.

In this case the idea of the proof goes rather in a backward direction. We already now that the joined table $\mathcal{S}$ contains a row $\mathbf{s}$ from which $\mathbf{r}$ could originate during its calculation, but since the joined table has more columns than the result table, some values in row $\mathbf{s}$ are unknown initially. By using edge constraints, we are able to guess some further values, resulting in a row $\mathbf{s}$ that has more values filled in than $\mathbf{r}$. Then we define the matching m based on the values in row $\mathbf{s}$, and finally we prove that this matching must also satisfy injectivity constraints together with its original database representation.

Following the projection and selection operations in backward direction. Now we have a row $\mathbf{r}$ in r_G^d. If an operation (such as projection and selection) cannot increase the number of rows, then it is sure that if we have a row in the result table, then this row should have an origin in the table, on which operations were performed. Formally, it is obvious (by using the definitions of projection and selection) that $\exists \mathbf{s} \in \sigma_{Inj \wedge Edge}(\mathcal{S}) \subseteq \mathcal{S} = \mathcal{T}_1 \times \cdots \times \mathcal{T}_{n_V + n_E}$, where $\mathcal{T}_i$ is the table that corresponds to the ith graph object (node or edge) of the pattern G as defined by the query construction algorithm. By investigating the columns to which projection was applied, we can guess what the values of row $\mathbf{s}$ should be before the projection was performed. More precisely, $\forall x \in V_G \, : \, c^d = \mathbf{r}[x^d] = \mathbf{s}[x^{cs}.id]$.

Matching definition for rule graph nodes. Let us examine an arbitrary node x of pattern G. According to the definition of $\mathcal{S}$, the column set x^{cs} that corresponds to x should originate from table $t(x)^d$ that was assigned to class $t(x)$. As a consequence, there should exist a row $\mathbf{t}_x$ in table $t(x)^d$ such that $\mathbf{s}[x^{cs}.id] = \mathbf{t}_x[id] = c^d$. Since our tables contain unique identifiers of objects in columns id, there should exist a single object c whose identifier is c^d. Now the consistency definition (Def. 41) can be used in right to left direction, which means that the direct type $t(c)$ of object c is a descendant of $t(x)$, so it is allowed to map node x to object c by matching m. So we can define the matching m for rule graph node x as $m(x) := c$.

Matching definition for many-to-one rule graph edges. Let us select an arbitrary many-to-one edge $u \overset{z}{\mapsto} v$ from pattern G. Recall how edge constraints look like for this specific edge. These constraints are $z^{cs}.id = u^{cs}.id$, and $z^{cs}.t(z)^d = v^{cs}.id$. Note that since u and v are nodes in pattern G, $\mathbf{s}[u^{cs}.id]$ and $\mathbf{s}[v^{cs}.id]$ have some values a^d and b^d being identifiers of objects a and b, respectively, as we determined earlier. Furthermore, we know that $t(u) \overset{*}{\leftarrow} t(a)$ and $t(v) \overset{*}{\leftarrow} t(b)$. Edge constraints must hold for all rows of $\mathcal{S}$ and as such $\mathbf{s}$ should also satisfy them, resulting in $\mathbf{s}[z^{cs}.id] = \mathbf{s}[u^{cs}.id] = a^d$ and $\mathbf{s}[z^{cs}.t(z)^d] = \mathbf{s}[v^{cs}.id] = b^d$. We know that the column set z^{cs} of $\mathcal{S}$ should originate from the table $src(t(z))^d$ that was assigned to class $src(t(z))$. Since $\mathbf{s}$ is in the joined table $\mathcal{S}$, $src(t(z))^d$ should have a row $\mathbf{t}_z$ such that $\mathbf{t}_z[id] = \mathbf{s}[z^{cs}.id] = a^d$ and $\mathbf{t}_z[t(z)^d] = \mathbf{s}[z^{cs}.t(z)^d] = b^d$. The consistency definition (Def. 41) for many-to-one links in right to left direction states that $\exists a \overset{e}{\mapsto} b \in E_M$ such that $t(z) = t(e)$. But this edge is an appropriate candidate to which pattern edge $u \overset{z}{\mapsto} v$ can be mapped by matching m.

Matching definition for many-to-many rule graph edges. Let us select an arbitrary many-to-many edge $u \overset{z}{\rightharpoonup} v$ from pattern G. Edge constraints for this specific edge are $z^{cs}.src = u^{cs}.id$ and $z^{cs}.trg = v^{cs}.id$. Since u and v are nodes of pattern G, $\mathbf{s}[u^{cs}.id]$ and $\mathbf{s}[v^{cs}.id]$ have some values a^d and b^d that are identifiers of objects a and b, respectively. Moreover, we know that $t(u) \overset{*}{\leftarrow} t(a)$ and $t(v) \overset{*}{\leftarrow} t(b)$. Edge constraints must be satisfied by row $\mathbf{s}$, which means that $\mathbf{s}[z^{cs}.src] = \mathbf{s}[u^{cs}.id] = a^d$ and $\mathbf{s}[z^{cs}.trg] = \mathbf{s}[v^{cs}.id] = b^d$ should hold. We know that column set z^{cs} of $\mathcal{S}$ derives from table $t(z)^d$, which has been created for association $t(z)$. Since $\mathbf{s}$ is in table $\mathcal{S}$, there should exist a row $\mathbf{t}_z$ in table $t(z)^d$ such that $\mathbf{t}_z[src] = \mathbf{s}[z^{cs}.src] = a^d$ and $\mathbf{t}_z[trg] = \mathbf{s}[z^{cs}.trg] = b^d$. The consistency definition (Def. 41) for many-to-many links in right to left direction states that there exists a link $a \overset{e}{\rightharpoonup} b \in E_M$ such that $t(z) = t(e)$. Now we may define matching m for edge $u \overset{z}{\rightharpoonup} v$ as $m(u \overset{z}{\rightharpoonup} v) := a \overset{e}{\rightharpoonup} b$.

Injectivity constraint check. Finally, we check that the matching m we have just defined cannot map different nodes (edges) to the same object (link).

Let us suppose by contradiction, that there are two different nodes x_j, x_k in G such that $t(x_j) \overset{*}{\leftarrow} t(x_k)$ and m maps them to the same object c. Formally, $m(x_j) = m(x_k) = c$. Since d is bijective, these objects have the same identifier in the database, formally $m(x_j)^d = m(x_k)^d = c^d$. We have some further knowledge about this identifier, namely $\mathbf{s}[x_j^{cs}.id] = c^d = \mathbf{s}[x_k^{cs}.id]$. Recall that injectivity constraints prescribed inequality for exactly the same columns, namely $x_j^{cs}.id \neq x_k^{cs}.id$. Injectivity constraints should be satisfied by row $\mathbf{s}$ in order to be the origin of row $\mathbf{r}$, which is a contradiction, since we found equality of elements in the mentioned columns in case of row $\mathbf{s}$.

Different pattern edges cannot be mapped to the same link, as in such a situation the pattern could not be a well-formed instance of the metamodel, since it would violate the non-existence of parallel edges. □

Corollary 1 *If we calculate the left outer join of tables $\mathcal{R}^{(m)}$ and $\mathcal{S}^{(n)}$, then for each row $\mathbf{r}$ of $\mathcal{R}$ there exists a row $\mathbf{t}$ in the joined table that contains row $\mathbf{r}$ in its first m columns. Formally, if $\mathcal{T} = \mathcal{R} \overset{F}{\ltimes} \mathcal{S}$ then $\forall \mathbf{r} \in \mathcal{R}, \exists \mathbf{t} \in \mathcal{T}$ such that $\mathbf{t}[i] = \mathbf{r}[i]$ for all the columns of $\mathbf{r}$.*

In the following, notation $\mathcal{S}_i$ will be used for $r_{LHS}^d \overset{F_1}{\ltimes} r_{NAC_1}^d \overset{F_2}{\ltimes} \ldots \overset{F_i}{\ltimes} r_{NAC_i}^d$. With this notation $\mathcal{S}_k$ corresponds to the table that has to be calculated for the view r_{PRE}^d.

Theorem 3 *Let us suppose that there exists a bijective mapping from $\mathfrak{G}_{GT}$ to $\mathfrak{G}_{DB}$. If model M is consistent with the database representation $\mathfrak{M}$, then a pattern r_{PRE} in $\mathfrak{G}_{GT}$ that* has *negative application condition is consistent with view r_{PRE}^d in $\mathfrak{G}_{DB}$. Formally, $M \cong \mathfrak{M} \Longrightarrow r_{PRE} \cong r_{PRE}^d$.*

Proof *($\Longrightarrow$)* The basic idea is to prove that $\mathcal{S}_k$ should contain a row $\mathbf{s}$ that has defined values only in columns that originate from view r_{LHS}^d, and all other values are undefined. This is done in an iterative process starting from $\mathcal{S}_0$, which corresponds to view r_{LHS}^d. In each step in order to generate $\mathcal{S}_i$, $r_{NAC_i}^d$ is attached to $\mathcal{S}_{i-1}$ by a left outer join operation using the formulae F_i for join condition. Finally, we show that the projection and selection performed in the last phases of

r^d_{PRE} calculation does not filter out row $\mathbf{s}$ from the set of results, yielding to an appropriate row $\mathbf{r}$ in view r^d_{PRE}.

Since m is a matching for pattern r_{PRE}, it is also a matching for r_{LHS}. By using Theorem 2, this means that $\exists \mathbf{t_0} \in r^d_{LHS} = \mathcal{S}_0$.

Lemma. Let us suppose by induction that we have already calculated $\mathbf{t_{i-1}} \in \mathcal{S}_{i-1}$ and $\mathbf{t_{i-1}} = (\mathbf{t_0}[x^d_1], \dots, \mathbf{t_0}[x^d_{n_V}], \varepsilon, \dots, \varepsilon)$. In other words the first n_V columns of $\mathbf{t_{i-1}}$ contains the same values as $\mathbf{t_0}$, while all the remaining values are undefined. We want to prove that $\mathbf{t_i}$ has the similar structure and that $\mathbf{t_i}$ can also be found in table $\mathcal{S}_i$.

Proof of the lemma. Let us calculate $\mathcal{S}_i$. By using Corollary 1, it can be stated that columns of $\mathbf{t_i}$ that originate from $\mathcal{S}_{i-1}$ have the same values as $\mathbf{t_{i-1}}$ independently of the fact whether the join condition F_i holds or not. The only thing to be checked is whether the last n_{V_i} columns of $\mathbf{t_i}$ (originating from $r^d_{NAC_i}$) are filled with undefined values.

Let us suppose by contradiction that there exists $\mathbf{r_i}$ in view $r^d_{NAC_i}$ that can be attached to $\mathbf{t_{i-1}}$ by left outer join in such way that F_i holds. By using Theorem 2 there should exist a matching m' for the graph objects of r_{NAC_i}.

If x is an arbitrary shared node (thus $x \in (V_{LHS} \cap V_{NAC_i})$), then because of the construction algorithm of views r^d_{LHS} and $r^d_{NAC_i}$, each of them has a column that represents this object x. But we assumed that F_i is satisfied, which means that $r^{cs}_{LHS}.x^d = r^{cs}_{NAC_i}.x^d$ should hold for all the rows, and as such for $\mathbf{t_i}$ as well. By summarizing our knowledge about $\mathbf{t_i}$ we get

$$\mathbf{t_0}[x^d] = \mathbf{t_{i-1}}[r^{cs}_{LHS}.x^d] = \mathbf{t_i}[r^{cs}_{LHS}.x^d] = \\ \mathbf{t_i}[r^{cs}_{NAC_i}.x^d] = \mathbf{r_i}[x^d].$$

$\mathbf{t_0}[x^d]$ and $\mathbf{r_i}[x^d]$ define the identifiers of objects to which x was mapped by m and m', respectively. Thus, $m(x)^d = \mathbf{t_0}[x^d] = \mathbf{r_i}[x^d] = m'(x)^d$. Since d is bijective, $m(x) = m'(x)$, which means that all the shared nodes of LHS and NAC_i had to be mapped onto the same object.

At this point we know that all the shared nodes of LHS and NAC_i are mapped to the same objects both by m and m', respectively. If the definition of matching for rule r_{PRE} is recalled from Sec. 5.3, then it can be seen that m cannot be a matching, since m and m' together violate the second part of the definition, which prohibits the existence of a matching for NAC_i. So our initial assumption to have a row $\mathbf{r_i}$ that satisfies F_i together with $\mathbf{t_{i-1}}$ failed. But if there are no such row $\mathbf{r_i}$ for which F_i could hold, then only the second part of the left join definition could have been used when calculating $\mathbf{t_i}$, which means that the columns of $\mathbf{t_i}$ originating from $r^d_{NAC_i}$ must be padded with undefined values. At this point we may conclude that we have found a row $\mathbf{t_i}$ in view $\mathcal{S}_i$ that has the prescribed structure.

Consequence of the lemma. By using our lemma k times, we get that there is a row $\mathbf{t_k} \in \mathcal{S}_k$, which contains defined values in columns originating from view r^d_{LHS} and all the other values are undefined.

The effect of selection. Since null conditions $Null$ of the selection operation pose restrictions only on columns originating from negative application condition views $r^d_{NAC_i}$, $\mathbf{t_k}$ surely satisfies all of them, since it contains undefined values in all such columns.

The effect of projection. The last operation is the projection, which selects the first n_V columns of $\mathbf{t_k}$ resulting in a row $\mathbf{r} \in r^d_{PRE}$. Note that the first n_V columns of $\mathbf{t_k}$ are the ones that contain identifiers originating from r^d_{LHS}, and they are never undefined. It can be now concluded that a row $\mathbf{r}$ is found in the view that represents r_{PRE}. $\square$

Proof ($\Longleftarrow$) We know that there exists a row $\mathbf{r}$ in view r^d_{PRE} and an appropriate matching m for rule r_{PRE} is to be found.

Proof by contradiction I. Let us suppose by contradiction that we have $\mathbf{r} \in r^d_{PRE}$, but no matching m exists for the LHS rule graph (r_{LHS}).

If no matchings exist for r_{LHS}, then Theorem 2 yields to an empty r^d_{LHS} view. But note that this view appears at the leftmost position of left join operations in the definition of r^d_{PRE}, which means that r^d_{PRE} should also be empty. But this contradicts to our initial assumption, since $\mathbf{r} \in r^d_{PRE}$.

Proof by contradiction II. Let us suppose by contradiction that we have $\mathbf{r} \in r^d_{PRE}$, and a matching m for r_{LHS}, and there is also a matching m' for a rule graph r_{NAC_i} such that each node and edge are mapped to the same object and link, respectively, by both m and m'.

By using Theorem 2 for matchings m and m', we get that $\mathbf{s_0} \in \mathcal{S}_0 = r^d_{LHS}$ and $\mathbf{r_i} \in r^d_{NAC_i}$. Let us suppose that row $\mathbf{s_k}$ of view $\mathcal{S}_k$ was calculated by using $\mathbf{s_0}$ and $\mathbf{r_i}$. For the sake of simplicity, let us focus only on columns of $\mathbf{s_k}$ that originate from $r^d_{NAC_i}$. Our statement is that this portion of $\mathbf{s_k}$ agrees with $\mathbf{r_i}$.

The portion of $\mathbf{s_k}$ originating from $r^d_{NAC_i}$ is introduced when $\mathcal{S}_i$ is calculated, and afterwards it is left unchanged by left outer join operations. But when the ith left outer join is executed its join condition F_i holds, and in this case inner join has to be executed resulting in our statement mentioned above. The only thing to be checked is why F_i is satisfied. Note that F_i is defined on the shared nodes of r_{LHS} and r_{NAC_i}. Each shared node x is mapped to the same object c by both m and m', so $\mathbf{s_{i-1}}[r^{cs}_{LHS}.x^d] = c^d = \mathbf{r_i}[x^d]$, which means that we have found correspondence in all columns of $\mathbf{s_{i-1}}$ and $\mathbf{r_i}$ for which correspondence was prescribed by F_i.

Note that null conditions require the image of shared nodes of r_{LHS} and r_{NAC_i} to be undefined in columns of $\mathbf{s_k}$ that originate from r_{NAC_i}, which is immediately violated, since they got their values just in the previous paragraph. So $\mathbf{s_k}$ violates null conditions of the selection operation, and as a consequence it should be filtered out inhibiting $\mathbf{s_k}$ to be the origin of $\mathbf{r}$. It means that under the supposed circumstances no origin of $\mathbf{r}$ exists in view $\sigma_{Null}(\mathcal{S}_k)$, which is a contradiction.

Final consequence. At this point we know that there should exist a matching m for r_{LHS}, but no matching m' for any r_{NAC_i}. Recalling the definition of matching of r_{PRE}, we get that the above-mentioned situation is the one that fulfils all the requirements, so matching m is also good for r_{PRE}. $\square$

Theorem 4 *Let us suppose that there exists a bijective mapping d from $\mathfrak{S}_{GT}$ to $\mathfrak{S}_{DB}$. If (i) model M is consistent with the database representation $\mathfrak{M}$, (ii) we have a matching m_r for rule r, together with a corresponding row $\mathbf{m}^d$ in view r^d, and m is consistent with $\mathbf{m}^d$, (iii) rule r is applied on matching m_r resulting in M', and (iv) Algorithms 2–5 are executed in the database for $\mathbf{m}^d \in r^d$ resulting in a database representation $\mathfrak{M}'$, then $M' \cong \mathfrak{M}'$.*

Formally, if

- *(i) $M \cong \mathfrak{M}$,*
- *(ii) $(m_r|r) \cong (\mathbf{m}^d|r^d)$ for a pair $(m_r, \mathbf{m}^d)$,*
- *(iii) $M \overset{r,m_r}{\Longrightarrow} M'$,*
- *(iv) $\mathfrak{M} \overset{Alg.\ 2-5}{\Longrightarrow} \mathfrak{M}'$,*

then $M' \cong \mathfrak{M}'$.

Proof The model manipulation phase of a rule application can be divided into a deletion and an insertion step. Our first goal is to prove that context model M_c is consistent with database representation $\mathfrak{M}_c$ resulted by the execution of Alg. 2 and 3. Then the consistency of derived model M' and database representation $\mathfrak{M}'$ is proven based on the consistency of M_c and $\mathfrak{M}_c$. Since skeletons of the proofs are exactly the same in these steps, we only present the technique for the more difficult (i.e., the deletion) step.

The proof of the deletion step is bidirectional and it has 7 cases in each direction, which use exactly the same technique and which have to be checked one by one. The 7 cases correspond to the deletion of (i) objects; (ii) many-to-one and (iii) many-to-many dangling links leaving an object to be deleted; (iv) many-to-one and (v) many-to-many dangling links leading into an object to be deleted; and (vi) many-to-one and (vii) many-to-many links selected by matching m for an edge $z \in E_{LHS} \setminus E_{RHS}$. We may identify a well-defined part of Alg. 2 and 3 for each case where the specific case is handled by these algorithms in on the database. Table 2 presents the cases and their corresponding handling routines.

Case	Object/link	Reason of selection	DB operation
(i)	object	selected by m	line 12 of Alg. 3
(ii)	many-to-one link	dangling/outgoing	line 12 of Alg. 3
(iii)	many-to-many link	dangling/outgoing	line 4 of Alg. 3
(iv)	many-to-one link	dangling/incoming	line 10 of Alg. 3
(v)	many-to-many link	dangling/incoming	line 7 of Alg. 3
(vi)	many-to-one link	selected by m	line 2 of Alg. 2
(vii)	many-to-many link	selected by m	line 5 of Alg. 2

Table 2 Different cases and corresponding lines of Alg. 2 and 3 participating in the proof

In order to avoid tedious and lengthy proofs of the same style, we only sketch the skeleton of the proof technique and we present a complete proof for only one case (i.e., for the deletion of nodes) in both directions. The proofs for the other cases can be derived from the presented complete proof by replacing object and lines of Algorithms 2 and 3 by a corresponding kind of link and lines of the same algorithms, respectively, as defined in Table 2 for the given case.

Proof ($\Longrightarrow$) The skeleton of the proof is as follows. We select an object (a link) from context model M_c. Since only deletions are performed on model M, M should also contain the same object (link). Then the consistency of M and $\mathfrak{M}$ is used in left to right direction to ensure that the object (link) is represented in the database (i.e., in $\mathfrak{M}$). Finally, it is examined why the database representation of the object (link) cannot be deleted from $\mathfrak{M}$ during the execution of Alg. 2 and 3.

Nodes. Let us select an object c from context model M_c and an arbitrary class $C \in V_{MM}$ such that $C \overset{*}{\leftarrow} t(c)$. Object c has to appear in model M as only deletions were allowed in this step. By using the consistency of model M and database representation $\mathfrak{M}$ (Def. 41) for objects in left to right direction, we get that $\exists \mathbf{c} \in C^d$ such that $\mathbf{c}[id] = c^d$.

The only position where either Alg. 2 or Alg. 3 can delete row $\mathbf{c}$ from C^d is line 12 of Alg. 3. (All other database operations either delete rows from tables assigned to many-to-many associations, or updates tables assigned to classes in columns not equal to id.) Line 12 of Alg. 3 would delete row $\mathbf{c}$, if $\exists x \in V_{LHS} \setminus V_{RHS}$ such that $m(x) = c$, but the existence of such node x would yield to the deletion of object c from model M, which is impossible as context model M_c still contains c. The result of this reasoning is that $\mathbf{c}$ could not be deleted by Alg. 2 and 3, which means that $\mathbf{c} \in C^d$ also in database $\mathfrak{M}_c$. $\square$

Proof ($\Longleftarrow$) Now the proof proceeds in the other direction. We have a row in a table of $\mathfrak{M}_c$, which was assigned to a class (many-to-many association). Since Alg. 2 and 3 can delete rows or set undefined values to columns with name not equal to id, src, trg, it is sure that a row with the same value in column id (in columns src and trg) can be found in the same table of $\mathfrak{M}$. In this case, we may apply the consistency of $\mathfrak{M}$ and M for objects or many-to-one links (for many-to-many links) in right to left direction, resulting in a corresponding object or many-to-one link (many-to-many link) in model M. Finally, it is investigated why this object or many-to-one link (many-to-many link) is not deleted in the deletion phase of GT rule application.

Nodes. We have a row $\mathbf{c'} \in C^{d'}$ with $\mathbf{c'}[id] = c^d$ where $C^{d'}$ represents a table that was assigned to a class $C \in V_{MM}$ and that has a content according to the database representation $\mathfrak{M}_c$. Since only row deletions and updates in columns with name not equal to id could be performed on table C^d during the execution of Alg. 2 and 3, it is sure that $\exists \mathbf{c} \in C^d$ such that $\mathbf{c}[id] = \mathbf{c'}[id] = c^d$. By using the consistency of model M and database representation $\mathfrak{M}$ (Def. 41) for objects in right to left direction, we get that $\exists c \in V_M$ such that $C \overset{*}{\leftarrow} t(c)$.

Let us suppose by contradiction that there is a node $x \in V_{LHS} \setminus V_{RHS}$ such that $m(x) = c$ and $t(x) \overset{*}{\leftarrow} t(c)$. Since $C \overset{*}{\leftarrow} t(m(x)) = t(c)$, class C should have been enumerated in the inverse topological order of $t(m(x))$, and as a consequence, line 12 of Alg. 3 should have been executed on table C^d with condition $id = c^d$, which means that $\mathbf{c}$ should have been removed, as $\mathbf{c}[id] = c^d$. This is a contradiction, since $\mathbf{c}$ remained in table C^d in database content $\mathfrak{M}_c$.

So $\nexists x \in V_{LHS} \setminus V_{RHS}$ that is mapped to c by m. But in this case c is not removed from model M, thus, c remains in context model M_c. $\square$