

List of Transformations for Code and Relational Database Evolution

Ondrej Macek and Karel Richta

May 10, 2014

Contents

1	Notation	4
1.1	State	4
1.2	Transformation Definition	4
2	Application Model	4
3	Database Model	4
3.1	Database Schema	4
3.2	Data	5
3.2.1	Mapping Between Instances	5
4	Software Model	5
5	Transformations	6
5.1	Application Manipulation	6
5.1.1	newApplication	6
5.1.2	addClass	6
5.1.3	addProperty	6
5.1.4	addAssociation	6
5.1.5	renameProperty	7
5.1.6	renameAssociation	7
5.1.7	renameClass	7
5.1.8	removeProperty	7
5.1.9	removeAssociation	7
5.1.10	removeClass	8
5.1.11	addParent	8
5.1.12	removeParent	8
5.1.13	pushDown	8
5.1.14	pullUp	9
5.2	Database Schema Manipulation	9
5.2.1	newDatabase	9
5.2.2	addTable	9
5.2.3	addColumn	9
5.2.4	addForeignKey	10
5.2.5	alterColumnName	10
5.2.6	alterForeignKeyName	10
5.2.7	alterTableName	10

5.2.8	dropColumn	11
5.2.9	dropEmptyColumn	11
5.2.10	dropForeignKey	11
5.2.11	dropEmptyForeignKey	11
5.2.12	dropTable	12
5.2.13	dropEmptyTable	12
5.2.14	copyColumn	12
5.2.15	copyTable	12
6	Evolution	13
6.1	Basic Transformations	13
6.1.1	newSoftware	13
6.1.2	newClass	13
6.1.3	newProperty	13
6.1.4	newAssociation	14
6.1.5	renameProperty	14
6.1.6	renameAssociation	14
6.1.7	renameClass	15
6.1.8	removeProperty	15
6.1.9	removeAssociation	15
6.1.10	removeClass	16
6.1.11	copyProperty	16
6.1.12	moveProperty	17
6.1.13	inlineClass	17
6.1.14	mergeClasses	17
6.1.15	splitClass	18
6.1.16	extractPropertyAsObject	18
6.1.17	inlineObjectAsProperty	19
6.1.18	addParent	19
6.1.19	removeParent	19
6.1.20	pushDown	20
6.1.21	pullUp	20
6.1.22	extractParent	20
6.1.23	extractCommonParent	21
7	Helpers	21
7.1	Application Helpers	21
7.1.1	propertyInParent	21
7.1.2	associationInParent	22
7.1.3	isReferenced	22
7.1.4	isParent	22
7.1.5	selfParentPossibility	23
7.2	Database Helpers	23
7.2.1	selectOne	23
7.2.2	selectAll	23
7.2.3	insertData	23
7.2.4	insertValue	24
7.2.5	isReferenced	24
7.2.6	reference	24
7.2.7	valueOfColumn	24
7.2.8	pairOfColumn	24
7.2.9	pairsColumnValue	24

7.2.10	pairsForeignKeyValue	25
7.2.11	copyPropertyAsTable	25
7.2.12	propToClass	25
7.2.13	children	25

1 Notation

- $X = A^*$ means X is defined as a sequence of elements from A ,
- $X = (A, B)$ means X is a tuple of pairs from A and B ,
- $X = A|B$ means X is either A or B .

1.1 State

To distinguish the initial and final state of the transformation we use the symbol ' (apostrophe) to annotate all elements of the final state. Two variables x, x' represent the same element of the model variable x in the initial state and x' in the final state. If there are no explicitly defined differences between x and x' then we assume $x = x'$. If x is a tuple and there are defined differences only for part of this tuple, then we assume the rest of the tuple stays unchanged.

1.2 Transformation Definition

There is defined only successful processing of a transformation in the following text. All undefined paths results in \perp i.e. in inconsistent state of software.

2 Application Model

$$\mathbf{AppType} = \mathit{APPSTRING} \mid \mathit{APPINTEGER} \mid \mathit{APPBOOLEAN} \quad (1)$$

$$\mathbf{InheritanceType} = \mathit{SINGLETABLE} \mid \mathit{TABLE - PER - CLASS} \mid \quad (2)$$

$$\mathit{JOINED} \quad (3)$$

$$\mathbf{Inheritance} = (\mathit{class}, \mathit{InheritanceType}) \cup \mathit{OBJECT} \quad (4)$$

$$\mathbf{Association} = (\mathit{label}, \mathit{classRef}, \mathit{startCardinality}, \mathit{endCardinality}) \quad (5)$$

$$\mathbf{Property} = (\mathit{label}, \mathit{AppType}, \mathit{DefaultValue}, \mathit{Cardinality}, \mathit{Mandatory}) \quad (6)$$

$$\mathbf{Class} = (\mathit{label}, \mathit{Property}^*, \mathit{Association}^*, \mathit{Inheritance}) \quad (7)$$

$$\mathbf{Application} = (\mathit{Class}^*) \quad (8)$$

$$(9)$$

Only one type of *InheritanceType* – *SINGLETABLE* – is used in definitions of transformations in Sec. 5 for the sake of abbreviation.

3 Database Model

3.1 Database Schema

$$\mathbf{Database} = (\mathit{TableSchema}^*, \mathit{TableData}^*, \mathit{Sequence}) \quad (10)$$

$$\mathbf{TableSchema} = (\mathit{label}, \mathit{primaryKey}, \mathit{Column}^*, \mathit{ForeignKey}^*) \quad (11)$$

$$\mathbf{Column} = (\mathit{label}, \mathit{DbType}, \mathit{DefaultValue}, \mathit{Constraint}^*) \quad (12)$$

$$\mathbf{ForeignKey} = (\mathit{label}, \mathit{TableSchema}, \mathit{Constraint}^*) \quad (13)$$

$$\mathbf{PrimaryKey} = (\mathit{label}) \quad (14)$$

$$\mathbf{DbType} = \mathit{DBSTRING} \mid \mathit{DBINT} \mid \mathit{DBBOOLEAN} \quad (15)$$

$$\mathbf{Constraint} = \mathit{NOTNULL} \mid \mathit{UNIQUE} \quad (16)$$

3.2 Data

$$\mathbf{TableData} = (Table, KeyPair, Pair*) \quad (17)$$

$$\mathbf{KeyPair} = (PrimaryKey, Value) \quad (18)$$

$$\mathbf{Pair} = ColumnValue \mid ForeignKeyValue \quad (19)$$

$$\mathbf{ColumnValue} = (Column, Value) \quad (20)$$

$$\mathbf{ForeignKeyValue} = (ForeignKey, Value) \quad (21)$$

3.2.1 Mapping Between Instances

Some transformations affect stored data. A relation between data from different *TableData* has to be known during execution of some transformations (e.g. *moveProperty*). The relation is defined as a mapping between *TableDatas*. The mapping is defined as follows:

$$mapping : TableData \rightarrow TableData \cup \emptyset \quad (22)$$

The mapping has a set of *TableData* in its range set, this allows to define one-to-many and many-to-many relations between data. The \emptyset represents a situation where there is no relation for a given element of the mapping's range. A special case of mapping is an empty mapping denoted as m_e , which is used when there are no *TableData* in the domain or the range i.e. the transformation takes part on the structural level only.

Each mapping has to fulfill constraints given by the structural definition of its range *TableData*. Concretely the uniqueness of column values:

$$\begin{aligned} \forall m \in Mapping; x_1, x_2 \in domain(m); p_1 \in pairs(m(x_1)), p_2 \in pairs(m(x_2)) : \\ x_1 \neq x_2 \wedge \exists c \in Column, UNIQUE \in constraints(c) \wedge \\ c \in pairs(columns(range(m))) \implies p_1 \neq p_2 \end{aligned} \quad (23)$$

if the principle of uniqueness is violated then usage of such mapping leads to an inconsistent database. Next constraint of mapping is the non emptiness of columns constrained with *NOTNULL* constraint:

$$\begin{aligned} \forall m \in Mapping; x \in domain(m) : \\ \exists c \in Column, NOTNULL \in constraints(c) \implies m(x) \neq \emptyset \end{aligned} \quad (24)$$

if this principle is violated then usage of such mapping leads to an inconsistent database.

There can occur data loss, when the mapping is a partial function. Usage of such mapping has to be reconsidered before its usage, because it can result in a semantically inconsistent state of the database.

The mapping can be implemented as a relation between two tables. This can be implemented e.g. as an equality of some columns or a reference from one table to the other. However, the real-life situation needs sometimes more difficult mappings, which can be implemented as a view or nested select commands.

4 Software Model

$$software(a, d, \rho) = \begin{cases} consistentSoftware(a, d, \rho) \text{ if } a \neq \perp \wedge d \neq \perp \\ \wedge \rho(a) = d \\ \perp \end{cases} \quad (25)$$

$a \in Application, d \in Database, \rho \in ORM$

5 Transformations

5.1 Application Manipulation

5.1.1 newApplication

Creates a new application, which does not contain classes.

$$newApplication \rightarrow Application \quad (26)$$

Semantics:

$$newApplication() = a \implies classes(a) = \emptyset \quad (27)$$

5.1.2 addClass

Inserts a class into the existing application.

$$addClass : Class \times Application \rightarrow Application \quad (28)$$

Semantics:

$$\begin{aligned} addClass(c, a) = a' &\implies classes(a') = classes(a) \cup c \\ \text{if } \forall c_a \in classes(a) : label(c_a) &\neq label(c) \end{aligned} \quad (29)$$

5.1.3 addProperty

Inserts a property into the given class in the application. Overriding of properties in inheritance hierarchy is prohibited.

$$addProperty : Class \times Property \times Application \rightarrow Application \quad (30)$$

Semantics:

$$\begin{aligned} addProperty(c, p, a) = a' &\implies properties(c') = properties(c) \cup \{p\} \wedge \\ classes(a') &= classes(a) \setminus c \cup \{c'\} \\ \text{if } \forall p_c \in properties(c) : label(p_c) &\neq label(p) \wedge c \in classes(a) \wedge !propertyInParent(p, c) \end{aligned} \quad (31)$$

5.1.4 addAssociation

Inserts an association between two classes existing in the application.

$$addAssociation : Class \times Association \times Application \rightarrow Application \quad (32)$$

Semantics:

$$\begin{aligned} addAssociation(c, as, a) = a' &\implies associations(c') = associations(c) \cup \{as\} \wedge \\ classes(a') &= classes(a) \setminus c \cup \{c'\} \\ \text{if } \forall as_c \in associations(c) : label(as_c) &\neq label(as) \wedge \\ reference(as) \in classes(a) \wedge c \in &classes(a) \wedge !associationInParent(a, c) \end{aligned} \quad (33)$$

5.1.5 renameProperty

Renames a property in the class.

$$\text{renameProperty} : \text{Class} \times \text{Property} \times \text{Label} \times \text{Application} \rightarrow \text{Application} \quad (34)$$

Semantics:

$$\begin{aligned} \text{renameProperty}(c, p, l, a) = a' &\implies \text{label}(p') = l \wedge \text{properties}(c') = \text{properties}(c) \setminus p \cup \{p'\} \wedge \\ &\text{classes}(a') = \text{classes}(a) \setminus c \cup \{c'\} \wedge \text{!propertyInParent}(p', c') \\ \text{if } \forall p_c \in \text{properties}(c) \wedge p_c \neq p : &\text{label}(p_c) \neq l \wedge c \in \text{classes}(a) \wedge p \in \text{properties}(c) \end{aligned} \quad (35)$$

5.1.6 renameAssociation

Renames an association in the class.

$$\text{renameAssociation} : \text{Class} \times \text{Association} \times \text{Label} \times \text{Application} \rightarrow \text{Application} \quad (36)$$

Semantics:

$$\begin{aligned} \text{renameAssociation}(c, as, l, a) = a' &\implies \text{label}(as') = l \wedge \text{land} \\ &\text{associations}(c') = \text{associations}(c) \setminus as \cup \{as'\} \wedge \text{classes}(a') = \text{classes}(a) \setminus c \cup \{c'\} \wedge \\ &\text{!associationInParent}(a', c') \\ \text{if } \forall as_c \in \text{associations}(c) \wedge as_c \neq as : &\text{label}(as_c) \neq l \wedge c \in \text{classes}(a) \wedge as \in \text{associations}(c) \end{aligned} \quad (37)$$

5.1.7 renameClass

Renames a class in the application.

$$\text{renameClass} : \text{Class} \times \text{Label} \times \text{Application} \rightarrow \text{Application} \quad (38)$$

Semantics:

$$\begin{aligned} \text{renameClass}(c, l, a) = a' &\implies \text{label}(c') = l \wedge \text{classes}(a') = \text{classes}(a) \setminus c \cup \{c'\} \\ \text{if } \forall c_a \in \text{classes}(a) \wedge c_a \neq c : &\text{label}(c_a) \neq l \wedge c \in \text{classes}(a) \end{aligned} \quad (39)$$

5.1.8 removeProperty

Removes property from the class and the application.

$$\text{removeProperty} : \text{Class} \times \text{Property} \times \text{Application} \rightarrow \text{Application} \quad (40)$$

Semantics:

$$\begin{aligned} \text{removeProperty}(c, p, a) = a' &\implies \text{properties}(c') = \text{properties}(c) \setminus p \wedge \\ &\text{classes}(a') = \text{classes}(a) \setminus c \cup \{c'\} \\ \text{if } p \in \text{properties}(c) \wedge c \in \text{classes}(a) & \end{aligned} \quad (41)$$

5.1.9 removeAssociation

Removes an association between two classes.

$$\text{removeAssociation} : \text{Class} \times \text{Association} \times \text{Application} \rightarrow \text{Application} \quad (42)$$

Semantics:

$$\begin{aligned} \text{removeAssociation}(c, as, a) = a' &\implies \text{associations}(c') = \text{associations}(c) \setminus as \wedge \\ &\text{classes}(a') = \text{classes}(a) \setminus c \cup \{c'\} \\ \text{if } c \in \text{classes}(a) \wedge as \in \text{associations}(c) & \end{aligned} \quad (43)$$

5.1.10 removeClass

Removes a class from the application.

$$removeClass : Class \times Application \rightarrow Application \quad (44)$$

Semantics:

$$\begin{aligned} removeClass(c, a) = a' &\implies classes(a') = classes(a) \setminus c \\ \text{if } c \in classes(a) \wedge !isReferenced(c, a) \wedge !isParent(c, a) \end{aligned} \quad (45)$$

5.1.11 addParent

Creates a child – parent relationship between two classes.

$$addParent : Class \times Inheritance \times Application \rightarrow Application \quad (46)$$

(47)

Semantics:

$$\begin{aligned} addParent(c, i, a) = a' &\implies parent(c') = i \\ classes(a') &= classes(a) \setminus c \cup \{c'\} \\ \text{if } c \in classes(a) \wedge class(i) \in classes(a) \wedge parent(c) &= OBJECT \wedge \\ !selfParentPossibility(c, class(i), a) \end{aligned} \quad (48)$$

5.1.12 removeParent

Destroys a child – parent relationship between two classes.

$$removeParent : Class \times Application \rightarrow Application \quad (49)$$

(50)

Semantics:

$$\begin{aligned} removeParent(c, a) = a' &\implies parent(c') = OBJECT \\ \text{if } c \in classes(a) \end{aligned} \quad (51)$$

5.1.13 pushDown

Moves the selected property from parent to all its child classes.

$$pushDown : Class \times Property \times Application \rightarrow Application \quad (52)$$

Semantics:

$$\begin{aligned} pushDown(c, p, a) = a' &\implies properties(c') = properties(c) \setminus p \wedge \\ \forall c_{ch} \in classes(a) \wedge class(parent(c_{ch})) = c : &properties(c'_{ch}) = properties(c_{ch}) \cup p \\ \wedge c'_{ch} \in classes(a') \wedge c_{ch} \notin classes(a') \wedge \\ \forall p' \in c_{ch} : label(p') \neq p \\ \text{if } c \in classes(a) \wedge \exists c_{ch} \in classes(a) : &class(parent(c_{ch})) = c \end{aligned} \quad (53)$$

5.1.14 pullUp

Moves the selected property from child to its parent.

$$pullUp : Class \times Property \times Application \rightarrow Application \quad (54)$$

Semantics:

$$\begin{aligned} pullUp(c, p, a) = a' &\implies \exists e = class(inheritance(c)) : properties(e') = properties(e) \cup p \wedge \\ &properties(c') = properties(c) \setminus p \\ &inheritance(c) \neq OBJECT \wedge p \in properties(c) \wedge \\ &\forall f \in children(e, a) \setminus c, q \in properties(f) : label(q) \neq label(p) \end{aligned} \quad (55)$$

5.2 Database Schema Manipulation

The section presents all transformations related with the database schema and data.

5.2.1 newDatabase

Creates a new empty database

$$newDatabase : \rightarrow Database \quad (56)$$

Semantics:

$$newDatabase() = d \implies tableSchema(d) = \emptyset \wedge tableData(d) = \emptyset \quad (57)$$

5.2.2 addTable

Adds a table schema into the database.

$$addTable : TableSchema \times Database \rightarrow Database \quad (58)$$

Semantics:

$$\begin{aligned} addTable(ts, d) = d' &\implies tableSchemas(d') = tableSchemas(d) \cup ts \\ &\text{if } \forall ts_d \in tableSchemas(d) : label(ts) \neq label(ts_d) \end{aligned} \quad (59)$$

5.2.3 addColumn

Adds a column into the table schema.

$$addColumn : TableSchema \times Column \times Database \rightarrow Database \quad (60)$$

Semantics:

$$\begin{aligned} addColumn(ts, col, d) = d' &\implies ts'.columns = ts.columns \cup col \wedge \\ &tableSchemas(d') = tableSchemas(d) \setminus ts \cup ts' \\ &\text{if } \forall col_{ts} \in ts.columns : label(col_{ts}) \neq label(col) \wedge ts \in tableSchemas(d) \end{aligned} \quad (61)$$

5.2.4 addForeignKey

Adds a foreign key into the table.

$$addForeignKey : TableSchema \times ForeignKey \times Mapping \times Database \rightarrow Database \quad (62)$$

Semantics:

$$\begin{aligned} addForeignKey(ts, fk, m, d) = d' &\implies ts'.foreignKeys = ts.foreignKeys \cup fk \wedge \\ tableSchemas(d') &= tableSchemas(d) \setminus ts \cup ts' \wedge \\ \forall p_i \in selectAll(ts, d), m_i \in m : p_i \in dom(m_i) &\implies insertValue(id(p_i), (fk, id(m(p_i))), d') \\ \text{if } \forall fk_{ts} \in ts.foreignKeys : label(fk) &\neq label(fk_{ts}) \wedge ts \in tableSchemas(d) \end{aligned} \quad (63)$$

5.2.5 alterColumnName

Changes the name of a column in the table schema.

$$alterColumnName : TableSchema \times Column \times Label \times Database \rightarrow Database \quad (64)$$

Semantics:

$$\begin{aligned} alterColumnName(ts, col, l, d) = d' &\implies label(col') = l \wedge \\ columns(ts') &= columns(ts) \setminus col \cup col' \wedge tableSchemas(d') = tableSchemas(d) \setminus ts \cup ts' \\ \text{if } \forall col_{ts} \in ts.columns : label(col_{ts}) &\neq l \wedge ts \in tableSchemas(d) \wedge col \in columns(ts) \end{aligned} \quad (65)$$

5.2.6 alterForeignKeyName

Changes the name of a foreign key in the table schema.

$$alterForeignKeyName : TableSchema \times ForeignKey \times Label \times Database \rightarrow Database \quad (66)$$

Semantics:

$$\begin{aligned} alterForeignKeyName(ts, fk, l, d) = d' &\implies \implies label(ts') = l \wedge \\ foreignKeys(ts') &= foreignKeys(ts) \setminus col \cup col' \wedge tableSchemas(d') = tableSchemas(d) \setminus ts \cup ts' \\ \text{if } \forall fk_{ts} \in ts.foreignKeys : label(fk_{ts}) &\neq l \wedge ts \in tableSchemas(d) \wedge \\ fk \in foreignKeys(ts) \end{aligned} \quad (67)$$

5.2.7 alterTableName

Changes the name of a table in the database schema.

$$alterTableName : TableSchema \times Label \times Database \rightarrow Database \quad (68)$$

Semantics:

$$\begin{aligned} alterTableName(ts, l, d) = d' &\implies label(ts') = l \wedge tableSchemas(d') = tableSchemas(d) \setminus ts \cup ts' \\ \text{if } \forall ts_d \in tableSchemas(d) : label(ts_d) &\neq l \wedge ts \in tableSchemas(d) \end{aligned} \quad (69)$$

5.2.8 dropColumn

Removes a column from the table schema.

$$\text{dropColumn} : \text{TableSchema} \times \text{Column} \times \text{Database} \rightarrow \text{Database} \quad (70)$$

Semantics:

$$\begin{aligned} \text{dropColumn}(ts, col, d) = d' \implies & \\ \left\{ \begin{array}{l} \text{columns}(ts') = \text{columns}(ts) \setminus col \wedge \\ \text{tableSchemas}(d') = \text{tableSchemas}(d) \setminus ts \cup ts' \\ \text{if } col \in \text{columns}(ts) \wedge |col| = 1 \\ \\ \text{dropColumn}(ts, \text{tail}(col), \text{dropColumn}(ts, \text{head}(col), d)) \\ \text{if } |col| > 1 \end{array} \right. & \quad (71) \end{aligned}$$

5.2.9 dropEmptyColumn

Removes a column from the table schema only if there are no data stored.

$$\text{dropEmptyColumn} : \text{TableSchema} \times \text{Column} \times \text{Database} \rightarrow \text{Database} \quad (72)$$

Semantics:

$$\begin{aligned} \text{dropEmptyColumn}(ts, col, d) = \text{dropColumn}(ts, col, d) \\ \text{if } \forall td \in \text{tableData}(d) : \text{table}(td) \neq ts \vee \forall cv \in \text{pairs}(ts) : \text{column}(cv) \neq col \end{aligned} \quad (73)$$

5.2.10 dropForeignKey

Removes a foreign key from the table schema.

$$\text{dropForeignKey} : \text{TableSchema} \times \text{ForeignKey} \times \text{Database} \rightarrow \text{Database} \quad (74)$$

Semantics:

$$\begin{aligned} \text{dropForeignKey}(ts, fk, d) = d' \implies & \text{foreignKeys}(ts') = \text{foreignKeys}(ts) \setminus fk \wedge \\ & \text{tableSchemas}(d') = \text{tableSchemas}(d) \setminus ts \cup ts' \\ & \text{if } fk \in \text{foreignKeys}(ts) \end{aligned} \quad (75)$$

5.2.11 dropEmptyForeignKey

Removes a foreign key from the table schema only if there are no data stored.

$$\text{dropEmptyForeignKey} : \text{TableSchema} \times \text{ForeignKey} \times \text{Database} \rightarrow \text{Database} \quad (76)$$

Semantics:

$$\begin{aligned} \text{dropEmptyForeignKey}(ts, fk, d) = \text{dropForeignKey}(ts, fk, d) \\ \text{if } \forall td \in \text{tableData}(d) : \text{table}(td) \neq ts \vee \forall fv \in \text{pairs}(ts) : \text{foreignKey}(fv) \neq fk \end{aligned} \quad (77)$$

5.2.12 dropTable

Removes a table schema from the database.

$$\text{dropTable} : \text{TableSchema} \times \text{Database} \rightarrow \text{Database} \quad (78)$$

Semantics:

$$\begin{aligned} \text{dropTable}(ts, d) = d' \implies & \text{tableSchemas}(d') = \text{tableSchemas}(d) \setminus ts \\ \text{if } & ts \in \text{tableSchemas}(d) \wedge ! \text{isReferenced}(ts, d) \end{aligned} \quad (79)$$

5.2.13 dropEmptyTable

Removes a table schema from the database only if there are no stored data.

$$\text{dropEmptyTable} : \text{TableSchema} \times \text{Database} \rightarrow \text{Database} \quad (80)$$

Semantics:

$$\begin{aligned} \text{dropEmptyTable}(ts, d) &= \text{dropTable}(ts, d) \\ \text{if } & \forall td \in \text{tableData}(d) : \text{table}(td) \neq ts \end{aligned} \quad (81)$$

5.2.14 copyColumn

The transformations copies structure of the column from one table schema to another. The data are copied according to the given mapping.

$$\begin{aligned} \text{copyColumn} : \text{TableSchema} \times \text{TableSchema} \times \text{Column} \times \text{Mapping} \times \text{Database} \\ \rightarrow \text{Database} \end{aligned} \quad (82)$$

Semantics:

$$\begin{aligned} \text{copyColumn}(ts_1, ts_2, col, m, d) = d' \implies \\ \left\{ \begin{array}{l} \text{columns}(ts_2') = \text{columns}(ts_2) \cup col \wedge \\ \forall p_i \in \text{selectAll}(ts_1, d) \wedge p_i \in \text{dom}(m_i), m_i \in m, q_i \in \text{ran}(m_i) : m(p_i) = q_i \implies \\ \text{insertValue}(q_i, (col, \text{valueOfColumn}(col, p_i)), d') \\ \text{if } ts_1 \neq ts_2 \wedge col \in \text{columns}(ts_1) \wedge |col| = 1 \\ \\ \text{copyColumn}(ts_1, ts_2, \text{tail}(col), m, \text{copyColumn}(ts_1, ts_2, \text{head}(col), m, d)) \\ \text{if } |col| > 1 \end{array} \right. \end{aligned} \quad (83)$$

5.2.15 copyTable

The transformations creates a copy of the given table. The new table (copy) has a new name defined by *label*. The data are copied as well.

$$\text{copyTable} : \text{TableSchema} \times \text{Label} \times \text{Database} \rightarrow \text{Database} \quad (84)$$

Semantics:

$$\begin{aligned} \text{copyTable}(ts, lm, d) = d' \implies & \exists ts_2 = (l, \text{primaryKey}(ts), \text{columns}(ts), \\ & \text{foreignKeys}(ts)) : ts_2 \in \text{tableSchemas}(d') \wedge ts_2 \notin \text{tableSchemas}(d) \\ & \forall q \in \text{ran}(m) : p \in \text{dom}(m) \wedge m(p) = q \wedge \\ & \text{insertData}(\text{next}(\text{sequence}(d)), \text{pairs}(p)) \\ \text{if } & \forall t \in \text{tableSchemas}(d) : \text{label}(t) \neq l \wedge ts \in \text{tableSchemas}(d) \end{aligned} \quad (85)$$

6 Evolution

6.1 Basic Transformations

6.1.1 newSoftware

Creates a new software with initialized application and database.

$$newSoftware : \rightarrow Software \quad (86)$$

$$newSoftware() = s \implies software(\Psi(newSoftware, \emptyset), \Phi(newSoftware, \emptyset)) \quad (87)$$

Interpretation:

$$\Psi(newSoftware, \emptyset) = newApplication() \quad (88)$$

$$\Phi(newSoftware, \emptyset) = newDatabase() \quad (89)$$

6.1.2 newClass

Inserts a class into the application and its image into the database.

$$newClass : Class \times Software \rightarrow Software \quad (90)$$

$$newClass(c, s) = s' \implies software(\Psi(newClass(c), application(s)), \Phi(newClass(c), database(s))) \quad (91)$$

Interpretation:

$$\Psi(newClass(c), a) = addClass(c, a) \quad (92)$$

$$\Phi(newClass(c), d) = addTable(ORM(c), d) \quad (93)$$

6.1.3 newProperty

Inserts a property into the given class in the application and its image into the database.

$$newProperty : Class \times Property \times Software \rightarrow Software \quad (94)$$

$$newProperty(c, p, s) = s' \implies software(\Psi(newProperty(c, p), application(s)), \Phi(newProperty(c, p), database(s))) \quad (95)$$

$$\text{if } (mandatory(d) = 1 \wedge selectAll(ORM(c), database(s)) = \emptyset) \vee mandatory(d) = 0 \quad (96)$$

Interpretation:

$$\Psi(newProperty(c, p), a) = addProperty(c, p, a) \quad (97)$$

$$\Phi(newProperty(c, p), d) = \begin{cases} addColumn(ORM(c), ORM(p), d) \\ \text{if } cardinality(p) = 1 \\ \\ addForeignKey(ORM(p), fk, m_e, \\ addTable(ORM(p), d)) \\ \text{if } cardinality(p) > 1 \\ \text{where } fk = (ORM(c) + "ref", ORM(C), \langle \rangle) \end{cases} \quad (98)$$

$$c \in Class, p \in Property, d \in Database$$

6.1.4 newAssociation

Inserts a new association between two existing classes into the application and its image into the database.

$$newAssociation : Class \times Association \times Mapping \times Software \rightarrow Software \quad (99)$$

$$newAssociation(c, as, s) = s' \implies software(\Psi(newAssociation(c, as), application(s)), \Phi(newAssociation(c, as), database(s))) \quad (100)$$

Interpretation:

$$\Psi(newAssociation(c, as), a) = addAssociation(c, as, a) \quad (101)$$

$$\begin{aligned} \Phi(newAssociation(c, as), d) &= \forall r \in selectAll(ORM(c), d') : r \in dom(m) \implies \\ &\quad insertData(m(r), d') \wedge d' = addTable(ORM(as), d) \\ \textbf{where } dom(m) &\in selectAll(ORM(c), d) \wedge ran(m) \in selectAll(ORM(reference(as)), d) \end{aligned} \quad (102)$$

6.1.5 renameProperty

The transformation changes the label of the given property.

$$\begin{aligned} renameProperty : Class \times Property \times Label \times Software &\rightarrow Software \\ renameProperty(c, p, l, s) &= software(\Psi(renameProperty(c, p, l), application(s)), \\ &\quad \Phi(renameProperty(c, p, l), database(s))) \end{aligned} \quad (103)$$

Interpretation:

$$\Psi(renameProperty(c, p, l), a) = renameProperty(c, p, l, a) \quad (104)$$

$$\Phi(renameProperty(c, p, l), d) = \begin{cases} alterColumnName(ORM(c), ORM(p), l, d) \\ \textbf{if } cardinality(p) = 1 \\ alterTable(ORM(p), l, d) \\ \textbf{if } cardinality(p) > 1 \end{cases} \quad (105)$$

6.1.6 renameAssociation

The transformation changes the label of the given association.

$$\begin{aligned} renameAssociation : Class \times Association \times Label \times Software &\rightarrow Software \\ renameAssociation(c, as, l, s) &= software(\Psi(renameAssociation(c, as, l), application(s)), \\ &\quad \Phi(renameAssociation(c, as, l), database(s))) \end{aligned} \quad (106)$$

Interpretation:

$$\Psi(renameAssociation(c, as, l), a) = renameAssociation(c, as, l, a) \quad (107)$$

$$\Phi(renameAssociation(c, as, l), d) = alterTableName(ORM(as), l, d) \quad (108)$$

6.1.7 renameClass

The transformation changes the label of the given class.

$$\begin{aligned}
& \text{renameClass} : \text{Class} \times \text{Label} \times \text{Software} \rightarrow \text{Software} \\
& \text{renameClass}(c, l, s) = \text{software}(\Psi(\text{renameClass}(c, l), \text{application}(s)), \\
& \quad \Phi(\text{renameClass}(c, l), \text{database}(s)))
\end{aligned} \tag{109}$$

Interpretation:

$$\Psi(\text{renameClass}(c, l), a) = \text{renameClass}(c, l, a) \tag{110}$$

$$\Phi(\text{renameClass}(c, l), d) = \text{alterTableName}(\text{ORM}(c), l, d) \tag{111}$$

6.1.8 removeProperty

The transformation removes the given property from the given class.

$$\begin{aligned}
& \text{removeProperty} : \text{Class} \times \text{Property} \times \text{Software} \rightarrow \text{Software} \\
& \text{removeProperty}(c, p, s) = \begin{cases} \text{software}(\Psi(\text{removeProperty}(c, p), \text{application}(s)), \\ \quad \Phi(\text{removeProperty}(c, p), \text{database}(s))) \\ \text{if } |p| = 1 \\ \text{removeProperty}(c, \text{tail}(p), \text{removeProperty}(c, \text{head}(p), s)) \\ \text{if } |p| > 1 \end{cases}
\end{aligned} \tag{112}$$

Interpretation:

$$\Psi(\text{removeProperty}(c, p), a) = \text{removeProperty}(c, p, a) \tag{113}$$

$$\begin{aligned}
& \Phi(\text{removeProperty}(c, p), d) = \begin{cases} \text{dropColumn}(\text{ORM}(c), \text{ORM}(p), d) \\ \text{if } \text{cardinality}(p) = 1 \\ \text{dropTable}(\text{ORM}(p), d) \\ \text{if } \text{cardinality}(p) > 1 \end{cases}
\end{aligned} \tag{114}$$

6.1.9 removeAssociation

The transformation removes the given association from the software.

$$\begin{aligned}
& \text{removeAssociation} : \text{Class} \times \text{Association} \times \text{Software} \rightarrow \text{Software} \\
& \text{removeAssociation}(c, as, s) = \begin{cases} \text{software}(\Psi(\text{removeAssociation}(c, as), \text{application}(s)), \\ \quad \Phi(\text{removeAssociation}(c, as), \text{database}(s))) \\ \text{if } |as| = 1 \\ \text{removeAssociation}(c, \text{tail}(as), \text{removeAssociation}(c, \text{head}(as), s)) \\ \text{if } |as| > 1 \end{cases}
\end{aligned} \tag{115}$$

Interpretation:

$$\Psi(\text{removeAssociation}(c, as), a) = \text{removeAssociation}(c, as, a) \quad (116)$$

$$\Phi(\text{removeAssociation}(c, as), d) = \text{dropTable}(\text{ORM}(as), d) \quad (117)$$

6.1.10 removeClass

The transformation removes the class from the software.

$$\begin{aligned} & \text{removeClass} : \text{Class} \times \text{Software} \rightarrow \text{Software} \\ & \text{removeClass}(c, s) = \begin{cases} \text{software}(\Psi(\text{removeClass}(c), \text{application}(s)), \\ \quad \Phi(\text{removeClass}(c), \text{database}(s))) \\ \quad \text{if } |c| = 1 \\ \text{removeClass}(\text{tail}(c), \text{removeClass}(\text{head}(c), s)) \\ \quad \text{if } |c| > 1 \end{cases} \end{aligned} \quad (118)$$

Interpretation:

$$\Psi(\text{removeClass}(c), a) = \text{removeClass}(c, a) \quad (119)$$

$$\Phi(\text{removeClass}(c), d) = \text{dropTable}(\text{ORM}(c), d) \quad (120)$$

6.1.11 copyProperty

The transformation copies the given property from one given class to another.

$$\text{copyProperty} : \text{Class} \times \text{Class} \times \text{Property} \times \text{Mapping} \times \text{Software} \rightarrow \text{Software} \quad (121)$$

$$\begin{aligned} & \text{copyProperty}(c_s, c_t, p, m, s) = \text{software}(\Psi(\text{copyProperty}(c_s, c_t, m, p), \text{application}(s)), \\ & \quad \Phi(\text{copyProperty}(c_s, c_t, m, p), \text{database}(s))) \end{aligned} \quad (122)$$

Interpretation:

$$\begin{aligned} & \Psi(\text{copyProperty}(c_s, c_t, p, m), a) = \text{newProperty}(c_t, p, a) \\ & \text{if } c_s \neq c_t \end{aligned} \quad (123)$$

$$\Phi(\text{copyProperty}(c_s, c_t, p, m), d) = \begin{cases} \text{copyColumn}(\text{ORM}(c_s), \text{ORM}(c_t), \text{ORM}(p), m, d) \\ \quad \text{if } \text{cardinality}(p) = 1 \\ \text{copyPropertyAsTable}(\text{ORM}(c_s), \text{ORM}(c_t), \text{ORM}(p), m, d) \\ \quad \text{if } \text{cardinality}(p) > 1 \end{cases} \quad (124)$$

6.1.12 moveProperty

The transformation moves the given property from one given class to another. A copy of given property is created and the original is then removed.

$$\text{moveProperty} : \text{Class} \times \text{Class} \times \text{Property} \times \text{Mapping} \times \text{Software} \rightarrow \text{Software} \quad (125)$$

$$\text{moveProperty}(c_s, c_t, p, m, s) = \begin{cases} \text{removeProperty}(c_s, p, \text{copyProperty}(c_s, c_t, p, m, s)) \\ \text{if } |p| = 1 \\ \text{moveProperty}(c_s, c_t, \text{tail}(p), m, \text{removeProperty}(c_s, \text{head}(p), \\ \text{copyProperty}(c_s, c_t, \text{head}(p), m, s))) \\ \text{if } |p| > 1 \end{cases} \quad (126)$$

6.1.13 inlineClass

$$\text{inlineClass} : \text{Class} \times \text{Class} \times \text{Mapping} \times \text{Software} \rightarrow \text{Software} \quad (127)$$

$$\text{inlineClass}(c_1, c_2, m, s) = \text{removeClass}(c_2, \text{moveProperties}(c_1, c_2, m, s)) \quad (128)$$

where $p = \text{properties}(c_2)$

$\text{moveProperties}(c_1, c_2, m, s) =$

$\forall p \in \text{properties}(c_2) : \text{moveProperty}(c_1, c_2, p, m, s)$

$$\text{if } !\text{isReferenced}(c_2, \text{application}(s)) \quad (129)$$

6.1.14 mergeClasses

The transformation merges two classes with the same structure into one class.

$$\text{mergeClasses} : \text{Class} \times \text{Class} \times \text{Label} \times \text{Software} \rightarrow \text{Software} \quad (130)$$

$$\text{mergeClasses}(c_1, c_2, l, s) = \text{software}(\Psi(\text{mergeClasses}(c_1, c_2, l, \text{application}(s)),$$

$$\Phi(\text{mergeClasses}(c_1, c_2, l, \text{database}(s)))$$

$$\text{if } \text{properties}(c_1) = \text{properties}(c_2) \wedge !\text{isReferenced}(c_1, d) \wedge !\text{isReferenced}(c_2, d) \quad (131)$$

$$\Psi(\text{mergeClasses}(c_1, c_2, l, a)) = \text{removeClass}(c_2, \text{renameClass}(c_1, l, a)) \quad (132)$$

$$\Phi(\text{mergeClasses}(c_1, c_2, l, d)) =$$

$$\text{alterTableName}(\text{ORM}(c_1), l, k(\text{ORM}(c_1), \text{selectAll}(\text{ORM}(c_2), d), d))$$

$$\text{if } !\text{isReferenced}(\text{ORM}(c_1), d) \wedge !\text{isReferenced}(\text{ORM}(c_2), d) \quad (133)$$

$$k : \text{TableSchema} \times \text{TableData} \times \text{Database} \rightarrow \text{Database} \quad (134)$$

$$k(ts, td, d) = \begin{cases} \text{insertData}(ts, \text{tableData}(ts, (pk, \text{next}(\text{sequence}(d))), \text{pairs}(td)), d) \\ \text{if } |td| = 1 \\ \text{where } pk = \text{primaryKey}(ts) \\ k(ts, \text{tail}(td), k(ts, \text{head}(td), d)) \text{if } |td| > 1 \end{cases} \quad (135)$$

6.1.15 splitClass

The transformation extract a subsequence of properties from the given class into a new class.

$$\text{splitClass} : \text{Class} \times \text{Label} \times \text{Property} \times \text{Software} \rightarrow \text{Software} \quad (136)$$

$$\begin{aligned} \text{splitClass}(c, l, p, s) &= \text{software}(\Psi(\text{splitClass}(c, l, p), \text{application}(s)), \\ &\quad \Phi(\text{splitClass}(c, l, p), \text{database}(s))) \end{aligned} \quad (137)$$

Interpretation:

$$\begin{aligned} \Psi(\text{splitClass}(c, l, p), a) &= \text{removeProperty}(c, p, \text{newProperty}(c_n, p, \text{addClass}(c_n, a))) \\ c_n &= \text{class}(l, \emptyset, \emptyset, \text{OBJECT}) \end{aligned} \quad (138)$$

$$\Phi(\text{splitClass}(c, l, p), d) = \begin{cases} \text{dropColumn}(\text{ORM}(c), \text{ORM}(p), \text{copyColumn}(\text{ORM}(c), \\ \quad \text{ORM}(\text{propToClass}(p)), m, \text{addTable}(\text{ORM}(\text{propToClass}(p, l)), d))) \\ \textbf{where } m = \forall r \in \text{selectAll}(\text{ORM}(c), d) : \\ \quad m(r) = \text{tableData}((\text{ORM}(\text{propToClass}(p, l)), \text{keyPair}(r), \\ \quad \text{pairOfColumn}(\text{ORM}(p), \text{pairs}(r)))) \\ \textbf{if } \text{cardinality}(p) = 1 \\ \\ \text{alterTableName}(\text{ORM}(p), l, \text{dropForeignKey}(\text{ORM}(p), \\ \quad \text{foreignKeys}(\text{ORM}(p), d)) \\ \textbf{if } \text{cardinality}(p) > 1 \end{cases} \quad (139)$$

6.1.16 extractPropertyAsObject

The transformation changes one property into an object with property containing original value.

$$\begin{aligned} \text{extractPropertyAsObject} : \text{Class} \times \text{Property} \times \text{Label} \times \text{Software} &\rightarrow \text{Software} \\ \text{extractPropertyAsObject}(c, p, l, s) &= \text{software}(\Psi(\text{extractPropertyAsObject}(c, p, l), \text{application}(s)), \\ &\quad \Phi(\text{extractPropertyAsObject}(c, p, l), \text{database}(s))) \end{aligned} \quad (140)$$

Interpretation:

$$\begin{aligned} \Phi(\text{extractPropertyAsObject}(c, p, l), a) &= \text{removeProperty}(c, p, \text{addAssociation}(c, \\ &\quad \text{association}(l, \text{propToClass}(p, l), 1, \text{cardinality}(p)), \\ &\quad \text{addClass}(\text{propToClass}(p, l), a)) \end{aligned} \quad (141)$$

$$\Psi(\text{extractPropertyAsObject}(c, p, l), d) = \quad (142)$$

$$\begin{cases} \text{dropColumn}(\text{ORM}(c), \text{ORM}(p), \text{addForeignKey}(ts_t, \text{foreignKey}(\text{label}(c), \text{ORM}(c), \text{NOTNULL}), \\ \quad \text{copyColumn}(\text{ORM}(c), ts_t, m, \text{addTable}(ts_t, d))) \\ \textbf{if } \text{cardinality}(p) = 1 \\ \textbf{where } ts_t = \text{ORM}(\text{propToClass}(p, l)) \wedge \\ \quad m \in \text{Mapping}, r \in \text{selectAll}(\text{ORM}(c), d) : m(r) = r \\ \\ \text{alterTableName}(\text{ORM}(p), l, d) \\ \textbf{if } \text{cardinality}(p) > 1 \end{cases} \quad (143)$$

6.1.17 inlineObjectAsProperty

The transformation *inlineObjectAsProperty* creates a property from the given class. The constraint is that the class to inline has only one property of primitive type and it is not referenced. The transformation *inlineObjectAsProperty* is the opposite do *extractPropertyAsObject*.

$$\begin{aligned}
& \text{inlineObjectAsProperty} : \text{Class} \times \text{Class} \times \text{Mapping} \times \text{Software} \rightarrow \text{Software} \\
& \text{inlineObjectAsProperty}(c_1, c_2, m, s) = \text{software}(\Psi(\text{inlineObjectAsProperty}(c_1, c_2, m), \\
& \quad \text{application}(s)), \Phi(\text{inlineObjectAsProperty}(c_1, c_2, m), \text{database}(s)))
\end{aligned} \tag{144}$$

Interpretation:

$$\Psi(\text{inlineObjectAsProperty}(c_1, c_2, m), a) = \text{removeClass}(c_2, \text{moveProperty}(c_1, c_2, p, a)) \tag{145}$$

$$\begin{aligned}
& \Phi(\text{inlineObjectAsProperty}(c_1, c_2, m), d) = \\
& \left\{ \begin{array}{l} \text{dropTable}(\text{ORM}(c_2), \text{copyColumn}(\text{ORM}(c_2), \text{ORM}(c_1), \text{ORM}(p), m, d) \\ \quad \text{if } \forall x, z \in \text{dom}(m), y \in \text{ran}(m) : m(x) = y \wedge m(z) = y \implies x = z \\ \\ d \\ \quad \text{if } \exists x \in \text{dom}(m), y, z \in \text{ran}(m) : m(x) = \{y, z\} \wedge y \neq z \end{array} \right. \\
& \text{if } p \in \text{properties}(c_2) \wedge |\text{properties}(c_2)| = 1 \wedge \text{isReferenced}(c, \text{application}(s))
\end{aligned} \tag{146}$$

6.1.18 addParent

Adds a parent - child relationship between the two given classes.

$$\begin{aligned}
& \text{addParent} : \text{Class} \times \text{Inheritance} \times \text{Mapping} \times \text{Software} \rightarrow \text{Software} \\
& \text{addParent}(c, ih, m, s) = \text{software}(\Psi(\text{addParent}(c, ih, m), \text{application}(s)), \\
& \quad \Phi(\text{addParent}(c, ih, m), \text{database}(s)))
\end{aligned} \tag{147}$$

Interpretation:

$$\begin{aligned}
& \Psi(\text{addParent}(c, ih, m, s) = \text{addParent}(c, ih, \text{application}(s)) \\
& \quad \text{if } ! \text{isReferenced}(c)
\end{aligned} \tag{148}$$

$$\begin{aligned}
& \Phi(\text{addParent}(c, ih, m, s) = \text{dropTable}(\text{ORM}(c), h(c, ih, p, m, s)) \\
& \quad \text{where } h : \text{TableSchema} \times \text{Inheritance} \times \text{Mapping} \times \text{Software} \rightarrow \text{Database} \\
& \quad h(c, ih, p, m, s) = \forall p \in \text{properties}(c) : \Phi(\text{moveProperty}(c, \text{class}(ih), p, m, s))
\end{aligned} \tag{149}$$

6.1.19 removeParent

Destroys the parent-child relationship between two classes. To simplify the model we assume the child class is a leaf of the inheritance hierarchy.

$$\begin{aligned}
& \text{removeParent} : \text{Class} \times \text{Software} \rightarrow \text{Software} \\
& \text{removeParent}(c, s) = \text{software}(\Psi(\text{removeParent}(c), \text{application}(s)), \\
& \quad \Phi(\text{removeParent}(c), \text{database}(s)) \\
& \quad \text{if } \forall e \in \text{classes}(\text{application}(a)) : \text{class}(\text{inheritance}(e)) \neq c
\end{aligned} \tag{150}$$

Interpretation:

$$\Psi(\text{removeParent}(c), a) = \text{removeParent}(c, a) \text{ if } \text{children}(c, a) = \emptyset \quad (151)$$

$$\begin{aligned} \Phi(\text{removeParent}(c), d) = & \text{dropColumn}(\text{ORM}(\text{class}(\text{inheritance}(c))), \text{ORM}(\text{properties}(c)) \setminus \\ & \text{ORM}(\text{properties}(\text{class}(\text{inheritance}(c))))) , \text{copyColumn}(\text{ORM}(\text{class}(\text{inheritance}(c))), \text{ORM}(c), \\ & \text{ORM}(\text{properties}(c)), m, \text{addTable}(\text{ORM}(c), d)) \\ \text{where } m = & \forall r \in \text{selectAll}(\text{ORM}(c), d) : \\ m(r) = & \text{tableData}((\text{ORM}(c), \text{keyPair}(r), \text{pairOfColumn}(\text{ORM}(\text{properties}(c)), \text{pairs}(r)))) \end{aligned} \quad (152)$$

6.1.20 pushDown

The transformation moves one property from the parent class to all child classes.

$$\begin{aligned} \text{pushDown} : & \text{Class} \times \text{Property} \times \text{Software} \rightarrow \text{Software} \\ \text{pushDown}(c, p, s) = & \text{software}(\Psi(\text{pushDown}(c, p), \text{application}(s)), \\ & \Phi(\text{pushDown}(c, p), \text{database}(s))) \end{aligned} \quad (153)$$

Interpretation:

$$\Psi(\text{pushDown}(c, p), a) = \text{pushDown}(c, p, a) \quad (154)$$

$$\Phi(\text{pushDown}(c, p), d) = d \quad (155)$$

6.1.21 pullUp

The transformation moves one property from the child class into the parent class.

$$\begin{aligned} \text{pullUp} : & \text{Class} \times \text{Property} \times \text{Software} \rightarrow \text{Software} \\ \text{pullUp}(c, p, s) = & \text{software}(\Psi(\text{pullUp}(c, p), \text{application}(s)), \Phi(\text{pullUp}(c, p), \text{database}(s))) \end{aligned} \quad (156)$$

Interpretation:

$$\Psi(\text{pullUp}(c, p), a) = \text{pullUp}(c, p, a) \quad (157)$$

$$\Phi(\text{pullUp}(c, p), d) = d \quad (158)$$

6.1.22 extractParent

The transformation creates a new class with the given property and creates the parent-child relationship between the new and the original class.

$$\begin{aligned} \text{extractParent} : & \text{Class} \times \text{Property} \times \text{InheritanceType} \times \text{Label} \times \\ & \text{ConsistentSoftware} \rightarrow \text{Software} \end{aligned} \quad (159)$$

$$\text{extractParent}(c, p, it, l, s) = \text{pullUp}(c, p, \text{addParent}(c, \text{inheritance}(e, it), m, \text{addClass}(e, s))) \quad (160)$$

where $e = \text{class}(l, \emptyset, \emptyset, \text{OBJECT})$

$$m \in \text{Mapping}, \text{dom}(m) = \text{selectAll}(\text{ORM}(c)) : \forall r \in m : m(r) = r$$

6.1.23 extractCommonParent

The transformation creates a new class with the given property, which exists in all given classes and creates the parent-child relationship between the new and all the original classes.

$$\begin{aligned} & \text{extractCommonParent} : \text{Class} \times \text{Property} \times \text{InheritanceType} \times \text{Label} \times \\ & \text{ConsistentSoftware} \rightarrow \text{Software} \end{aligned} \quad (161)$$

$$\text{extractCommonParent}(c, p, it, l, s) = \begin{cases} \text{extractParent}(c, p, it, l, s) \\ \text{if } |c| = 1 \\ h(\text{tail}(c), p, it, l, \text{extractParent}(\text{head}(c), p, it, l, s)) \\ \text{if } |c| > 1 \end{cases} \quad (162)$$

$$\text{if } \forall e \in c : p \in \text{properties}(e) \quad (163)$$

$$\begin{aligned} & h : \text{Class} \times \text{Property} \times \text{InheritanceType} \times \text{Label} \times \\ & \text{ConsistentSoftware} \rightarrow \text{Software} \end{aligned} \quad (164)$$

$$h(c, p, it, l, s) = \begin{cases} \text{moveProperty}(c, f, p, m, \text{addParent}(c, f, s)) \\ \text{if } |c| = 1 \\ h(\text{tail}(c), p, it, l, \text{moveProperty}(\text{head}(c), f, p, m, \text{addParent}(\text{head}(c), f, s))) \\ \text{if } |c| > 1 \end{cases}$$

$$\text{where } f = \text{class}(l, \emptyset, \emptyset, \text{OBJECT}) \wedge \quad (165)$$

$$m \in \text{Mapping}, \text{dom}(m) = \text{selectAll}(\text{ORM}(c)) : \forall r \in m : m(r) = r \quad (166)$$

7 Helpers

This section contains functions, which serves to obtain information about a model element.

7.1 Application Helpers

7.1.1 propertyInParent

Returns true if a property with the same name already exists in some class of the inheritance hierarchy.

$$\text{propertyInParent} : \text{Property} \times \text{Class} \rightarrow \text{Boolean} \quad (167)$$

$$\text{propertyInParent}(p, c) = \begin{cases} \text{false} \\ \text{if } \text{inheritance}(c) = \text{OBJECT} \\ \text{propertyInParent}(p, \text{class}(\text{inheritance}(c))) \\ \text{if } \forall p_c \in \text{properties}(c) : \text{label}(p_c) \neq \text{label}(p) \\ \text{true} \\ \text{if } \exists p_c \in \text{properties}(c) : \text{label}(p_c) = \text{label}(p) \end{cases} \quad (168)$$

7.1.2 associationInParent

Returns true if a association with the same name already exists in some class of the inheritance hierarchy.

$$associationInParent : Association \times Class \rightarrow Boolean \quad (169)$$

$$associationInParent(a, c) = \begin{cases} false & \text{if } inheritance(c) = OBJECT \\ associationInParent(a, class(inheritance(c))) & \text{if } \forall a_c \in associations(c) : label(a_c) \neq label(a) \\ true & \text{if } \exists a_c \in associations(c) : label(a_c) = label(a) \end{cases} \quad (170)$$

7.1.3 isReferenced

Returns true if the class is referenced in the given application.

$$isReferenced : Class \times Application \rightarrow Boolean \quad (171)$$

$$isReferenced(c, a) = \begin{cases} false & \text{if } \forall x \in classes(a), a_x \in associations(x) : reference(a_x) \neq c \\ true & \text{if } \exists x \in classes(a), a_x \in associations(a) : reference(a_x) = c \end{cases} \quad (172)$$

7.1.4 isParent

Returns true if the class is a parent of any other class in the model.

$$isParent : Class \times Application \rightarrow Boolean \quad (173)$$

$$isParent(c, a) = \begin{cases} false & \text{if } \forall x \in classes(a) : class(inheritance(x)) \neq c \\ true & \text{if } \exists x \in classes(a) : class(inheritance(x)) = c \end{cases} \quad (174)$$

7.1.5 selfParentPossibility

Returns true if the first given class is in the direct or indirect parent of the second class.

$$\text{selfParentPossibility} : \text{Class} \times \text{Class} \rightarrow \text{Boolean} \quad (175)$$

$$\text{selfParentPossibility}(c, c_{\text{parent}}) = \begin{cases} \text{false} & \text{if } \text{class}(\text{inheritance}(c_{\text{parent}})) = \text{OBJECT} \wedge c_{\text{parent}} \neq c \\ \text{selfParentPossibility}(c, \text{class}(\text{inheritance}(c_{\text{parent}}))) & \text{if } \text{class}(\text{inheritance}(c_{\text{parent}})) \neq c \wedge \\ & \text{class}(\text{inheritance}(c_{\text{parent}})) \neq \text{OBJECT} \\ \text{true} & \text{if } c_{\text{parent}} = c \end{cases} \quad (176)$$

7.2 Database Helpers

7.2.1 selectOne

Select one *TableData* which references given *TableSchema* and has the given primary key value.

$$\text{selectOne} : \text{TableSchema} \times \text{ID} \times \text{Database} \rightarrow \text{TableData} \quad (177)$$

Semantics:

$$\begin{aligned} \text{selectOne}(ts, id, d) = td &\implies td \in \text{tableData}(d) \wedge id(td) = id \wedge \text{table}(td) = ts \\ &\text{if } ts \in \text{tableSchemas}(d) \end{aligned} \quad (178)$$

7.2.2 selectAll

Select all *TableData* which references given *TableSchema*.

$$\text{selectAll} : \text{TableSchema} \times \text{Database} \rightarrow \text{TableData} \quad (179)$$

Semantics:

$$\begin{aligned} \text{selectAll}(ts, d) = td &\implies td \in \text{tableData}(d) \wedge \text{table}(td) = ts \\ &\text{if } ts \in \text{tableSchemas}(d) \end{aligned} \quad (180)$$

7.2.3 insertData

Inserts one *TableData* into the database.

$$\text{insertData} : \text{TableData} \times \text{Database} \rightarrow \text{Database} \quad (181)$$

Semantics:

$$\text{insertData}(td, d) = d' \implies \text{tableData}(d') = \text{tableData}(d) \cup td \quad (182)$$

7.2.4 insertValue

Inserts one *Pair* into the *TableData* in the database.

$$insertValue : TableData \times Pair \times Database \rightarrow Database \quad (183)$$

Semantics:

$$insertValue(td, p, d) = d' \implies tableData(d') = tableData(d) \setminus td \cup td' \wedge pairs(td') = pairs(td) \cup p \quad (184)$$

7.2.5 isReferenced

Returns true if there is a foreign key, which references given class, in the database.

$$notReferenced : TableSchema \times Database \rightarrow Boolean \quad (185)$$

$$notReferenced(t, d) = \begin{cases} false & \text{if } \forall t_s \in tableSchemas(d), fk \in foreignKeys(t_s) : reference(fk) \neq t \\ true & \text{if } \exists t_s \in tableSchemas(d), fk \in foreignKeys(t_s) : reference(fk) = t \end{cases} \quad (186)$$

7.2.6 reference

Returns a sequence of tableSchemas, which contains reference to a given tableSchema.

$$reference : TableData \times Database \rightarrow TableData \quad (187)$$

$$reference(td, d) = e \quad (188)$$

$$e \in tableSchemas(d) \wedge \exists f \in pairsForeignKeyValue(pairs(e)) \wedge reference(f) = td \quad (189)$$

7.2.7 valueOfColumn

Returns the value of given column from set of pairs.

$$valueOfColumn : Column \times Pair \rightarrow Value \quad (190)$$

$$valueOfColumn(c, p) = value(p) \wedge c = column(p) \quad (191)$$

7.2.8 pairOfColumn

Returns the *Pair* in *TableData* which reference given column.

$$pairOfColumn : Column \times TableData \rightarrow Pair \quad (192)$$

$$pairOfColumn(c, td) = p \implies p \in pairs(td) \wedge column(p) = c \quad (193)$$

7.2.9 pairsColumnValue

Returns all *Pairs* which references a column.

$$pairsColumnValue : Pair \rightarrow ColumnValue \quad (194)$$

$$pairsColumnValue(p) = q \implies q \in p \wedge column(q) \neq \emptyset \quad (195)$$

7.2.10 pairsForeignKeyValue

Returns all *Pairs* which references a foreign key.

$$pairsForeignKeyValue : Pair \rightarrow ForeignKeyValue \quad (196)$$

$$pairsForeignKeyValue(p) = q \implies q \in p \wedge reference(q) \neq \emptyset \quad (197)$$

7.2.11 copyPropertyAsTable

The transformation copies the property which is represented as a table in the database.

$$copyPropertyAsTable : TableSchema \times TableSchema \times Mapping \times Database \rightarrow Database \quad (198)$$

$$\begin{aligned} copyPropertyAsTable(ts_s, ts_t, m, d) = d' \implies \\ &fk_1 = foreignKeys(ts_s) \wedge fk_2 = foreignKey(ts_t, constraints(fk)) \wedge \\ &\exists ts_2 = (l, primaryKey(ts), columns(ts), fk_2) : \\ &ts_2 \in tableSchemas(d') \wedge ts_2 \notin tableSchemas(d) \\ &\forall q \in ran(m) : p \in dom(m) \wedge m(m) = q \wedge \\ &insertData(next(sequence(d)), pairsColumnValue(p) \cup (fk_2, id(q)), d') \\ &\mathbf{if} \forall t \in tableSchemas(d) : label(t) \neq l \wedge ts \in tableSchemas(d) \end{aligned} \quad (199)$$

7.2.12 propToClass

The transformation creates a class from the given property.

$$propToClass : Property \times Label \rightarrow Class \quad (200)$$

$$propToClass(p, l) = class(l, p, \emptyset, OBJECT) \quad (201)$$

7.2.13 children

Returns all children of the given class.

$$children : Class \times Application \rightarrow Class \quad (202)$$

$$children(c, a) = e \implies class(inheritance(e)) = c \wedge c \in classes(a) \wedge e \in classes(a) \quad (203)$$