List of Transformations for Code and Relational Database Evolution

Ondrej Macek and Karel Richta

November 26, 2013

Contents

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

$$\textbf{AppType} = APPSTRING \mid APPINTEGER \mid APPBOOLEAN \tag{1}$$
$$\textbf{InheritanceType} = SINGLETABLE \mid TABLE - PER - CLASS \mid \tag{2}$$
$$JOINED \tag{3}$$
$$\textbf{Inheritance} = (class, InheritanceType) \cup OBJECT \tag{4}$$
$$\textbf{Association} = (label, classRef, startCardinality, endCardinality) \tag{5}$$
$$\textbf{Property} = (label, AppType, DefaultValue,$$
$$Cardinality, Mandatory) \tag{6}$$
$$\textbf{Class} = (label, Property*, Association*, Inheritance) \tag{7}$$
$$\textbf{Application} = (Class*) \tag{8}$$
$$\tag{9}$$

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

$$\textbf{Database} = (TableSchema*, TableData*) \tag{10}$$
$$\textbf{TableSchema} = (label, primaryKey, Column*, ForeignKey*) \tag{11}$$
$$\textbf{Column} = (label, DbType, DefaultValue, Constraint*) \tag{12}$$
$$\textbf{ForeignKey} = (label, TableSchema, Constraint*) \tag{13}$$
$$\textbf{PrimaryKey} = (label, Sequence) \tag{14}$$
$$\textbf{DbType} = DBSTRING \mid DBINT \mid DBBOOLEAN \tag{15}$$
$$\textbf{Constraint} = NOTNULL \mid UNIQUE \tag{16}$$

3.2 Data

$$\textbf{TableData} = (Table, KeyPair, Pair*) \tag{17}$$
$$\textbf{KeyPair} = (PrimaryKey, Value) \tag{18}$$
$$\textbf{Pair} = ColumnValue \mid ForeignKeyValue \tag{19}$$
$$\textbf{ColumnValue} = (Column, Value) \tag{20}$$
$$\textbf{ForeignKeyValue} = (ForeignKey, Value) \tag{21}$$

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 $TableData$s. The mapping is defined as follows:

$$mapping : TableData \rightarrow TableData \cup \emptyset \tag{22}$$

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:

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

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

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

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

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

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

5 Transformations

5.1 Application Manipulation

5.1.1 newApplication

Creates a new application, which does not contain classes.

$$newApplication \rightarrow Application \tag{26}$$

Semantics:

$$newApplication() = a \implies classes(a) = \emptyset \tag{27}$$

5.1.2 addClass

Inserts a class into the existing application.

$$addClass : Class \times Application \rightarrow Application \tag{28}$$

Semantics:

$$addClass(c, a) = a' \implies classes(a') = classes(a) \cup c$$
$$\textbf{if } \forall c_a \in classes(a) : label(c_a) \neq label(c) \tag{29}$$

5.1.3 addProperty

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

$$addProperty : Class \times Property \times Application \rightarrow Application \tag{30}$$

Semantics:

$$addProperty(c, p, a) = a' \implies properties(c') = properties(c) \cup \{p\} \wedge$$
$$classes(a') = classes(a) \setminus c \cup \{c'\}$$
$$\textbf{if } \forall p_c \in properties(c) : label(p_c) \neq label(p) \wedge c \in classes(a) \wedge\, ! \, propertyInParent(p, c) \tag{31}$$

5.1.4 addAssociation

Inserts an association between two classes existing in the application.

$$addAssociation : Class \times Association \times Application \rightarrow Application \tag{32}$$

Semantics:

$$addAssociation(c, as, a) = a' \implies associations(c') = associations(c) \cup \{as\} \wedge$$
$$classes(a') = classes(a) \setminus c \cup \{c'\}$$
$$\textbf{if } \forall as_c \in associations(c) : label(as_c) \neq label(as) \wedge$$
$$reference(as) \in classes(a) \wedge c \in classes(a) \wedge\, ! \, associationInParent(a, c) \tag{33}$$

5.1.5 renameProperty

Renames a property in the class.

$$renameProperty : Class \times Property \times Label \times Application \rightarrow Application \tag{34}$$

Semantics:

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

5.1.6 renameAssociation

Renames an association in the class.

$$renameAssociation : Class \times Association \times Label \times Application \rightarrow Application \tag{36}$$

Semantics:

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

5.1.7 renameClass

Renames a class in the application.

$$renameClass : Class \times Label \times Application \rightarrow Application \tag{38}$$

Semantics:

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

5.1.8 removeProperty

Removes property from the class and the application.

$$removeProperty : Class \times Property \times Application \rightarrow Application \tag{40}$$

Semantics:

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

5.1.9 removeAssociation

Removes an association between two classes.

$$removeAssociation : Class \times Association \times Application \rightarrow Application \tag{42}$$

Semantics:

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

5.1.10 removeClass

Removes a class from the application.

$$removeClass : Class \times Application \rightarrow Application \tag{44}$$

Semantics:

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

5.1.11 addParent

Creates a child – parent relationship between two classes.

$$addParent : Class \times Inheritance \times Application \rightarrow Application \tag{46}$$
$$\tag{47}$$

Semantics:

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

5.1.12 removeParent

Destroys a child – parent relationship between two classes.

$$removeParent : Class \times Application \rightarrow Application \tag{49}$$
$$\tag{50}$$

Semantics:

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

5.1.13 pushDown

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

$$pushDown : Class \times Property \times Application \rightarrow Application \tag{52}$$

Semantics:

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

5.1.14 pullUp

Moves the selected property from child to its parent.

$$pullUp : Class \times Property \times Application \rightarrow Application \tag{54}$$

Semantics:

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

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

$$newDatabase :\rightarrow Database \tag{56}$$

Semantics:

$$newDatabase() = d \implies tableSchema(d) = \emptyset \wedge tableData(d) = \emptyset \tag{57}$$

5.2.2 addTable

Adds a table schema into the database.

$$addTable : TableSchema \times Database \rightarrow Database \tag{58}$$

Semantics:

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

5.2.3 addColumn

Adds a column into the table schema.

$$addColumn : TableSchema \times Column \times Database \rightarrow Database \tag{60}$$

Semantics:

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

5.2.4 addForeignKey

Adds a foreign key into the table.

$$addForeignKey : TableSchema \times ForeignKey \times Mapping \times Database \to Database \tag{62}$$

Semantics:

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

5.2.5 alterColumnName

Changes the name of a column in the table schema.

$$alterColumnName : TableSchema \times Column \times Label \times Database \to Database \tag{64}$$

Semantics:

$$\begin{aligned} &alterColumnName(ts, col, l, d) = d' \implies label(col') = l \wedge \\ &\quad columns(ts') = columns(t) \setminus col \cup col' \wedge tableSchemas(d') = tableSchemas(d) \setminus ts \cup ts' \\ &\quad \textbf{if } \forall col_{ts} \in ts.columns : label(col_{ts}) \neq l \wedge ts \in tableSchemas(d) \wedge col \in columns(ts) \end{aligned} \tag{65}$$

5.2.6 alterForeignKeyName

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

$$alterForeignKeyName : TableSchema \times ForeignKey \times Label \times Database \to Database \tag{66}$$

Semantics:

$$\begin{aligned} &alterForeignKeyName(ts, fk, l, d) = d' \implies \implies label(ts') = l \wedge \\ &\quad foreignKeys(ts') = foreignKeys(t) \setminus col \cup col' \wedge tableSchemas(d') = tableSchemas(d) \setminus ts \cup ts' \\ &\quad \textbf{if } \forall fk_{ts} \in ts.foreignKeys : label(fk_{ts}) \neq l \wedge ts \in tableSchemas(d) \wedge \\ &\qquad fk \in foreignKeys(ts) \end{aligned} \tag{67}$$

5.2.7 alterTableName

Changes the name of a table in the database schema.

$$alterTableName : TableSchema \times Label \times Database \to Database \tag{68}$$

Semantics:

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

5.2.8 dropColumn

Removes a column from the table schema.

$$dropColumn : TableSchema \times Column \times Database \to Database \tag{70}$$

Semantics:

$$dropColumn(ts, col, d) = d' \implies \tag{71}$$
$$\begin{cases} columns(ts') = columns(ts) \setminus col \wedge \\ tableSchemas(d') = tableSchemas(d) \setminus ts \cup ts' \\ \textbf{if } col \in columns(ts) \wedge |col| = 1 \\ \\ dropColumn(ts, tail(col), dropColumn(ts, head(col), d)) \\ \textbf{if } |col| > 1 \end{cases}$$

5.2.9 dropEmptyColumn

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

$$dropEmptyColumn : TableSchema \times Column \times Database \to Database \tag{72}$$

Semantics:

$$dropEmptyColumn(ts, col, d) = dropColumn(ts, col, d)$$
$$\textbf{if } \forall td \in tableData(d) : table(td) \neq ts \vee \forall cv \in pairs(ts) : column(cv) \neq col \tag{73}$$

5.2.10 dropForeignKey

Removes a foreign key from the table schema.

$$dropForeignKey : TableSchema \times ForeignKey \times Database \to Database \tag{74}$$

Semantics:

$$dropForeignKey(ts, fk, d) = d' \implies foreignKeys(ts') = foreignKeys(ts) \setminus fk \wedge$$
$$tableSchemas(d') = tableSchemas(d) \setminus ts \cup ts'$$
$$\textbf{if } fk \in foreignKeys(ts) \tag{75}$$

5.2.11 dropEmptyForeignKey

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

$$dropEmptyForeignKey : TableSchema \times ForeignKey \times Database \to Database \tag{76}$$

Semantics:

$$dropEmptyForeignKey(ts, fk, d) = dropForeignKey(ts, fk, d)$$
$$\textbf{if } \forall td \in tableData(d) : table(td) \neq ts \vee \forall fv \in pairs(ts) : foreignKey(fv) \neq fk \tag{77}$$

5.2.12 dropTable

Removes a table schema from the database.

$$dropTable : TableSchema \times Database \rightarrow Database \tag{78}$$

Semantics:

$$dropTable(ts, d) = d' \implies tableSchemas(d') = tableSchemas(d) \setminus ts$$
$$\textbf{if } ts \in tableSchemas(d) \wedge ! \, isReferenced(t, d) \tag{79}$$

5.2.13 dropEmptyTable

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

$$dropEmptyTable : TableSchema \times Database \rightarrow Database \tag{80}$$

Semantics:

$$dropEmptyTable(ts, d) = dropTable(ts, d)$$
$$\textbf{if } \forall td \in tableData(d) : table(td) \neq ts \tag{81}$$

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.

$$copyColumn : TableSchema \times TableSchema \times Column \times Mapping \times Database$$
$$\rightarrow Database \tag{82}$$

Semantics:

$$copyColumn(ts_1, ts_2, col, m, d) = d' \implies \tag{83}$$
$$\begin{cases} columns(ts_2') = columns(ts_2) \cup col \wedge \\ \forall p_i \in selectAll(ts_1, d) \wedge p_i \in dom(m_i), m_i \in m, q_i \in ran(m_i) : m(p_i) = q_i \implies \\ insertValue(q_i, (col, valueOfColumn(col, p_i)), d') \\ \textbf{if } ts_1 \neq ts_2 \wedge col \in columns(ts_1) \wedge |col| = 1 \\ \\ copyColumn(ts_1, ts_2, tail(col), m, copyColumn(ts_1, ts_2, head(col), m, d)) \\ \textbf{if } |col| > 1 \end{cases}$$

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.

$$copyTable : TableSchema \times Label \times Database \rightarrow Database \tag{84}$$

Semantics:

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

6 Evolution

6.1 Basic Transformations

6.1.1 newSoftware

Creates a new software with initialized application and database.

$$newSoftware :\rightarrow Software \tag{86}$$
$$newSoftware() = s \implies software(\Psi(newSoftware, \emptyset), \Phi(newSoftware, \emptyset)) \tag{87}$$

Interpretation:

$$\Psi(newSoftware, \emptyset) = newApplication() \tag{88}$$

$$\Phi(newSoftware, \emptyset) = newDatabase() \tag{89}$$

6.1.2 newClass

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

$$newClass : Class \times Software \rightarrow Software \tag{90}$$
$$newClass(c, s) = s' \implies software(\Psi(newClass(c), application(s)), \Phi(newClass(c), database(s)) \tag{91}$$

Interpretation:

$$\Psi(newClass(c), a) = addClass(c, a) \tag{92}$$

$$\Phi(newClass(c), d) = addTable(ORM(c), d) \tag{93}$$

6.1.3 newProperty

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

$$newProperty : Class \times Property \times Software \rightarrow Software \tag{94}$$
$$newProperty(c, p, s) = s' \implies software(\Psi(newProperty(c, p), application(s)), \Phi(newProperty(c, p), database(s)) \tag{95}$$

Interpretation:

$$\Psi(newProperty(c, p), a) = addProperty(c, p, a) \tag{96}$$

$$\Phi(newProperty(c, p), d) = \begin{cases} addColumn(ORM(c), ORM(p), d) \\ \textbf{if } cardinality(p) = 1 \\ \\ addTable(ORM(p), d) \\ \textbf{if } cardinality(p) > 1 \end{cases} \tag{97}$$

6.1.4 newAssociation

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

$$newAssociation : Class \times Association \times Mapping \times Software \rightarrow Software \tag{98}$$

$$newAssociation(c, as, s) = s' \implies software(\Psi(newAssociation(c, as), application(s)), \Phi(newAssociation(c, as), database(s)) \tag{99}$$

Interpretation:

$$\Psi(newAssociation(c, as), a) = addAssociation(c, as, a) \tag{100}$$

$$\Phi(newAssociation(c, as), d) = \forall r \in selectAll(ORM(c), d') : r \in dom(m) \implies insertData(m(r), d') \wedge d' = addTable(ORM(as), d)$$
$$\textbf{where } dom(m) \in selectAll(ORM(c), d) \wedge ran(m) \in selectAll(ORM(reference(as)), d) \tag{101}$$

6.1.5 renameProperty

The transformation changes the label of the given property.

$$renameProperty : Class \times Property \times Label \times Software \rightarrow Software$$
$$renameProperty(c, p, l, s) = software(\Psi(renameProperty(c, p, l), application(s)), \Phi(renameProperty(c, p, l), database(s)) \tag{102}$$

Interpretation:

$$\Psi(renameProperty(c, p, l), a) = renamePropery(c, p, l, a) \tag{103}$$

$$\Phi(renameProperty(c, p, l), d) = \begin{cases} alterColumnName(ORM(c), ORM(p), l, d) \\ \textbf{if } cardinality(p) = 1 \\ \\ alterTable(ORM(p), l, d) \\ \textbf{if } cardinality(p) > 1 \end{cases} \tag{104}$$

6.1.6 renameAssociation

The transformation changes the label of the given association.

$$renameAssociation : Class \times Association \times Label \times Software \rightarrow Software$$
$$renameAssociation(c, as, l, s) = software(\Psi(renameAssociation(c, as, l), application(s)), \Phi(renameAssociation(c, as, l), database(s)) \tag{105}$$

Interpretation:

$$\Psi(renameAssociation(c, as, l), a) = renameAssociation(c, as, l, a) \tag{106}$$

$$\Phi(renameAssociation(c, as, l), d) = alterTableName(ORM(as), l, d) \tag{107}$$

6.1.7 renameClass

The transformation changes the label of the given class.

$$
\begin{aligned}
&renameClass : Class \times Label \times Software \rightarrow Software \\
&renameClass(c, ls) = software(\Psi(renameClass(c, l), application(s)), \\
&\quad \Phi(renameClass(c, l), database(s))
\end{aligned} \tag{108}
$$

Interpretation:

$$\Psi(renameClass(c, l), a) = renameClass(c, l, a) \tag{109}$$

$$\Phi(renameClass(c, l), d) = alterTableName(ORM(c), l, d) \tag{110}$$

6.1.8 removeProperty

The transformation removes the given property from the given class.

$$
\begin{aligned}
&removeProperty : Class \times Property \times Software \rightarrow Software \\
&removeProperty(c, p, s) = \begin{cases} software(\Psi(removeProperty(c, p), application(s)), \\ \quad \Phi(removeProperty(c, p), database(s)) \\ \textbf{if } |p| = 1 \\ \\ removeProperty(c, tail(p), removeProperty(c, head(p), s)) \\ \textbf{if } |p| > 1 \end{cases}
\end{aligned} \tag{111}
$$

Interpretation:

$$\Psi(removeProperty(c, p), a) = removeProperty(c, p, a) \tag{112}$$

$$
\Phi(removeProperty(c, p), d) = \begin{cases} dropColumn(ORM(c), ORM(p), d)) \\ \textbf{if } cardinality(p) = 1 \\ \\ dropTable(ORM(p), d) \\ \textbf{if } cardinality(p) > 1 \end{cases} \tag{113}
$$

6.1.9 removeAssociation

The transformation removes the given association from the software.

$$
\begin{aligned}
&removeAssociation : Class \times Association \times Software \rightarrow Software \\
&removeAssociation(c, as, s) = \begin{cases} software(\Psi(removeAssociation(c, as), application(s)), \\ \quad \Phi(removeAssociation(c, as), database(s)) \\ \textbf{if } |as| = 1 \\ \\ removeAssociation(c, tail(as), removeAssociation(c, head(as), s)) \\ \textbf{if } |as| > 1 \end{cases}
\end{aligned} \tag{114}
$$

Interpretation:

$$\Psi(removeAssociation(c, as), a) = removeAssociation(c, as, a) \tag{115}$$

$$\Phi(removeAssociation(c, as), d) = dropTable(ORM(as), d) \tag{116}$$

6.1.10 removeClass

The transformation removes the class from the software.

$$removeClass : Class \times Software \to Software$$

$$removeClass(c, s) = \begin{cases} software(\Psi(removeClass(c), application(s)), \\ \quad \Phi(removeClass(c), database(s)) \\ \textbf{if } |c| = 1 \\ \\ removeClass(tail(c), removeClass(head(c), s)) \\ \textbf{if } |c| > 1 \end{cases} \tag{117}$$

Interpretation:

$$\Psi(removeClass(c), a) = removeClass(c, a) \tag{118}$$

$$\Phi(removeClass(c), d) = dropTable(ORM(c), d) \tag{119}$$

6.1.11 copyProperty

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

$$copyProperty : Class \times Class \times Property \times Mapping \times Software \to Software \tag{120}$$

$$copyProperty(c_s, c_t, p, m, s) = software(\Psi(copyProperty(c_s, c_t, m, p), application(s)), \\ \Phi(copyProperty(c_s, c_t, m, p), database(s))) \tag{121}$$

Interpretation:

$$\Psi(copyProperty(c_s, c_t, p, m), a) = removeProperty(c_s, p, newProperty(c_t, p, a)) \tag{122}$$

$$\Phi(copyProperty(c_s, c_t, p, m), d) = \begin{cases} copyColumn(ORM(c_s), ORM(c_t), ORM(p), m, d) \\ \textbf{if } cardinality(p) = 1 \\ \\ copyPropertyAsTable(ORM(c_s), ORM(c_t), ORM(p), m, d) \\ \textbf{if } cardinality(p) > 1 \end{cases} \tag{123}$$

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.

$$movePropery : Class \times Class \times Property \times Mapping \times Software \rightarrow Software \tag{124}$$

$$movePropery(c_s, c_t, p, m, s) = \begin{cases} removeProperty(c_s, p, copyProperty(c_s, c_t, p, m, s)) \\ \textbf{if} |p| = 1 \\ \\ movePropery(c_s, c_t, tail(p), m, removeProperty(c_s, head(p), \\ \quad copyProperty(c_s, c_t, head(p), m, s))) \\ \textbf{if} |p| > 1 \end{cases} \tag{125}$$

6.1.13 inlineClass

$$inlineClass : Class \times Class \times Mapping \times Software \rightarrow Software \tag{126}$$

$$inlineClass(c_1, c_2, m, s) = removeClass(c_2, movePropery(c_1, c_2, p, m, s)) \tag{127}$$

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

$$\textbf{if} !isReferenced(c_2, application(s)) \tag{128}$$

6.1.14 mergeClasses

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

$$mergeClasses : Class \times Class \times Label \times Software \rightarrow Software \tag{129}$$

$$mergeClasses(c_1, c_2, l, s) = software(\Psi(mergeClasses(c_1, c_2, l, application(s)), \\ \Phi(mergeClasses(c_1, c_2, l, database(s))$$

$$\textbf{if} properties(c_1) = properties(c_2) \wedge\ !isReferenced(c_1, d) \wedge\ !isReferenced(c_2, d) \tag{130}$$

$$\Psi(mergeClasses(c_1, c_2, l, a)) = removeClass(c_2, renameClass(c_1, l, a)) \tag{131}$$

$$\Phi(mergeClasses(c_1, c_2, l, d)) = \\ alterTableName(ORM(c_1), l, k(ORM(c_2), selectAll(ORM(c_1), d), d))$$

$$\textbf{if } !\, isReferenced(ORM(c_1), d) \wedge\ !\, isReferenced(ORM(c_2), d) \tag{132}$$

$$k : TableSchema \times TableData \times Database \rightarrow Database \tag{133}$$

$$k(ts, td, d) = \begin{cases} insertData(ts, tableData(ts, (pk, next(pk)), pairs(td)), d) \\ \textbf{if} |td| = 1 \\ \textbf{where } pk = primaryKey(ts) \\ \\ k(ts, tail(td), k(ts, head(td), d)) \textbf{if} |td| > 1 \end{cases} \tag{134}$$

6.1.15 splitClass

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

$$splitClass : Class \times Label \times Property \times Software \to Software \tag{135}$$

$$splitClass(c, l, p, s) = software(\Psi(splitClass(c, l, p), application(s)), \Phi(splitClass(c, l, p), database(s))) \tag{136}$$

Interpretation:

$$\Psi(splitClass(c, l, p), a) = removeProperty(c, p, newProperty(c_n, p, addClass(c_n, a))) \quad c_n = class(l, \emptyset, \emptyset, OBJECT) \tag{137}$$

$$\Phi(splitClass(c, l, p), d) = \begin{cases} dropColumn(ORM(c), ORM(p), copyColumn(ORM(c), ORM(propToClass(p)), m, addTable(ORM(propToClass(p, l)), d))) \\ \textbf{where } m = \forall r \in selectAll(ORM(c), d) : \\ \quad m(r) = tableData((ORM(propToClass(p, l)), keyPair(r), \\ \quad pairOfColumn(ORM(p), pairs(r)))) \\ \textbf{if } cardinality(p) = 1 \\ \\ alterTableName(ORM(p), l, dropForeignKey(ORM(p), \\ \quad foreignKeys(ORM(p), d)) \\ \textbf{if } cardinality(p) > 1 \end{cases} \tag{138}$$

6.1.16 extractPropertyAsObject

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

$$extractPropertyAsObject : Class \times Property \times Label \times Software \to Software$$
$$extractPropertyAsObject(c, p, l, s) = software(\Psi(extractPropertyAsObject(c, p, l), application(s)), \Phi(extractPropertyAsObject(c, p, l), database(s))) \tag{139}$$

Interpretation:

$$\Phi(extractPropertyAsObject(c, p, l), a) = addAssociation(c, association(l, propToClass(p, l), 1, cardinality(p)), addClass(propToClass(p, l), a)) \tag{140}$$

$$\Psi(extractPropertyAsObject(c, p, l), d) = \tag{141}$$

$$\begin{cases} addForeignKey(ts_t, foreignKey(label(c), ORM(c), NOTNULL), \\ \quad copyColumn(ORM(c), ts_t, m, addTable(ts_t, d))) \\ \textbf{if } cardinality(p) = 1 \\ \textbf{where } ts_t = ORM(propToClass(p, l)) \wedge \\ m \in Mapping, r \in selectAll(ORM(c), d) : m(r) = r \\ \\ alterTableName(ORM(p), l, d) \\ \textbf{if } cardinality(p) > 1 \end{cases} \tag{142}$$

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*.

$$
\begin{aligned}
&inilineObjectAsProperty : Class \times Class \times Mapping \times Software \rightarrow Software \\
&inilineObjectAsProperty(c_1, c_2, m, s) = software(\Psi(inilineObjectAsProperty(c_1, c_2, m), \\
&\quad application(s)), \Phi(inilineObjectAsProperty(c_1, c_2, m), database(s)))
\end{aligned} \tag{143}
$$

Interpretation:

$$
\Psi(inilineObjectAsProperty(c_1, c_2, m), a) = removeClass(c_2, moveProperty(c_1, c_2, p, a)) \tag{144}
$$

$$
\begin{aligned}
&\Phi(inilineObjectAsProperty(c_1, c_2, m), d) = \\
&\quad \begin{cases} dropTable(ORM(c_2), copyColumn(ORM(c_2), ORM(c_1), ORM(p), m, d) \\ \quad \textbf{if } \forall x, z \in dom(m), y \in ran(m) : m(x) = y \wedge m(z) = y \implies x = z \\ \\ d \\ \quad \textbf{if } \exists x \in dom(m), y, z \in ran(m) : m(x) = \{y, z\} \wedge y \neq z \end{cases} \\
&\quad \textbf{if } p \in properties(c_2) \wedge |properties(c_2)| = 1 \wedge !isReferenced(c, application(s))
\end{aligned} \tag{145}
$$

6.1.18 addParent

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

$$
\begin{aligned}
&addParent : Class \times Inheritance \times Mapping \times Software \rightarrow Software \\
&addParent(c, ih, m, s) = software(\Psi(addParent(c, ih, m), application(s)), \\
&\quad \Phi(addParent(c, ih, m), database(s)))
\end{aligned} \tag{146}
$$

Interpretation:

$$
\Psi(addParent(c, ih, m), a) = addParent(c, ih, a) \tag{147}
$$

$$
\Phi(addParent(c, ih, m), d) = \Phi(mergeClasses(c, class(ih), m), d) \tag{148}
$$

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.

$$
\begin{aligned}
&removeParent : Class \times Software \rightarrow Software \\
&removeParent(c, s) = software(\Psi(removeParent(c), application(s)), \\
&\quad \Phi(removeParent(c), database(s)) \\
&\textbf{if } \forall e \in classes(application(a)) : class(inheritance(e)) \neq c
\end{aligned} \tag{149}
$$

Interpretation:

$$
\Psi(removeParent(c), a) = removeParent(c, a) \textbf{if } children(c, a) = \emptyset \tag{150}
$$

$$\Phi(removeParent(c), d) = dropColumn(ORM(class(inheritance(c))), ORM(properties(c)) \setminus ORM(properties(class(inheritance(c)))), copyColumn(ORM(class(inheritance(c))), ORM(c), ORM(properties(c)), m, addTable(ORM(c), d))$$

where $m = \forall r \in selectAll(ORM(c), d) :$

$$m(r) = tableData((ORM(c), keyPair(r), pairOfColumn(ORM(properties(c)), pairs(r)))) \quad (151)$$

6.1.20 pushDown

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

$$pushDown : Class \times Property \times Software \to Software$$

$$pushDown(c, p, s) = software(\Psi(pushDown(c, p), application(s)), \Phi(pushDown(c, p), database(s)) \quad (152)$$

Interpretation:

$$\Psi(pushDown(c, p), a) = pushDown(c, p, a) \quad (153)$$

$$\Phi(pushDown(c, p), d) = d \quad (154)$$

6.1.21 pullUp

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

$$pullUp : Class \times Property \times Software \to Software$$

$$pullUp(c, p, s) = software(\Psi(pullUp(c, p), application(s)), \Phi(pullUp(c, p), database(s)) \quad (155)$$

Interpretation:

$$\Psi(pullUp(c, p), a) = pullUp(c, p, a) \quad (156)$$

$$\Phi(pullUp(c, p), d) = d \quad (157)$$

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.

$$extractParent : Class \times Property \times InheritanceType \times Label \times ConsistentSoftware \to Software \quad (158)$$

$$extractParent(c, p, it, l, s) = pullUp(c, p, addParent(c, inheritance(e, it), m, addClass(e, s))) \quad (159)$$

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

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

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.

$$extractCommonParent : Class \times Property \times InheritanceType \times Label \times ConsistentSoftware \rightarrow Software \tag{160}$$

$$extractCommonParent(c, p, it, l, s) = \begin{cases} extractParent(c, p, it, l, s) \\ \textbf{if } |c| = 1 \\ \\ h(tail(c), p, it, l, extractParent(head(c), p, it, l, s)) \\ \textbf{if } |c| > 1 \end{cases} \tag{161}$$

$$\textbf{if } \forall\, e \in c : p \in properties(e) \tag{162}$$

$$h : Class \times Property \times InheritanceType \times Label \times ConsistentSoftware \rightarrow Software \tag{163}$$

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

$$\textbf{where } f = class(l, \emptyset, \emptyset, OBJECT) \wedge \tag{164}$$

$$m \in Mapping, dom(m) = selectAll(ORM(c)) : \forall r \in m : m(r) = r \tag{165}$$

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.

$$propertyInParent : Property \times Class \rightarrow Boolean \tag{166}$$

$$propertyInParent(p, c) = \begin{cases} false \\ \textbf{if } inheritance(c) = OBJECT \\ \\ propertyInParent(p, class(inheritance(c))) \\ \textbf{if } \forall\, p_c \in properties(c) : label(p_c) \neq label(p) \\ \\ true \\ \textbf{if } \exists\, p_c \in properties(c) : label(p_c) = label(p) \end{cases} \tag{167}$$

7.1.2 associationInParent

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

$$associationInParent : Association \times Class \rightarrow Boolean \tag{168}$$

$$associationInParent(a, c) = \begin{cases} false \\ \textbf{if } inheritance(c) = OBJECT \\ \\ associationInParent(a, class(inheritance(c))) \\ \textbf{if } \forall\ a_c \in associations(c) : label(a_c) \neq label(a) \\ \\ true \\ \textbf{if } \exists\ a_c \in associations(c) : label(a_c) = label(a) \end{cases} \tag{169}$$

7.1.3 isReferenced

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

$$isReferenced : Class \times Application \rightarrow Boolean \tag{170}$$

$$isReferenced(c, a) = \begin{cases} false \\ \textbf{if } \forall\ x \in classes(a), a_x \in associations(x) : reference(a_x) \neq c \\ \\ true \\ \textbf{if } \exists\ x \in classes(a), a_x \in associations(a) : reference(a_x) = c \end{cases} \tag{171}$$

7.1.4 isParent

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

$$isParent : Class \times Application \rightarrow Boolean \tag{172}$$

$$isParent(c, a) = \begin{cases} false \\ \textbf{if } \forall\ x \in classes(a) : class(inheritance(x)) \neq c \\ \\ true \\ \textbf{if } \exists\ x \in classes(a) : class(inheritance(x)) = c \end{cases} \tag{173}$$

7.1.5 selfParentPossibility

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

$$selfParentPossibility : Class \times Class \to Boolean \tag{174}$$

$$selfParentPossibility(c, c_{parent}) = \begin{cases} false \\ \textbf{if } class(inheritance(c_{parent})) = OBJECT \wedge c_{parent} \neq c \\ \\ selfParentPossibility(c, class(inheritance(c_{parent})) \\ \textbf{if } class(inheritance(c_{parent})) \neq c \wedge \\ \quad class(inheritance(c_{parent})) \neq OBJECT \\ \\ true \\ \textbf{if } c_{parent} = c \end{cases} \tag{175}$$

7.2 Database Helpers

7.2.1 selectOne

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

$$selectOne : TableSchema \times ID \times Database \to TableData \tag{176}$$

Semantics:

$$selectOne(ts, id, d) = td \implies td \in tableData(d) \wedge id(td) = id \wedge table(td) = ts$$
$$\textbf{if } ts \in tableSchemas(d) \tag{177}$$

7.2.2 selectAll

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

$$selectAll : TableSchema \times Database \to TableData \tag{178}$$

Semantics:

$$selectAll(ts, d) = td \implies td \in tableData(d) \wedge table(td) = ts$$
$$\textbf{if } ts \in tableSchemas(d) \tag{179}$$

7.2.3 insertData

Inserts one *TableData* into the database.

$$insertData : TableData \times Database \to Database \tag{180}$$

Semantics:

$$insertData(td, d) = d' \implies tableData(d') = tableData(d) \cup td \tag{181}$$

7.2.4 insertValue

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

$$insertValue : TableData \times Pair \times Database \to Database \tag{182}$$

Semantics:

$$insertValue(td, p, d) = d' \implies tableData(d') = tableData(d) \setminus td \cup td' \wedge pairs(td') = pairs(td) \cup p \tag{183}$$

7.2.5 isReferenced

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

$$notReferenced : TableSchema \times Database \to Boolean \tag{184}$$

$$notReferenced(t, d) = \begin{cases} false \\ \textbf{if } \forall t_s \in tableSchemas(d), fk \in foreignKeys(t_s) : reference(fk) \neq t \\ \\ true \\ \textbf{if } \exists t_s \in tableSchemas(d), fk \in foreignKeys(t_s) : reference(fk) = t \end{cases} \tag{185}$$

7.2.6 reference

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

$$reference : TableData \times Database \to TableData \tag{186}$$

$$reference(td, d) = e \tag{187}$$

$$e \in tableSchemas(d) \wedge \exists f \in pairsForeignKeyValue(pairs(e)) \wedge reference(f) = td \tag{188}$$

7.2.7 valueOfColumn

Returns the value of given column from set of pairs.

$$valueOfColumn : Column \times Pair \to Value \tag{189}$$

$$valueOfColumn(c, p) = value(p) \wedge c = column(p) \tag{190}$$

7.2.8 pairOfColumn

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

$$pairOfColumn : Column \times TableData \to Pair \tag{191}$$

$$pairOfColumn(c, td) = p \implies p \in pairs(td) \wedge column(p) = c \tag{192}$$

7.2.9 pairsColumValue

Returns all *Pairs* which references a column.

$$pairsColumValue : Pair \to ColumnValue \tag{193}$$

$$pairsColumValue(p) = q \implies q \in p \wedge column(q) \neq \emptyset \tag{194}$$

7.2.10 pairsForeignKeyValue

Returns all *Pairs* which references a foreign key.

$$pairsForeignKeyValue : Pair \rightarrow ForeignKeyValue \tag{195}$$
$$pairsForeignKeyValue(p) = q \implies q \in p \wedge reference(q) \neq \emptyset \tag{196}$$

7.2.11 copyPropertyAsTable

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

$$copyPropertyAsTable : TableSchema \times TableSchema \times Mapping \times Database \rightarrow Database \tag{197}$$

$$\begin{aligned}
&copyPropertyAsTable(ts_s, ts_t, m, d) = d' \implies \\
&\quad fk_1 = foreignKeys(ts_s) \wedge fk_2 = foreignKey(ts_t, constraints(fk)) \wedge \\
&\quad \exists ts_2 = (l, primaryKey(ts), columns(ts), fk_2) : \\
&\quad ts_2 \in tableSchemas(d') \wedge ts_2 \notin tableSchemas(d) \\
&\quad \forall q \in ran(m) : p \in dom(m) \wedge m(m) = q \wedge \\
&\quad insertData(next(sequence(primaryKey(ts_2))), pairsColumValue(p) \cup (fk_2, id(q)), d') \\
&\quad \textbf{if } \forall t \in tableSchemas(d) : label(t) \neq l \wedge ts \in tableSchemas(d)
\end{aligned} \tag{198}$$

7.2.12 propToClass

The transformation creates a class from the given property.

$$propToClass : Property \times Label \rightarrow Class \tag{199}$$
$$propToClass(p, l) = class(l, p, \emptyset, OBJECT) \tag{200}$$

7.2.13 childern

Returns all children of the given class.

$$childern : Class \times Application \rightarrow Class \tag{201}$$
$$childern(c, a) = e \implies class(inheritance(e)) = c \wedge c \in classes(a) \wedge e \in classes(a) \tag{202}$$