Formal Transformation

The formal transformation is entirely based on the formal models of the core SBVR and the UML proposed in the previous section. It formally demonstrates how business vocabulary and rules of the core SBVR are mapped to the UML class diagram. The transformation is composed of a set of transformation rules. Each rule takes into account one or more elements of the SBVR model and translate them into elements of the UML model. When a rule is applied it changes the current UML model to a new UML model. Each rule specifies the pre-conditions (which have to be satisfied for the rule to be applied) and the post-conditions (which specified the result of the application; i.e. the changes made to the UML model). Although each transformation rule can be proved to be correct by itself (i.e. the application of a rule changes the current valid UML model into a new one, which is also valid), a logical proof is provided for each transformation rule.

The transformation rules are to be interpreted as follows. Given a valid SBVR model and a valid partial UML model, which contains a subset of the features expressed in the SBVR model. Then, if the pre-conditions of a rule are satisfied, then when the transformation applies the rule, it transforms the UML model to a new one, which remains to be valid. The proof of correctness is essentially to show that the new UML model is valid. This means we need to show that all the constraints expressed in the formal UML model must be satisfied.

<table>
<thead>
<tr>
<th>Group No.</th>
<th>Representation</th>
<th>Concept Type</th>
<th>General Concept</th>
<th>Mapped to</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Not equal to 'enumerated type' or 'data type'</td>
<td>Object Type</td>
<td>Unspecified</td>
<td>A class at top of hierarchy</td>
</tr>
<tr>
<td>2</td>
<td>Any</td>
<td>Object Type</td>
<td>A noun concept from sources of classes 1, 2, or 3 (super class)</td>
<td>A sub-class plus inheritance without generalization set</td>
</tr>
<tr>
<td>3</td>
<td>Any</td>
<td>Categorization type instance</td>
<td>A noun concept from source of classes 1, 2, or 3 (super class)</td>
<td>A sub-class plus inheritance with generalization set.</td>
</tr>
</tbody>
</table>

Table 1. The three groups of classes sources in the formal transformation.

Before presenting the transformation rules, there is a need to classify the sources of UML classes as the SBVR specification defines real world entities (classes), basic data types, and enumerations as noun concepts of object type. Additionally, noun concepts of object type are not the only sources of classes in the SBVR as categorization type instances can be also mapped to classes. These two points make the generation of classes more complex. However, to overwhelm this complexity, we categorize the noun concepts that can be mapped to classes into three groups of classes’ sources as presented in table 1.

To ensure that all UML classes are generated correctly and precisely within the transformation, there is a need to define four different sets to collect noun concepts of each group presented in table 4. These four sets can clearly distinguish between the three groups (table 1) and make the generation of classes more accurate.
**Definition 5** $NCGroup123$ is a set of noun concepts that belong to group 1, 2, and 3 (object types that are not instances of enumerated types or data types and categorization type instances).

$$NCGroup123 = \{ nc: \text{NounConcepts} | 
\begin{align*}
  nc.\text{conceptType} &= \text{'objectType'} \land (nc.\text{generalConcept} \neq \text{'enumerated type'} \land \\
  & \text{nc.\text{generalConcept}} \neq \text{'data type'}) \lor \\
  nc.\text{conceptType} \in \{ x: \text{NounConcepts} | x.\text{conceptType} = \text{'categorizationType'} \cdot \\
  & x.\text{representation} \}\}
\end{align*}$$

**Definition 6** $NCGroup1$ is a set of noun concepts that belong to group 1 (object types whose representations are not 'enumerated type' or 'data type' and general concepts are unspecified).

$$NCGroup1 = \{ nc: \text{NounConcepts} | 
\begin{align*}
  nc.\text{representation} &= \text{'enumerated Type'} \land nc.\text{representation} = \text{'data type'} \land \\
  nc.\text{conceptType} &= \text{'objectType'} \land nc.\text{generalConcept} = \bot 
\end{align*}$$

**Definition 7** $NCGroup2$ is a set of noun concepts that belong to group 2 (object types whose general concepts belong to $NCGroup123$).

$$NCGroup2 = \{ nc: \text{NounConcepts} | 
\begin{align*}
  nc.\text{conceptType} &= \text{'objectType'} \land nc.\text{generalConcept} \in \{ x: NCGroup123 \cdot \\
  & x.\text{representation} \}\}
\end{align*}$$

**Definition 8** $NCGroup3$ is a set of noun concepts that belong to group 3 (categorization type instances whose general concepts belong to $NCGroup123$).

$$NCGroup3 = \{ nc: \text{NounConcepts} | 
\begin{align*}
  nc.\text{conceptType} \in \{ x: \text{NounConcepts} | x.\text{conceptType} = \text{'categorizationType'} \cdot \\
  & x.\text{representation} \}\ \land nc.\text{generalConcept} \in \{ y: NCGroup123 \cdot y.\text{representation} \}\}
\end{align*}$$

**Rule 1: Generate Classes from Noun Concepts of Group 1, 2, and 3**

Rule 1 maps a noun concept of $NCGroup123$ to a new class and assigns the representation of the noun concept to the name of the generated class.

**Args:**

- $nc: \text{NounConcepts}$ // $nc$ is a noun concept

**Pre-conditions:**

- $nc \in NCGroup123 \land$ // $nc$ belongs to $NCGroup123$
- $nc.\text{representation} \notin \{ c: \text{Classes} \cdot c.\text{name} \}$ // there is no class with name $nc.\text{representation}$

**Post-conditions:**

- $\exists \text{newClass: Classes'} \cdot$ // generate a new class $\text{newClass}$
- $\text{newClass} = (nc.\text{representation}, \emptyset)$ // let the name of $\text{newClass}$ equals $nc.\text{representation}$
- $\text{Classes'} = \text{Classes} \cup \{ \text{newClass} \}$ // add the new class to $\text{Classes}$
Proof of Correctness:

With the exception of constraint 4.1.1, all other constraints are satisfied because they are constraints on sets of elements that are not changed by rule 1. As for constraint 4.1.1, consider two classes $c_1$ and $c_2$ in $Classes$’ such that $c_1 \neq c_2$. Then, there are two cases:

**Case 1:** $c_1, c_2 \in Classes$, and then $c_1.name \neq c_2.name$ due to the fact that constraint 4.1.1 is valid for the $classDiagram$.

**Case 2:** Either $c_1$ or $c_2$ does not belong to $Classes$. WLOG, let $c_1 \in Classes$ and $c_2 \notin Classes$, then,

$c_1.name \in \{x: Classes \cdot x.name\}$ and

$c_2.name = nc.representation \notin \{x: Classes \cdot x.name\}$

It follows that $c_1.name \neq c_2.name$

Therefore, constraint 4.1.1 remains valid after any execution of rule 1.

**Rule 2: Create Inheritances from Noun Concepts of Group 2 and Group 3**

This rule creates a new inheritance between two generated classes based on information provided by noun concepts belongs to group 2 or Group 3.

**Args:**

$nc: NounConcepts$  

// $nc$ is a noun concept

**Pre-conditions:**

$nc \in NCGroup2 \cup NCGroup3$  

// $nc$ belongs to NCGroup2 or NCGroup3

$nc.representation \in \{c_1: Classes \cdot c_1.name\}$  

// there is a sub-class $c_1$ whose name equals  

// $nc.representation$

$nc.generalConcept \in \{c_2: Classes \cdot c_2.name\}$  

// there is a parent-class $c_2$ whose name equals  

// $nc.generalConcept$

$\neg \exists inher: Inheritances \cdot inher.parentEnd = nc.generalConcept \land$  

inher.childEnd = $nc.representation$  

// there is no inheritance between the classes  

// whose names are equivalent to $nc.representation$  

// and $nc.generalConcept$

**Post-conditions:**

$\exists newInher: Inheritances \cdot$  

// generate a new inheritance $newInher$

$newInher = (getClass(nc.generalConcept), getClass(nc.representation))$  

// let the parent end of $newInher$ be the class whose  

// name equals $nc.generalConcept$ and the child end be  

// the class whose name equals $nc.representation$

$Inheritances = Inheritances \cup \{newInher\}$  

// add the new inheritance to $Inheritances$
**Function 4.** `getClass` gets a string as an input then retrieves the class whose name matches the input string in the set `Classes`.

\[
\text{getClass} : \text{Text} \mapsto \text{Classes} \\
\forall \text{name} : \text{Text}, \text{class} : \text{Classes} \\
\text{(name, class)} \in \text{getClass} \iff \text{class.name} = \text{name}
\]

**Proof of Correctness**

All but the two constraints considered below are obviously satisfied because they are constraints on sets that are not changed by rule 2. The first constraint that needs to be checked is Constraint 4.4.1, which ensures that the child and parent classes of the inheritance exist.

Now, the new inheritance is \((\text{getClass}(\text{nc.generalConcept}), \text{getClass}(\text{nc.representation}))\). Hence, the parent class is \(\text{getClass}(\text{nc.generalConcept})\), which is the class \(\text{parent}\) such that \(\text{parent.name} = \text{nc.generalConcept}\) (by the definition of \(\text{getClass}\) function). This class exists because of the pre-condition \(\text{nc.generalConcept} \in \{x : \text{Classes} \cdot x . \text{name}\}\). Similarly, the child class of the inheritance relationship exists. Thus, the first constraint (4.4.1) is satisfied.

The second constraint to be checked is constraint 4.4.2 that is a new set of inheritances cannot give rise to a circular relationship. The proof of correctness is facilitated by the fact that this is the only rule that adds inheritance relationships to the `classDiagram`. This means that a pair of classes \((\text{parent}, \text{child}) \in \text{Inheritances} \ (\text{of the classDiagram})\) if and only if there is a categorization fact type \(f\) such that \(f.\text{noun}_1 = \text{parent.name}\) and \(f.\text{noun}_2 = \text{child.name}\). And because the categorization fact types cannot gives rise to circular relationships among the noun concepts (constraint 4.4.2), it follows that the inheritance relationships of the `classDiagram` cannot give rises to circular relationships among the classes involved (constraint 4.4.2).

**Rule 3: Create Generalization Sets from Categorization Schemes**

Rule 3 generates a new generalization set from a categorization scheme. In this rule, if the type of the categorization scheme \((\text{cs.type})\) is ‘segmentation’ the constraint of the generated generalization set acquires the value ‘partial-disjoint’. On the other hand, if the type is ‘categorization scheme’ the constraint obtains the value ‘total-disjoint’. Below, rule 3 maps a categorization scheme whose type is ‘segmentation’.

**Args:**
\[\text{cs} : \text{CategorizationSchemes} \quad // \text{cs is a categorization scheme}\]

**Pre-conditions:**
\[
\neg \exists \text{genSet} : \text{GeneralizationSets} \cdot \text{genSet.name} = \text{cs.associatedCategorizationType} \\
\quad // \text{There is no generalization set whose name equals} \\
\quad // \text{cs.associatedCategorizationType} \\
\text{cs.type} = \text{‘segmentation’} \quad // \text{cs is a segmentation} \\
\text{cs.parentNounConcept} \in \{c_1 : \text{Classes} \cdot c_1.\text{name}\}\]
/ There is a parent class whose name equals
/ cs.parentNounConcept

cs.childrenNounConcepts ⊆ \{c_x: Classes • c_x.name\}

// There are sub-classes whose names in the set
// cs.parentNounConcept

∀ child: cs.childrenNounConcepts •
∃ inher: Inheritances • inher = (getClass (cs.parentNounConcept), getClass (child))

// there is an inheritance between each pair of parent-
// child noun concepts in cs

Post-conditions:

∃ newGenSet: GeneralizationSets’ •

// generate a new generalization set newGenSet

newGenSet = (cs.associatedCategorizationType, 'partial-disjoint' cs.parentNounConcept, cs.childrenNounConcepts)

// let the name of newGenSet equals
// cs.associatedCategorizationType the constraint of
// newGenSet equals to 'partial-disjoint', the parent of
// newGenSet equals to cs.parentNounConcept, and the
// children set of newGenSet equals cs.childrenNounConcepts

GeneralizationSets’ = GeneralizationSets ∪ \{newGenSet\}

// add the new generalization set to
// GeneralizationSets.

Proof of Correctness

Again, because this is the only rule to introduce generalization set, it can be shown (by induction) that there is a
one-to-one correspondence between the generalization sets of the class diagram and the categorization schemes of the
core SBVR model. Then, the constraints on the categorization schemes of the core SBVR model implies that the
corresponding constraints (conditions) on the UML model is also satisfied at every stage in the generation of the UML
class diagram.

Rule 4: Generate Associations from Associative Fact Types

In this rule, an associative fact type is transformed to a new association between two existing classes. The name of
the generated association is obtained from the verb used in the associative fact type.

Args:

f: FactTypes // f is a fact type

Pre-conditions:

f.conceptType = ‘associative’ // f is associative fact type
\[f.\text{noun}_1 \in \{nc_1: \text{NCGroup}123 \cdot nc_1.\text{representation}\}\]
\[f.\text{noun}_2 \in \{nc_2: \text{NCGroup}123 \cdot nc_2.\text{representation}\}\]
\[
\text{// f is between two noun concepts in Group 1, 2, or 3}
\]
\[f.\text{noun}_1 \in \{c_1: \text{Classes} \cdot c_1.\text{name}\}\]
\[f.\text{noun}_2 \in \{c_2: \text{Classes} \cdot c_2.\text{name}\}\]
\[
\text{// there are two classes } c_1 \text{ and } c_2. \text{ The name of } c_1 \text{ is}
\]
\[
\text{// equivalent to noun}_1 \text{ and the name of } c_2 \text{ is equivalent to noun}_2
\]

\[\neg \exists \text{ asso: Associations} \cdot \text{asso.name} = f.\text{verb} \land
\]
\[\text{asso.end}_1.\text{class.name} = f.\text{noun}_1 \land
\]
\[\text{asso.end}_2.\text{class.name} = f.\text{noun}_2\]
\[
\text{// there is no association with the name } f.\text{verb} \text{ between}
\]
\[
\text{// the classes whose names equals } f.\text{noun}_1 \text{ and } f.\text{noun}_2
\]

Post-conditions:
\[\exists \text{ newAsso: Associations'} \cdot \]
\[\text{newAsso} = (f.\text{verb}, (\text{getClass}(f.\text{noun}_1), f.\text{role}_1, \perp), \]
\[\text{(getClass}(f.\text{noun}_2), f.\text{role}_2, \perp))\]
\[
\text{// let the name of newAsso equals to } f.\text{verb}, \text{ the class}
\]
\[
\text{// whose name equals } f.\text{noun}_1 \text{ is the class of } \text{end}_1, \text{ the}
\]
\[
\text{// class whose name equals to } f.\text{noun}_2 \text{ is the class of } \text{end}_2
\]
\[
\text{// and the classRole of } \text{end}_1 \text{ and } \text{end}_2 \text{ equals to } f.\text{role}_1 \text{ and}
\]
\[
\text{// f.\text{role}_2 \text{ respectively}
\]
\[\text{Associations'} = \text{Associations} \cup \{\text{newAsso}\}\]
\[
\text{// add the new association to Associations}
\]

Proof of Correctness

After performing rule 4, all constraints are not changed excluding constraint 4.2.1. To satisfy this constraint, rule 4 applies a pre-condition (last pre-condition) that is used to check if there is an association with the same name between the same classes of the association being created or not. If there is an equivalent association, rule 4 does not generate any new association. If not, a new association is generated. Thus, constraint 4.2.1 remains valid after the application of rule 4.

**Rule 5: Generate Aggregations from Associative Fact Types**

A partitive fact type is mapped to an aggregation relationship between two particular classes from the classes set.

**Args:**
\[f: \text{FactTypes}\]
\[
\text{// f is a fact type}
\]

**Pre-conditions:**
\[f.\text{conceptType} = \text{'partitive'}\]
\[
\text{// f is partitive fact type}
\]
\[f.\text{originalNoun}_1 \in \{nc_1: \text{NCGroup}123 \cdot nc_1.\text{representation}\}\]
\[f.\text{originalNoun}_2 \in \{nc_2: \text{NCGroup}123 \cdot nc_2.\text{representation}\}\]
\[
\text{// f is between two noun concepts in Group 1, 2, or 3}
\]
Post-conditions:

\[\exists \text{newAggr}: \text{Aggregations}' \uplus \{\text{newAggr}\}\]

\[\text{newAggr} = (\text{f.originalVerb}, (\text{getClass(f.originalNoun}_1), \text{f.role}_1, \bot), (\text{getClass(f.originalNoun}_2), \text{f.role}_2, \bot))\]

\[\text{Aggregations}' = \text{Aggregations} \cup \{\text{newAggr}\}\]

\[
\text{Rule 6: Generate Enumerations with Literals}
\]

In this rule, an object type whose general concept equals ‘enumerated type’ is mapped to an enumeration and each of its related individual concepts are mapped to literals of the generated enumeration.

Args:

\[\text{nc: NounConcepts}\]

\[\text{individualSet: } \exists \text{ NounConcepts}\]

Pre-conditions:

\[\text{nc.conceptType} = \text{"objectType"}\]

\[\text{nc.generalConcept} = \text{"enumeratedType"}\]

\[\forall x: \text{individualSet \cdot x.conceptType} = \text{"individualConcept" \land}\]

\[\text{x.generalConcept} = \text{nc.representation}\]

\[\neg \exists x: \text{NounConcepts \setminus individualSet \cdot x.conceptType} = \text{"individualConcept" \land x.generalConcept} = \text{nc.representation}\]
// there is no instance of nc that is not included in
// individualSet
¬∃ enum: Enumerations • enum.name = nc.representation

// there is no enumeration whose name equals
// nc.representation

**Post-conditions:**

∃ newEnum: Enumerations' • // generate a new enumeration newEnum
newEnum = (nc.representation, {x: individualSet • x.representation})

// let the name of newEnum equals to nc.representation
// and the representation of the elements of
// individualSet to be the literals of newEnum

Enumerations' = Enumerations ∪ {newEnum}

// add the new enumeration to Enumerations

**Proof of Correctness**

In the core SBVR model, names of enumerated types are distinct. This is additionally ensured by the last pre-condition in the above transformation rule. In addition, the class diagram obtains all values from the core SBVR model. Strictly speaking, this desirable property is outside the framework for our proof of correctness but this fact together with our overall algorithm to apply the transformation rules would imply the correctness of the class diagram with respect to the given core SBVR model.

In the similar manner of generate enumerations (without literals) by rule 6, the transformation has another rule to generate basic data types. This rule receives its arguments as object types whose general concept is ‘data type’ then transforms them to basic data types of the UML class diagram.

**Rule 7: Generate non-Boolean Attributes from Is-property-of Fact Types**

Rule 6 transforms an is-property-of fact type to a new attribute of an existing class. The representation of the noun concept of role in the is-property-of fact type is assigned to the name of the generated attribute.

**Args:**

f: FactTypes // f is a fact type

**Pre-conditions:**

f.conceptType = 'is-property-of' // f is is-property-of fact type
f.noun1 ∈ {nc: NCGroup123 • nc.representation}

// noun1 belongs to Group123
f.role2 ∈ {nc: NounConcepts • nc.conceptType= 'role' • nc.representation}

// role2 is the representation of an existing role
∃ c: Classes • c.name = f.noun₁ ∧ f.role₂ ∉ {att: c.attributes • att.name}

// there is a class with the name noun₁ and the class
// does not have an attribute with name role₂

Post-conditions:

∃ class: Classes, updateClass: Classes' •
class.name = f.noun₁
updateClass = addAttribute(class, f.role₂,
{x: NounConcepts • x.representation = f.role₂ • x.generalConcept})
Classes' = Classes \ {class} U {updateClass}

// add a new attribute to class and let its name
// equals to role₂ and its data type equals to the
// general concept of role₂

Rule 6 uses the function addAttribute, which takes a class, an attribute’s name, and attribute’s data type and returns the updated class.

Function 5. addAttribute is defined as the following:

addAttribute: CLASS × Text → CLASS
∀ class: CLASS, attName, attType: Text •
AddAttribute (class, attName, attType) =
(class.name, class.attribute U {(attName, attType, ⊥)})

Proof of Correctness

After the execution of rule 7, all constraints are not affected except constraint 4.1.2 as this constraint is applied on attributes of classes.

Considering two attributes att₁ and att₂ for the class c such that att₁ ≠ att₂. Then, there are two cases:

Case 1: att₁, att₂ ∈ c.attributes, and then att₁.name ≠ att₂.name due to the fact that constraint 4.1.2 is valid for the classDiagram.

Case 2: Either att₁ or att₂ does not belong to c.attributes. WLOG, let att₁ ∈ c.attributes and c₂ ∉ c.attributes, then,
att₁.name ∈ {x: Classes • c.attributes • x.name} and
att₂.name = f.role₂ ∉ {x: Classes • c.attributes • x.name}

It follows that att₁.name ≠ att₂.name

As a confirmation of the fulfilment of constraint 4.1.2, rule 6 additionally applies the pre-condition (f.role₂ ∉ {att: c.attributes • att.name}), which verifies whether there is any attribute whose name equals the name of the attribute being created in the same class or not. Thus, constraint 4.1.2 is still valid after the execution of rule 4.
The above transformation rule demonstrates how an attribute is produced from an is-property-of fact type when the first noun concept of the fact type belongs to NCGroup123 and the second noun concept is role. Conversely, if the first noun concept is role and the second noun concept is a member of NCGroup123 set, this rule acquires the name of the relevant class from f.noun2 and assign the representation of f.role1 to the name of the generated attribute.

**Rule 8: Generate Boolean Attributes from Characteristic Fact Types**

This rule that generates Boolean attributes from characteristic fact types, which include only one noun concept and verb. This rule is quite analogous to rule 7 but it acquires the name of the generated attribute from the verb used in the characteristic fact types and by default it assigns ‘Boolean’ to the data type of the generated attribute.

**Args:**

\[f: \text{FactTypes} \quad // f \text{ is a fact type}\]

**Pre-conditions:**

\[f\text{.conceptType} = \text{`characteristic'} \quad // f \text{ is characteristic fact type}\]

\[f\text{.originalNoun1} \in \{\text{nc: NCGroup123} \bullet \text{nc.representation}\} \quad // \text{originalNoun1 belongs to Group123}\]

\[\exists c: \text{Classes} \bullet c\text{.name} = f\text{.originalNoun1} \land f\text{.originalVerb} \notin \{\text{att: c\text{.attributes} \bullet att.name}\} \quad // \text{there is a class with the name originalNoun1 and the class does not have an attribute whose name equals to originalVerb}\]

**Post-conditions:**

\[\exists \text{class: Classes, updateClass: Classes'} \bullet \text{class.name} = f\text{.originalNoun1}\]

\[\text{updateClass} = \text{addAttribute(class, f\text{.originalVerb, 'Boolean'})}\]

\[\text{Classes'} = \text{Classes} \setminus \{\text{class}\} \cup \{\text{updateClass}\} \quad // \text{add a new attribute to class and let its name equals to originalVerb and its data type equals to Boolean}\]

**Proof of Correctness**

Similar to rule 9, but requires that in the SBVR Model, there is no two characteristic fact types with the same noun and verb.

**Rule 9: Generate Cardinalities for Associations Based on Actual Verb**

This rule transforms the second quantifier in a business rule that includes an associative fact type with the actual verb form to cardinality for the second end of the relevant existing association.
Args:

- br: BusinessRules
  
  // br is a business rule

- f: FactTypes
  
  // f is a fact type

Pre-conditions:

- br.modal = 'necessary' ∨ 'impossible'
  
  // br has necessity or impossibility statement

- br.factType = f.representation
  
  // f is included in br

- f.conceptType = 'associative'
  
  // f is an associative fact type

- br.verbForm = 'actual'
  
  // br has a fact type in actual form

- ∃ asso: Associations • asso.name = f.verb ∧ asso.end1.class.name = f.noun1 ∧ asso.end2.class.name = f.noun2
  
  // there is an association with the name f.verb
  // between the two classes whose names equal
  // noun1 and noun2

Post-conditions:

- ∃ asso: Associations, updateAsso: Associations' •
  
  asso.name = f.verb ∧ asso.end1.class.name = f.noun1 ∧ asso.end2.class.name = f.noun2
  
  updateAsso = updateEnd2Card(asso, quantifierInterpreter(br.quantifier2))
  
  Associations' = Associations \ {asso} ∪ {updateAsso}
  
  // generate a new cardinality for end2 and let its value
  // equals the quantifier2 then add it to asso

The above rule employs the two functions, which are updateEnd2Card and quantifierInterpreter. The function updates the second end of an association while the function quantifierInterpreter interprets the quantification keywords based on the modal keyword included in the statements of business rules as demonstrated in table 2.

**Function 6. updateEnd2Card** is defined as follow:

- updateEnd2Card: ASSOCIATION × Rang

∀ asso: ASSOCIATION, card: Rang •

updateEnd2Card (asso, card) = (asso.name, asso.end1, (asso.end2.class, asso.end2.classRole, asso.end2.card))

**Function 7. quantifierInterpreter** is defined as the following:

- quantifierInterpreter: MODAL × QUANTIFIER → Card
Table 2. Interpretations of the quantification keywords.

**Proof of Correctness**

The only difference between the *Class Diagram* before and after the application of rule 9 is to update the cardinality at one end of an existing association. The expression for the updated second end of the association is \((a\text{name}, a\text{.end}_1, (a\text{.end}_2\text{.class}, a\text{.end}_2\text{.classRole}, a\text{.end}_2\text{.card})),\) which is valid because \(\text{card}\) is a valid range in the core SBVR model. \((asso\text{name}, asso\text{.end}_1, (asso\text{.end}_2\text{.class}, asso\text{.end}_2\text{.classRole}, \text{card})).\)

**Rule 10: Generate Cardinalities for Associations Based on Actual Verb**

Rule 10 maps the first quantifier in a business rule that includes an associative fact type with the reversed verb form to cardinality for the second end of the relevant existing association.

**Arg:**

\(br: BusinessRules \quad // br\) is a business rule

\(f: FactTypes \quad // f\) is a fact type

**Pre-conditions:**

\(br\text{.expression.model} = \text{‘necessary’} \lor \text{‘impossible’} \quad // br\) has necessity or impossibility statement

\(br\text{.expression.factType} = f\text{.representation} \quad // f\) is included in \(br\)

\(f\text{.conceptType} = \text{‘associative’} \quad // f\) is an associative fact type

\(br\text{.expression.verbForm} = \text{‘reversedFrom’} \quad // br\) has a fact type in reversed verb form

\(\exists asso: Associations \bullet asso\text{.name} = f\text{.originalVerb} \land asso\text{.end}_1\text{.class.name} = f\text{.originalNoun}_1 \land asso\text{.end}_2\text{.class.name} = f\text{.originalNoun}_2 \quad //\) there is an association with the name \(f\text{.originalVerb}\) between

\(\quad //\) the two classes whose names equal to \(f\text{.originalNoun}_1\) and

\(\quad // f\text{.originalNoun}_2\)

**Post-conditions:**

\(\exists asso: Associations, updateAsso: Associations' \bullet \quad \quad asso\text{.name} = f\text{.originalVerb} \land asso\text{.end}_1\text{.class.name} = f\text{.originalNoun}_1 \land \)
asso.end2.class.name = f.originalNoun2
updateAsso = updateEnd1Card(asso, quantifierInterpreter(br.expression.quantifier2))
Associations' = Associations \ {asso} ∪ {updateAsso}
// generate a new cardinality for end1 and let its value equals to
// the quantifier2 then add it to asso

Rule 10 employs the function updateEnd1Card.

Function 8. updateEnd1Card is defined as follow:

updateEnd1Card: ASSOCIATION × Rang
∀ asso: ASSOCIATION, card: Rang •
updateEnd1Card (asso, card) = (asso.name, (asso.end1.class, asso.end1.classRole, card), asso.end2)

Proof of Correctness

Similar argument as for Rule 9.

Rule 11: Generate Cardinalities for Aggregations Based on Actual Verb

This rule transforms the second quantifier in a business rule that includes a partitive fact type with the actual verb form to cardinality for the contained end of the relevant existing aggregation.

Args:
br: BusinessRules  // br is a business rule
f: FactTypes  // f is a fact type

Pre-conditions:
br.expression.model = ‘necessary’ ∨ ‘impossible’
// br has necessity or impossibility statement
br.expression.factType = f.representation
// f is included in br
f.conceptType = ‘partitive’  // f is a partitive fact type
br.expression.verbForm = ‘originalForm’
// br has a fact type in original verb form
∃ Aggr: Aggregations • aggr.name = f.originalVerb ∧
aggr.containerEnd.class.name = f.originalNoun1 ∧
Aggr.containedEnd.class.name = f.originalNoun2
// there is an aggregation with the name f.originalVerb between
// the two classes whose names equal to originalNoun1 and
// originalNoun2
Post-conditions:

\[ \exists \text{aggr}: \text{Aggregations}, \text{updateAggr}: \text{Aggregations}' \bullet \]
\[ \text{aggr.name} = f\text{.originalVerb} \land \text{aggr.continerEnd.class.name} = f\text{.originalNoun}_1 \land \]
\[ \text{Aggr.continedEnd.class.name} = f\text{.originalNoun}_2 \]
\[ \text{updateAggr} = \text{updateContainedEndCard}(\text{aggr}, \text{quantifierInterpreter} (\text{br.expression.quantifier}_2)) \]
\[ \text{Aggregations}' = \text{Aggregations} \backslash \{ \text{aggr} \} \cup \{ \text{updateAggr} \} \]

// generate a new cardinality for containedEnd and let its value
// equals the quantifier_2 then add it to aggr

Rule 11 employs the function updateContainedEndCard.

Function 9. updateContainedEndCard is defined as follow:

updateContainedEndCard: \text{AGGREGATION} \times \text{Rang}

\[ \forall \text{aggr}: \text{ASSOCIATION}, \text{card}: \text{Rang} \bullet \]
\[ \text{updateContainedEndCard} (\text{aggr}, \text{card}) = (\text{aggr.name}, \text{aggr.continerEnd}, \]
\[ (\text{aggr.containedEnd.class}, \text{aggr.containedEnd.classRole}, \text{card})) \]

Proof of Correctness

Similar to the proof of correctness for Rule 9.

Rule 12: Generate Cardinalities for Aggregations Based on Reversed Verb

Rule 12 transforms the first quantifier in a business rule that includes a partitive fact type with the reversed verb form to cardinality for the container end of the relevant existing aggregation.

Args:

\[ \text{br}: \text{BusinessRules} \]
\[ \text{f}: \text{FactTypes} \]

// br is a business rule
// fis a fact type

Pre-conditions:

\[ \text{br.expression.model} = 'necessary' \lor 'impossible' \]
// br has necessity or impossibility statement
\[ \text{br.expression.factType} = \text{f}\text{.representation} \]
// fis included in br
\[ \text{f.conceptType} = 'partitive' \]
// fis a partitive fact type
\[ \text{br.expression.verbForm} = 'reversedForm' \]
// br has a fact type in reversed verb form

\[ \exists \text{Aggr}: \text{Aggregations} \bullet \text{aggr.name} = f\text{.originalVerb} \land \]
\[ \text{aggr.continerEnd.class.name} = f\text{.originalNoun}_1 \land \]
\[ \text{Aggr.continedEnd.class.name} = f\text{.originalNoun}_2 \]
// there is an aggregation with the name \textit{f.originalVerb} between
// the two classes whose names equal to \textit{originalNoun}_1 and
// \textit{originalNoun}_2

**Post-conditions:**

\[ \exists \text{aggr: Aggregations, updateAggr: Aggregations}' \bullet \]
\[ \text{aggr.name} = f.\text{originalVerb} \land \text{aggr.continerEnd.class.name} = f.\text{originalNoun}_1 \land \]
\[ \text{Aggr.continedEnd.class.name} = f.\text{originalNoun}_2 \]
\[ \text{updateAggr} = \text{updateContainerEndCard}(\text{aggr, quantifierInterpreter (br.expression.quantifier}_2)) \]
\[ \text{Aggregations}' = \text{Aggregations} \setminus \{\text{aggr}\} \cup \{\text{updateAggr}\} \]

// generate a new cardinality for \textit{containerEnd} and let its value
// equals the \textit{quantifier}_2 then add it to \textit{aggr}

Rule 12 employs the function \textit{updateContainerEndCard}.

**Function 9.** \textit{updateContainerEndCard} is defined as follow:

\[ \text{updateContainerEndCard: AGGREGATION} \times \text{Rang} \]
\[ \forall \text{aggr: AGGREGATION , card: Rang} \bullet \]
\[ \text{updateContainerEndCard (aggr, card)} = (\text{aggr.name,}
\]
\[ (\text{aggr.containerEnd.class, aggr.containerEnd.classRole, card), aggr.containedEnd}) \]

**Proof of Correctness**

Similar argument as for Rule 9.

**Rule 13: Generate Cardinalities for Attributes**

This rule maps the quantifiers that are included in business rules built on is-property-of fact types and associated with the noun concepts of \textit{role} to cardinalities of the relevant attributes. This means if the first noun concept of the fact type is \textit{role}, the first quantifier is transformed to a cardinality of the related attribute. However, if the second noun concept is \textit{role}, the second quantifier is mapped to a cardinality of the relevant attribute. Rule 13 below shows how to generate a cardinality from the second quantifier in business rule that includes an is-property-of fact type.

**Args:**

\[ br: \text{BusinessRules} \quad \text{// br is a business rule} \]
\[ f: \text{FactTypes} \quad \text{// f is a fact type} \]

**Pre-conditions:**

\[ br.modal = \text{‘necessary’} \lor \text{‘impossible’} \quad \text{// br has necessity or impossibility statement} \]
\[ br.factType = f.\text{representation} \quad \text{// f is included in br} \]
\[ f.\text{conceptType} = \text{‘is-property-of’} \quad \text{// f is an is-property-of fact type} \]
\[ \exists \text{class: Classes} \bullet \text{class.name} = f.\text{noun}_1 \land f.\text{noun}_2 \in \{\text{att: class.attributes} \bullet \text{att.name}\} \]
// there is a class with the name noun₁ and the class
// has an attribute with name noun₂

Post-conditions:

∃ class: Classes, att: class.attributes, updateClass: Classes' •
class.name = f.noun₁ ∧ att = f.noun₂
updateClass = updateAttributeCard(class, att, quantifierInterpreter (br.quantifier₂))
Classes' = Classes \ {class} \ {updateClass} // generate a new cardinality for att and let its
// value equals quantifier₂

Rule 8 uses the function updateAttributeCard that takes a class, an attribute, and attribute’s cardinality and returns
the updated attribute.

Function 10. updateAttributeCard is defined as the following:

updateAttributeCard: CLASS × ATTRIBUTE × Rang → CLASS
∀ class: CLASS, att: class.attributes, card: Rang •
updateAttributeCard (class, att, card) = (class.name, class.attributes, {(att.name, att.dataType, card)})