Formal Models of Core SBVR and UML

To establish a rigorous basis for the transformation, there is a need to formalize the core SBVR subset and the UML class diagram precisely. The formal model of the core SBVR subset is the input of the transformation while the formal model of the UML class diagram is the output. These models accurately comprise all requisite elements and properties for the transformation. Moreover, they are restricted by sets of constraints that are used to validate their elements and properties. Any valid model of these two models must have the structure specified as well as fulfill all related constraints. The two formal models, as a result, play a significant role to obtain properly defined and validated inputs and outputs for the transformation. This section presents these formal models associated with their required constraints.

1 Formal Model of Core SBVR

The formal model of the core SBVR subset is divided into three main parts, which are SBVR vocabulary, SBVR keywords, and SBVR business rules.

1.1 SBVR Vocabulary

OMG [9] uses noun concepts and verbs to build fact types but it does not conceptually recognize verbs as a part of the SBVR vocabulary. However, to use the verbs in the practical manner correctly, each verb has to be defined before it is used in any fact type. To overcome this, we add the verbs to the SBVR vocabulary. Consequently, the formal definition of the SBVR vocabulary has four main elements instead of three, which are noun concepts, verbs, fact types, and categorization schemes.

Definition 1. SBVR vocabulary is a tuple \(SBVRVocab = (\text{NounConcepts}, \text{Verbs}, \text{FactTypes}, \text{CategorizationSchemes})\).

Definition 1.1. \(\text{NounConcepts}\) is a set of noun concepts. A noun concept \(nc \in \text{NounConcepts}\) is a tuple \(nc = (\text{representation}, \text{conceptType}, \text{generalConcept})\), where:

a) \(\text{representation}\) is the representation of the noun concept.

b) \(\text{conceptType}\) is the concept type. It can be one of \(\text{objectType, role, individualConcept, categorizationType, or categorizationTypeInstance}\).

c) \(\text{generalConcept}\) is the general concept and it must be an existing \(\text{objectType or categorizationTypeInstance}\).

Constraint 1.1.1. The representations of noun concepts must be unique. That is,

\[\forall nc_1, nc_2: \text{NounConcepts} \bullet nc_1 \neq nc_2 \Rightarrow nc_1.\text{representation} \neq nc_2.\text{representation}\]

// \(nc_1, nc_2\) are unique noun concepts

Constraint 1.1.2. The representations and concept types of noun concepts have to be specified. That is,

\[\forall nc: \text{NounConcept} \bullet nc.\text{representation} \neq \perp \land nc.\text{conceptType} \neq \perp\]

// the representation and conceptType of \(nc\) are not unspecified
**Constraint 1.1.3.** The concept types of `categorizationTypeInstances` have to be an existing `categorizationTypes`. That is,

\[
\forall \text{nc}_1, \text{nc}_2: \text{NounConcepts} \bullet \text{nc}_1.\text{conceptType} = \text{nc}_2.\text{representation} \Rightarrow \\
\text{nc}_2.\text{conceptType} = \text{‘categorizationType’} \\
// the conceptType of nc1 is an existing categorizationType nc2
\]

**Constraint 1.1.4.** The general concepts of `roles`, `individualConcepts`, and `categorizationTypeInstances` have to be specified. That is,

\[
\forall \text{nc}: \text{NounConcepts} \bullet \text{nc}.\text{conceptType} = \text{‘role’} \lor \text{nc}.\text{conceptType} = \text{‘individualConcept’} \lor \\
\text{nc}.\text{conceptType} = \{y: \text{NounConcepts} \bullet y.\text{conceptType} = \text{‘categorizationType’} \bullet y.\text{representation}\} \\
\rightarrow \\
\text{nc}.\text{generalConcept} \neq \bot \\
// the generalConcept of the role, individualConcept, or \\
// categorizationTypeInstance nc is specified
\]

**Constraint 1.1.5.** The general concepts of noun concepts can be only existing `objectTypes` or `categorizationTypeInstances`. That is,

\[
\forall \text{nc}: \text{NounConcepts} \bullet \text{nc}.\text{generalConcept} = \{x: \text{NounConcepts} \bullet x.\text{conceptType} = \text{‘objectType’} \lor \\
x.\text{conceptType} = \{y: \text{NounConcepts} \bullet y.\text{conceptType} = \text{‘categorizationType’} \bullet y.\text{representation}\} \bullet \\
x.\text{representation}\} \\
// the generalConcept of nc is an existing objectType or \\
// categorizationTypeInstance
\]

**Constraint 1.1.6.** Although the general concept of noun concepts can be existing `objectType` or `categorizationTypeInstances`, the general concepts of `categorizationTypes` must be unspecified. That is,

\[
\forall \text{nc}: \text{NounConcepts} \bullet \text{nc}.\text{conceptType} = \text{‘categorizationType’} \Rightarrow \text{nc}.\text{generalConcept} = \bot \\
// the generalConcept of the categorization type nc is not specified
\]

For example, the noun concept `manager` presented in table 3 can be specified as:

a) `nc.\text{representation} = \text{‘manager’}`

b) `nc.\text{conceptType} = \text{role}`

c) `nc.\text{generalConcept} = \text{‘employee’}`

**Definition 1.2.** `Verbs` is a set of verbs. A verb \(v \in \text{Verbs}\) is an element of the set `Verbs`.

**Constraint 1.2.1.** The verbs must be unique. That is,

\[
\forall v_1, v_2: \text{Verbs}\bullet v_1 \neq v_2 \\
// v_1, v_2 are unique verbs
\]

**Constraint 1.2.2.** The verbs have to be specified. That is,

\[
\forall v: \text{Verbs} \Rightarrow v \neq \bot \\
// the verb v is not unspecified
\]
Definition 1.3. FactTypes is a set of fact types. A fact type \( f \in \text{FactTypes} \) is a tuple \( f = (\text{representation}, \text{conceptType}, \text{noun}_1, \text{noun}_2, \text{role}_1, \text{role}_2, \text{verb}, \text{reversedVerb}) \), where:

\begin{enumerate}
\item representation is the primary representation of the fact type.
\item conceptType is the concept type. It can be one of associative, partitive, categorization, is-property-of, characteristic.
\item noun\(_1\) is the first noun concept (objectType or categorizationTypeInstances) that is included in the representation of fact type. However, if the first noun concept is role, the noun\(_1\) has to be the general concept of the role.
\item noun\(_2\) is the second noun concept that is involved in the representation of fact type. Similar to the noun\(_1\), if the second noun concept is role, it has to be the general concept of the role. When the fact type has only one noun concept, noun\(_2\) is unspecified.
\item role\(_1\) is the first noun concept in the representation of the fact type if the noun concept is role. Otherwise, it obtains the same value of noun\(_1\) if the noun concept is objectType or categorizationTypeInstances.
\item role\(_2\) is the second noun concept that is included the representation if the noun concept is role. Else, it gains the identical value of noun\(_2\) when the noun concept is objectType or categorizationTypeInstances.
\item verb is the actual verb that is used in the representation of the fact type.
\item reversedVerb is the reverse verb of the actual verb that is included in the representation of the fact type. The reverse verb allows the second noun concept to come first in the representation with the same meaning for the fact type. The fact types that have only one noun concept do not have a reversedVerb.
\end{enumerate}

The properties noun\(_1\), noun\(_2\), role\(_1\), role\(_2\), and verb are directly extracted from the representation of the fact types. However, the reversedVerb property has to be explicitly defined in the entry of the fact types, as it is not a part from the representation of the fact type. All these properties are added to the SBVR fact types to reduce the complexity and increase the correctness level of the transformation.

Constraint 1.3.1. The representations of fact types must be unique. That is,
\[ \forall f_1, f_2: \text{FactTypes} \ni f_1 \neq f_2 \Rightarrow f_1.\text{representation} \neq f_2.\text{representation} \]
\[ \quad // f_1, f_2 \text{ are unique fact types} \]

Constraint 1.3.2. The representations and concept types of fact types have to be specified. That is,
\[ \forall f: \text{FactTypes} \ni f.\text{representation} \neq \bot \land f.\text{conceptType} \neq \bot \]
\[ \quad // \text{the representation and conceptType of } f \text{ are not unspecified} \]

Constraint 1.3.3. The first noun concept and the verb in the representation of fact types must be specified. This implies that the noun\(_1\), role\(_1\), verb have to be specified as they are directly extracted from the representation of fact types. That is,
\[ \forall f: \text{FactTypes} \ni f.\text{noun}_1 \neq \bot \land f.\text{role}_1 \neq \bot \land f.\text{verb} \neq \bot \]
\[ \quad // \text{the noun}_1, \text{role}_1, \text{and verb of } f \text{ are not unspecified} \]
**Constraint 1.3.4.** The representations of fact types can only include existing objectTypes, roles, or categorizationTypeInstances. Therefore, noun$_1$ and noun$_2$ can be only objectTypes or categorizationTypeInstances and role$_1$ and role$_2$ can be only roles as they all are extracted from the representations of fact types. That is,

\[
\forall f: \text{FactTypes} \bullet f.\text{noun}_1 = \{x: \text{NounConcepts} \bullet x.\text{conceptType} = \text{'objectType'} \lor x.\text{conceptType} = \\
\{y: \text{NounConcepts} \bullet y.\text{conceptType} = \text{'categorizationType'} \bullet y.\text{representation}\} \bullet x.\text{representation}\} \land \\
f.\text{noun}_2 = \{x: \text{NounConcepts} \bullet x.\text{conceptType} = \text{'objectType'} \lor x.\text{conceptType} = \\
\{y: \text{NounConcepts} \bullet y.\text{conceptType} = \text{'categorizationType'} \bullet y.\text{representation}\} \bullet x.\text{representation}\} \land \\
f.\text{role}_1 = \{x: \text{NounConcepts} \bullet x.\text{conceptType} = \text{'role'} \bullet x.\text{representation}\} \land \\
f.\text{role}_2 = \{x: \text{NounConcepts} \bullet x.\text{conceptType} = \text{'role'} \bullet x.\text{representation}\}
\]

// the noun concepts that are included in the representation of f are object
// Types, roles, or categorizationTypesInstances.

In table 3, the entry example of the fact type ‘department is managed by manager’ can be specified as:

a) \(f.\text{representation} = \text{‘department is managed by manager’}\)

b) \(f.\text{conceptType} = \text{associative}\)

c) \(f.\text{noun}_1 = \text{‘department’}\)

d) \(f.\text{noun}_2 = \text{‘employee’}\)

e) \(f.\text{role}_1 = \text{‘department’}\)

f) \(f.\text{role}_2 = \text{‘manager’}\)

g) \(f.\text{verb} = \text{‘is managed by’}\)

h) \(f.\text{reversedVerb} = \text{‘manages’}\)

**Definition 1.4.** CategorizationSchemes is a set of categorization schemes. A categorization scheme \(cs \in \text{CategorizationSchemes}\) is a tuple \(cs = (\text{representation}, \text{type}, \text{definition}, \text{necessity}, \text{associatedCategorizationType}, \text{parentNounConcept}, \text{childrenNounConcepts})\), where:

a) \(\text{representation}\) is the primary representation of the categorization scheme.

b) \(\text{type}\) is the type of the categorization scheme, which can be either categorizationScheme or Segmentation.

c) \(\text{definition}\) is the definition of the categorization scheme. It defines the parentNounConcept and associatedCategorizationType.

d) \(\text{necessity}\) is the necessity statement of the categorization scheme. It is mainly used to identify the childrenNounConcepts for the categorization scheme.

e) \(\text{associatedCategorizationType}\) is a CategorizationType that is used with the categorization scheme to categorize the parentNounConcept into the childrenNounConcepts set.

f) \(\text{parentNounConcept}\) is an objectType or categorizationTypeInstance that are categorized and constrained by the categorization scheme and the associated categorization type.

g) \(\text{childrenNounConcepts}\) is a set of categorizationTypeInstances, which contains the categories of parentNounConcept.
The associatedCategorizationType and parentNounConcept properties are extracted from the definition of the categorization scheme cs. And the childrenNounConcepts set is extracted from the necessity of the categorization scheme cs.

**Constraint 1.4.1.** The representations of categorization schemes must be unique. That is,

\[ \forall cs_1, cs_2: \text{CategorizationSchemes} \cdot cs_1 \neq cs_2 \implies cs_1.\text{representation} \neq cs_2.\text{representation} \]

// cs1 and cs2 are unique categorization schemes

**Constraint 1.4.2.** The representations, definitions, and necessitates of categorization schemes have to be specified. That is,

\[ \forall cs: \text{CategorizationSchemes} \cdot cs.\text{representation} \neq \bot \land cs.\text{definition} \neq \bot \land cs.\text{necessity} \neq \bot \]

// the representation, definition, and necessity of cs are not unspecified

**Constraint 1.4.3.** The associated categorization types of categorization schemes must be existing categorizationTypes. That is,

\[ \forall cs: \text{CategorizationSchemes} \cdot cs.\text{associatedCategorizationType} = \{ x: \text{NounConcepts} \cdot x.\text{conceptType} = \text{`categorizationType`} \cdot x.\text{representation} \} \]

// the associatedCategorizationType of cs is an existing categorizationType

**Constraint 1.4.4.** The parent noun concepts of categorization schemes must be existing objectTypes or categorizationTypeInstances. That is,

\[ \forall cs: \text{CategorizationSchemes} \cdot cs.\text{parentNounConcept} = \{ x: \text{NounConcepts} \cdot x.\text{conceptType} = \text{`objectType`} \vee x.\text{conceptType} = \{ y: \text{NounConcepts} \cdot y.\text{conceptType} = \text{`categorizationType`} \cdot y.\text{representation} \} \cdot x.\text{representation} \}

// the parentNounConcept of cs is an existing objectType or // categorizationTypeInstance

**Constraint 1.4.5.** The elements of the children noun concepts sets of categorization schemes must be existing categorizationTypeInstances of the associatedCategorizationType. That is,

\[ \forall cs: \text{CategorizationSchemes} \cdot cs.\text{childrenNounConcepts} \in \{ \forall x: \text{NounConcepts} \cdot x.\text{conceptType} = \{ cs.\text{associatedCategorizationType} \cdot x.\text{representation} \} \} \]

// the elements of childrenNounConcepts set of cs are existing

// categorizationTypeInstances of the associatedCategorizationType

The entry example of the categorization scheme ‘employee by type’ in table 3 can be specified as:

a) cs.representation = \text{employee by type}

b) cs.type = categorization scheme

c) cs.definition = \text{categorization scheme that is for the concept \text{employee} and subdivides employee based on employment type}

d) cs.necessity = \text{Employee by type contains the categories part time employee and full time employee}

e) cs.associatedCategorizationType = \text{employment type}
f) \( cs\.parentNounConcept = \) \texttt{employee}’
g) \( cs\.childrenNounConcepts = \{ \) \texttt{part time employee}, \texttt{full time employee} \}’

1.2 SBVR Keywords

Definition 2 SBVR Keywords is a set of Keywords. A keyword \( k \in \text{Keywords} \) is a tuple \( k = (\text{representation}, \text{type}) \), where:

a) \text{representation} is the primary representation of the keyword.
b) \text{type} is the type of the keyword. It can be one of modal, quantification, logical, or general.

Constraint 2.1. The representations of keywords must be unique. That is,

\[ \forall k_1, k_2: \text{Keywords} \cdot k_1 \neq k_2 \Rightarrow k_1\.representation \neq k_2\.representation \]

// \( k_1, k_2 \) are unique keywords

Constraint 2.2. The representations and types of keywords have to be specified. That is,

\[ \forall k: \text{Keywords} \cdot k\.representation \neq \bot \land k\.type \neq \bot \]

// the representation and type of \( k \) are not unspecified

Constraint 2.3. The types of keywords must be modal, quantification, logical, or general. That is,

\[ \forall k: \text{Keywords} \Rightarrow k\.type = \) \texttt{‘modal’}’ \lor k\.type = \texttt{‘quantification’}’ \lor k\.type = \texttt{‘logical’}’ \lor k\.type = \texttt{‘general’}’

// the type of \( k \) is modal, quantification, logical, or general

For example, the keyword \texttt{each} can be specified as:

a) \( k\.representation = \) \texttt{each}
b) \( k\.type = \texttt{quantification} \)

1.3 SBVR Business Rules

Definition 3. SBVR BusinessRules is a set of business rules. A business rule \( br \in \text{BusinessRules} \) is a tuple \( br = (\text{statement}, \text{factType}, \text{verbForm}, \text{modal}, \text{quantifier}_1, \text{quantifier}_2) \), where:

a) \text{statement} is the primary statement of the business rule. It is built by combining fact types and keywords.
b) \text{factType} is the fact type that is used to build the business rule statement.
c) \text{verbForm} shows the form of the verb that are used in the fact type involved in the business rules statement. The verb form can be either actual or reversed form. The actual form means the verb used in the representation of the fact type not the active voice of the verb).
d) \text{modal} is the modal keyword that is included in the business rule statement.
e) \text{quantifier}_1 is the first quantification keyword that is involved in the business rule statement. It always belongs to the first noun concept of the fact type.
f) \text{quantifier}_2 is the second quantification keyword that is employed in the business rule statement. It always belongs to the second noun concept in the fact type.
Constraint 3.1. The statement of business rules must be unique. That is,
\[ \forall \ br_1, \ br_2: BusinessRules \cdot br_1 \not= \ br_2 \Rightarrow \ br_1.\statement \not= \ br_2.\statement \]
// \( br_1, br_2 \) are unique business rules

Constraint 3.2. The statements of business rules have to be specified. That is,
\[ \forall \ br: BusinessRules \cdot br.\statement \not= \bot \]
// the statement of \( br \) are not unspecified

Constraint 3.3. The statements of business rules must include existing fact types. That is,
\[ \forall \ br: BusinessRules \cdot br.\factType \in \{ x: \FactTypes \cdot x.\representation \} \]
// the statement of \( br \) has an existing fact type

Constraint 3.4. The modal keywords in the statements of business rules must be specified. That is,
\[ \forall \ br: BusinessRules \cdot br.\modal \not= \bot \]
// the model in the statement of \( br \) are not unspecified

Constraint 3.5: The statements of business rules must include existing modal keywords. That is,
\[ \forall \ br: BusinessRules \cdot br.\modal = \{ x: Keywords \cdot x.\type = \text{`modal'} \cdot x.\representation \} \]
// the statement of \( br \) has an existing modal keyword

Constraint 3.6: The statements of business rules can only include existing quantification keywords. That is,
\[ \forall \ br: BusinessRules \cdot br.\quantifier_1 \in \{ x: Keywords \cdot x.\type = \text{`quantification'} \cdot x.\representation \} \]
\[ \forall \ br.\quantifier_1 = \bot \land \ br.\quantifier_2 \in \{ y: Keywords \cdot y.\type = \text{`quantification'} \cdot y.\representation \} \]
\[ \forall \ br.\quantifier_2 = \bot \]
// the statement of \( br \) has existing quantification keywords

The \( \factType, \modal, \quantifier_1, \) and \( \quantifier_2 \) properties are directly extracted from the statement of the business rules while the \( \verbForm \) property is determined (whether actual or reversed) by the \( \verb \) involved in the \( \factType \) property. These properties are appended to the SBVR business rules to decrease the complexity of the transformation and increase its level of correctness.

For example, the business rule ‘It is necessary that each department is managed by exactly one manager’ provided in table 3 can be specified as:
a) \( br.\statement = \text{‘It is necessary that each department is managed by exactly one manager’} \)
b) \( br.\factType = \text{‘agent manages property’} \)
c) \( br.\verbForm = \text{actual} \)
d) \( br.\modal = \text{‘It is necessary’} \)
e) \( br.\quantifier_1 = \text{‘each’} \)
f) \( br.\quantifier_2 = \text{‘at most ten’} \)
The formal model of the core SBVR discussed above provides some substantial benefits for the implementation stage (discussed later), which make the development of the automated transformation tool more accurate and efficient. Therefore, in developing the automated tool, the structure defined by the formal model of the core SBVR are directly translated and converted to the schema design of the SBVR repository. As a result, each element, attribute, or relationship in the schema design of the SBVR repository reflects an element, property, or relationship in the formal model of the core SBVR. Again, the structure of the core SBVR formal model is used to design the user interfaces and the parsing engine that are employed to capture all kinds of the core SBVR elements with their properties. This can ensure that there is no SBVR element or property can be missed. Moreover, the constraints included in the formal model of the core SBVR are entirely converted and added into the validation engine that is responsible for validating any SBVR element before it is stored in the SBVR repository. This can ensure that all stored SBVR elements are valid and fulfil their related constraints defined in the formal model of the core SBVR.

1.1 UML Formal Model

The UML class diagram essentially contains seven main elements, which are classes, associations, aggregations, inheritances, generalization sets, enumerations, and basic data types. Each element has its own properties associated with a set of constraints that is used for validation. The UML formal model precisely describes these main elements with their properties and constraints.

Definition 4. UML class diagram is a tuple \( \text{classDiagram} = (\text{Classes}, \text{Associations}, \text{Aggregations}, \text{Inheritances}, \text{GeneralizationSets}, \text{Enumerations}, \text{BasicDataTypes}) \).

Definition 4.1. \textit{Classes} is a set of classes. A class \( c \in \text{Classes} \) is a tuple \( c = (\text{name}, \text{attributes}) \), where:

- a) name is the name of the class.
- b) An attribute \( att \in \text{attributes} \) is a tuple \( att = (\text{name}, \text{dataType}, \text{card}) \), where:
  - i. name is the name of the attribute.
  - ii. dataType is the data type of the attribute. A data type can be one of the basic data types that are defined in SBVR standard or user-defined data types (enumerations).
  - iii. card is the cardinality of the attribute. It is a range \( \text{min} .. \text{max} \), where \( \text{min} \) can be 0 and \( \text{max} \) can be infinitive (\( \infty \)) but a range cannot be 0..0. (In this paper, for simplicity, the cardinality of discontinuous range such as 1..5, 10 is not considered).

Constraint 4.1.1. The names of classes must be unique. That is,

\[ \forall \ c_1, c_2: \text{Classes} \cdot c_1 \neq c_2 \Rightarrow c_1.\text{name} \neq c_2.\text{name} \]

// \( c_1 \) and \( c_2 \) are unique classes

Constraint 4.1.2. The names of classes have to be specified. That is,

\[ \forall \ c: \text{Classes} \cdot c.\text{name} \neq \bot \]

// the name of \( c \) is not unspecified
**Constraint 4.1.3.** The names of the attributes within a class must be unique. That is,

\[ \forall c: \text{Classes} \cdot \forall \text{att}_1, \text{att}_2: \text{c.attributes} \cdot \text{att}_1 \neq \text{att}_2 \Rightarrow \text{att}_1.\text{name} \neq \text{att}_2.\text{name} \]

// \text{att}_1 and \text{att}_2 are unique attributes in the class \text{c}

**Constraint 4.1.4.** The names of the attributes of a class have to be specified. That is,

\[ \forall c: \text{Classes} \cdot \forall \text{att}: \text{c.attributes} \cdot \text{att}.\text{name} \neq \perp \]

// the name of \text{att} in \text{c.attributes} for the class \text{c} are not unspecified

**Constraint 4.1.5.** The data types of attributes must be an existing basic data type or enumeration. That is,

\[ \forall c: \text{Classes} \cdot \forall \text{att}: \text{c.attributes} \cdot \text{att}.\text{dataType} \in \text{BasicDataTypes} \lor \text{att}.\text{dataType} \in \{x: \text{Enumerations} \cdot x.\text{name}\} \]

// the \text{dataType} of \text{att} is an existing basic data type or enumeration

**Constraint 4.1.6.** The minimum of cardinality range of attributes must be greater than or equal to zero. That is,

\[ \forall c: \text{Classes} \cdot \forall \text{att}: \text{c.attributes} \cdot \text{att}.\text{card.min} \geq \text{zero} \]

// the \text{min of card} for \text{att} is greater than or equal to \text{zero}

**Constraint 4.1.7.** The maximum of cardinality range of attributes must be greater than zero. That is,

\[ \forall c: \text{Classes} \cdot \forall \text{att}: \text{c.attributes} \cdot \text{att}.\text{card.max} > \text{zero} \]

// the \text{max of card} for \text{att} is greater than \text{zero}

**Constraint 4.1.8.** The minimum of cardinality range of attributes must be less than or equal to the maximum. That is,

\[ \forall c: \text{Classes} \cdot \forall \text{att}: \text{c.attributes} \cdot \text{att}.\text{card.min} \leq \text{att}.\text{card.max} \]

// the \text{min of card} for \text{att} is less than or equal to \text{max}

**Definition 4.2.** **Associations** is a set of association relationships. An association \( \text{asso} \in \text{Associations} \) is a tuple \( \text{asso} = (\text{name}, \text{end}_1, \text{end}_2) \), where:

a) \text{name} is the name of the association.

b) \text{end}_1 is the first end in the association. \text{end}_1 is a tuple \( \text{end}_1 = (\text{class}, \text{classRole}, \text{card}) \), where:
   i. \text{class} is the associated class with the end.
   ii. \text{classRole} is the role of the associated class with the \text{end}_1 in the association.
   iii. \text{card} is the cardinality of the \text{end}_1. It is a range \text{min} ... \text{max}, where \text{min} can be 0 and \text{max} can be infinitive \((\infty)\) which means there is no finite upper limits. A range cannot be 0...0.

c) \text{end}_2 is the second end in the association. \text{end}_2 is also a tuple and it has the same elements as \text{end}_1.
Constraint 4.2.1: The names of associations do not have to be unique. However, associations between the same two classes must have different names. That is,

\[ \forall \text{asso}_1, \text{asso}_2: \text{Associations} \cdot \{ \text{asso}_1 \cdot \text{asso}_1.\text{end}_1.\text{class}, \text{asso}_1.\text{end}_2.\text{class} \} = \{ \text{asso}_2 \cdot \text{asso}_2.\text{end}_1.\text{class}, \text{asso}_2.\text{end}_2.\text{class} \} \land \text{asso}_1 \neq \text{asso}_2 \Rightarrow \text{asso}_1.\text{name} \neq \text{asso}_2.\text{name} \]

// asso1 and asso2 between the same classes have different names

Constraint 4.2.2: The associated classes with end1 and end2 for associations have to be specified. That is,

\[ \forall \text{asso}: \text{Associations} \cdot \text{asso}.\text{end}_1.\text{class} \neq \bot \land \text{asso}.\text{end}_2.\text{class} \neq \bot \]

// the associated classes with end1 and end2 in asso are not unspecified

Constraint 4.2.3: The roles of end1 and end2 within an association have to be unique. That is,

\[ \forall \text{asso}: \text{Associations} \cdot \text{asso}.\text{end}_1.\text{classRole} \neq \text{asso}.\text{end}_2.\text{classRole} \]

// the classRoles of end1 and end2 in asso are unique

Constraint 4.2.4: The minimum of cardinality range of end1 and end2 must be greater than or equal to zero. That is,

\[ \forall \text{asso}: \text{Associations} \cdot \text{asso}.\text{end}_1.\text{card}.\text{min} \geq \text{zero} \land \text{asso}.\text{end}_2.\text{card}.\text{min} \geq \text{zero} \]

// the min of card for end1 and end2 is greater than or equal to zero

Constraint 4.2.5: The maximum of cardinality range of end1 and end2 must be greater than zero. That is,

\[ \forall \text{asso}: \text{Associations} \cdot \text{asso}.\text{end}_1.\text{card}.\text{max} > \text{zero} \land \text{asso}.\text{end}_2.\text{card}.\text{max} > \text{zero} \]

// the max of card for end1 and end2 is greater than zero

Constraint 4.2.6: The minimum of cardinality range of end1 and end2 must be less than or equal to the maximum. That is,

\[ \forall \text{asso}: \text{Associations} \cdot \text{asso}.\text{end}_1.\text{card}.\text{min} \leq \text{asso}.\text{end}_1.\text{card}.\text{max} \land \text{asso}.\text{end}_2.\text{card}.\text{min} \leq \text{asso}.\text{end}_2.\text{card}.\text{max} \]

// the min of card for end1 and end2 is less than or equal to the max

Definition 4.3 Aggregations is a set of aggregation relationships. An aggregation aggr (name, containerEnd, containedEnd), where:

a) name is the name of the aggregation.
b) containerEnd is the end that is connected with the container class in the aggregation. containerEnd is a tuple containerEnd = (class, classRole, card), where:
   i. class is the container class.
   ii. classRole is the role of the container class in the aggregation.
   iii. card is the cardinality of the container end. It is a range min .. max.
c) containedEnd is the end that is associated with the contained class in the aggregation. containedEnd is also a tuple and it has the same elements as containerEnd.

Constraint 4.3.1: The names of aggregation do not have to be unique. However, aggregations between the same two classes must have different names. That is,
\( \forall \text{aggr}_1, \text{aggr}_2: \) Aggregations \( \bullet \) \{aggr\_1.containerEnd.class, aggr\_1.containedEnd.class\} =
\{aggr\_2.containerEnd.class, aggr\_2.containedEnd.class\} \land \text{aggr}_1 \neq \text{aggr}_2 \Rightarrow \text{aggr}_1.name \neq \text{aggr}_2.name

// aggr\_1 and aggr\_2 between the same classes have different names

**Constraint 4.3.2**: The container and contained classes in aggregation ends have to be specified. That is,
\( \forall \text{aggr}: \) Aggregations \( \bullet \) \text{aggr}.containerEnd.class \neq \bot \land \text{aggr}.containedEnd.class \neq \bot

// the container and contained classes of the ends in aggr are not unspecified

**Constraint 4.3.3**: The roles of container and contained classes within an aggregation have to be unique. That is,
\( \forall \text{Aggr}: \) Aggregations \( \bullet \) \text{aggr}.containerEnd.classRole \neq \text{aggr}.containedEnd.classRole

// the classRoles of containerEnd and containedEnd in aggr are unique

**Constraint 4.3.4**: A contained class in an aggregation cannot be a container for its original container class in another aggregation (Anti-symmetric). e.g. if class \( c_1 \) contains class \( c_2 \) in an aggregation \( aggr_1 \), then class \( c_2 \) cannot contain class \( c_1 \) in another aggregation \( aggr_2 \). That is,
\( \neg \exists c_1, c_2: \) Classes \( \bullet c_1 \neq c_2 \land (c_1, c_2) \in \text{contains} \land (c_2, c_1) \in \text{contains} \)

// \( c_1 \) and \( c_2 \) do not contain each other

**Constraint 4.3.5**: A class cannot contain itself directly or indirectly (Acyclic). That is,
\( \neg \exists c: \) Classes \( \bullet (c, c) \in \text{contains}^* \setminus \text{contains} \)

// the class \( c \) does not contain itself directly or indirectly

The constraints 4.3.4 and 4.3.5 use the function \text{contains} that check whether a class contains another class in an aggregation or not.

**Function 1** \text{contains} is defined as follow:

\text{contains}: \text{Classes} \rightarrow \text{Classes}
\forall c_1, c_2: \text{Classes}((c_1, c_2) \in \text{contains} \Leftrightarrow
\exists \text{aggr}: \text{Aggregations} \bullet \text{aggr}.containerEnd.class = c_1 \land \text{aggr}.containedEnd.class = c_2)

**Constraint 4.3.6**: The minimum of cardinality range for both container and contained ends must be greater than or equal to zero. That is,
\( \forall \text{aggr}: \) Aggregations \( \bullet \text{aggr}.containerEnd.card.min \geq \) zero \land \text{aggr}.containedEnd.card.min \geq zero

// the min of card for containerEnd and containedEnd is greater than or equal to zero

**Constraint 4.3.7**: The maximum of cardinality range of container and contained ends must be greater than zero. That is,
\( \forall \text{aggr}: \) Aggregations \( \bullet \text{aggr}.containerEnd.card.max > \) zero \land \text{aggr}.containedEnd.card.max > zero

// the max of card for containerEnd and containedEnd is greater than zero

11
**Constraint 4.3.8**: The minimum of cardinality range of container and contained ends must be less than or equal to the maximum. That is,

\[ \forall \text{aggr}:\text{Aggregations} \bullet \text{aggr.containerEnd.card.min} \leq \text{aggr.containerEnd.card.max} \land \text{aggr.containedEnd.card.min} \leq \text{aggr.containedEnd.card.max} \]

// the min of card for containerEnd and containedEnd is less than or equal to the max

**Definition 4.4** *Inheritances* is a set of inheritance relationships. Inheritance relationships are quite different from other relationships (associations and aggregations) as their ends do not have names, roles, and cardinality. Thus, an inheritance \( \text{inher} \in \text{Inheritances} \) is a set \( \text{inher} = \{\text{parentEnd}, \text{childEnd}\} \), where:

a) \( \text{parentEnd} \) is the parent class in the inheritance relationship.

b) \( \text{childEnd} \) is the child class that is inherited from the parent class in the inheritance relationship.

**Constraint 4.4.1**: The parent and child ends in inheritances have to be specified. That is,

\[ \forall \text{inher}: \text{Inheritances} \bullet \text{inher.parentEnd} \neq \bot \land \text{inher.childEnd} \neq \bot \]

// the parentEnd and childEnd of inher are not unspecified

**Constraint 4.4.2**: A child class in an inheritance cannot be a parent of its original parent class in another inheritance (Anti-symmetric). e.g. if class \( c_1 \) is parent of class \( c_2 \) in an inheritance \( \text{inher}_1 \), then class \( c_2 \) cannot be parent of class \( c_1 \) in another inheritance \( \text{inher}_2 \). That is,

\[ \neg \exists \ c_1, c_2: \text{Classes} \bullet c_1 \neq c_2 \land (c_1, c_2) \in \text{isParentOf} \land (c_2, c_1) \in \text{isParentOf} \]

// \( c_1 \) and \( c_2 \) do not inherit from each other

**Constraint 4.4.3**: A class cannot be a parent of itself directly or indirectly (Acyclic). That is,

\[ \neg \exists \ c: \text{Classes} \bullet (c, c) \in \text{isParentOf} \land \neg \text{isParentOf} \]

// the class \( c \) does not inherit itself directly or indirectly

The constraints 4.4.2 and 4.4.3 use the function \( \text{isParentOf} \) that check whether a class is a parent of another class in an inheritance or not.

**Function 2** \( \text{isParentOf} \) is defined as follow:

\[ \text{isParentOf}: \text{Classes} \Rightarrow \text{Classes} \]

\[ \forall \ c_1, c_2: \text{Classes}((c_1, c_2) \in \text{isParentOf} \Rightarrow \exists \text{inher}: \text{Inheritances} \bullet \text{inher.parentEnd} = c_1 \land \text{inher.childEnd} = c_2) \]

**Definition 4.5** *GeneralizationSets* is a set of generalization sets. A generalization set \( \text{genSet} \in \text{GeneralizationSets} \) is a tuple \( \text{genSet} = (\text{name}, \text{constraint}, \text{parent}, \text{children}) \), where:

a) \( \text{name} \) is the name of the generalization set.

b) \( \text{constraint} \) is the constraint of the generalization set.

c) \( \text{parent} \) is the parent class associated with the generalization set.

d) \( \text{children} \) is a set of the children classes associated with the generalization set.
Constraint 4.5.1: The names of generalization sets have to be unique. That is,
\[ \forall \text{genSet}_1, \text{genSet}_2: \text{GeneralizationSets} \cdot \text{genSet}_1 \neq \text{genSet}_2 \Rightarrow \text{genSet}_1.name \neq \text{genSet}_2.name \]
// \text{genSet}_1 and \text{genSet}_2 are unique generalization sets

Constraint 4.5.2: The names, constraints, and parents of generalization sets have to be specified. That is,
\[ \forall \text{genSet}: \text{GeneralizationSets} \cdot \text{genSet}.name \neq \bot \land \text{genSet.constraint} \neq \bot \land \text{genSet.parent} \neq \bot \]
// the name, constraint, and parent of \text{genSet} are not unspecified

Constraint 4.5.3: The children of generalization sets must be non-empty sets. That is,
\[ \forall \text{genSet}: \text{GeneralizationSets} \cdot \text{genSet}.children \neq \emptyset \]
// the children set of \text{genSet} is not an empty set

Constraint 4.5.4: The parents and children classes must belong to the classes set (existing classes). That is,
\[ \forall \text{genSet}: \text{GeneralizationSets} \cdot \text{genSet}.parent \in \text{Classes} \land \text{genSet.children} \in \text{Classes} \]
// the parent and children of \text{genSet} belong to \text{Classes}

Constraint 4.5.5: The parents and children classes in the generalization sets must have existing inheritance relationships between them. That is,
\[ \forall \text{genSet}: \text{GeneralizationSets} \cdot \forall \text{child}: \text{genSet}.children \cdot \exists \text{inher}: \text{Inheritances} \cdot \]
\[ \text{inher} = (\text{getClass}(\text{genSet}.parent), \text{getClass}(\text{child})) \]
// the parent class and each child class of \text{genSet} have \text{Inheritances} // between them

Constraint 4.5.5 employs the function \text{getClass}, which obtains a string as an input then retrieves the class whose name matches the input string from the \text{Classes} set.

Function 3 \text{getClass} is defined as following:
\[ \text{getClass} : \text{Text} \rightarrow \text{Classes} \]
\[ \forall \text{name}: \text{Text}, \text{class} : \text{Classes} \]
\[ (\text{name}, \text{class}) \in \text{getClass} \Leftrightarrow \text{class.name} = \text{name} \]

Constraint 4.5.6: The parents and children combined cannot appear more than once in generalization sets. That is,
\[ \neg \exists \text{genSet}_1, \text{genSet}_2: \text{GeneralizationSets} \cdot \text{genSet}_1.name \neq \text{genSet}_2.name \land \text{genSet}_1.parent = \text{genSet}_2.parent \land \text{genSet}_1.children = \text{genSet}_2.children \]
// the same parent and children do not appear in more than one generalization set

Definition 4.6 \text{Enumerations} is a set of enumerations. An enumeration \text{enum} \in \text{Enumerations} is a tuple \text{enum} = (\text{name}, \text{literals}), where:

a) \text{name} is the name of the enumeration.

b) \text{literals} is a set of literals for the enumeration. Each literal \text{literal} \in \text{literals} has to be a unique and not repeated in each literal set but it can be used in another literal set for another enumeration.
**Constraint 4.6.1**: The names of enumerations have to be unique. That is,

\[ \forall \text{enum}_1, \text{enum}_2: \text{Enumerations} \bullet \text{enum}_1 \neq \text{enum}_2 \Rightarrow \text{enum}_1\text{.name} \neq \text{enum}_2\text{.name} \]

// enum\(_1\) and enum\(_2\) are unique enumerations

**Constraint 4.6.2**: The names of enumerations have to be specified. That is,

\[ \forall \text{enum}: \text{Enumerations} \bullet \text{enum}\text{.name} \neq \bot \]

// the name of enum is not unspecified

**Constraint 4.6.3**: The literal sets of enumerations must be non-empty sets. That is,

\[ \forall \text{enum}: \text{Enumerations} \bullet \text{enum}\text{.literals} \neq \emptyset \]

// the literals of enum is not an empty set

**Constraint 4.6.4**: The literals of the literal set within an enumeration must be unique. That is,

\[ \forall \text{enum}: \text{Enumerations} \bullet \forall \text{literal}_1, \text{literal}_2: \text{enum}\text{.literals} \bullet \text{literal}_1 \neq \text{literal}_2 \]

// literal\(_1\) and literal\(_2\) of literals for enum are unique literals

**Definition 4.7** \textit{BasicDataTypes} is a set of basic data types. A basic data type \(bdt \in \text{BasicDataTypes}\) is an element of the set \text{BasicDataTypes}.

**Constraint 4.7.1**: The basic data types must be unique. That is,

\[ \forall bdt_1, bdt_2: \text{BasicDataTypes} \bullet \Rightarrow bdt_1 \neq bdt_2 \]

\ // bdt\(_1\) and bdt\(_2\) are unique basic data types

**Constraint 4.7.2**: The basic data types have to be specified. That is,

\[ \forall bdt: \text{BasicDataTypes} \Rightarrow bdt \neq \bot \]

\ // the basic data type bdt is not unspecified

Similar to the formal model of the core SBVR, having the UML formal model defined above makes the development of the automated tool (discussed later) more precise and efficient. This is because that we directly use the structure of the UML formal model to produce the schema design of the UML repository. We also embed the constraints of the UML formal model into the tables of the UML repository and this can ensure that only valid UML elements are stored in the UML repository.