

Contents

1 Axiomatic Data Types

1.1 Atomic Data Types

This section identifies the atomic data types that are assumed for the rest of this specification, specifically:

- **SCTID** – a SNOMED CT identifier
- **TERM** – a fully specified name, preferred term or synonym for a SNOMED CT Concept
- **REAL** – a real number
- **STRING** – a string literal
- **GROUP** – a role group identifier
- \mathbb{N} – a non-negative integer
- \mathbb{Z} – an integer

We also introduce several synonyms for *SCTID*:

- **SUBJECT** – a *SCTID* that appears in the *sourceId* position of a relationship.
- **ATTRIBUTE** – a *SCTID* that appears in the *typeId* position of a relationship.
- **OBJECT** – a *SCTID* that appears in the *destinationId* position of a relationship.
- **REFSETID** – a *SCTID* that identifies a reference set

[*SCTID*, *TERM*, *REAL*, *STRING*, *GROUP*]

SUBJECT == *SCTID*

ATTRIBUTE == *SCTID*

OBJECT == *SCTID*

REFSETID == *SCTID*

1.2 Composite Data Types

While we can't fully specify the behavior of the concrete data types portion of the specification at this point, it is still useful to spell out the anticipated behavior on an abstract level.

- **CONCRETEVALUE** – a string, integer or real literal
- **TARGET** – the target of a relationship that is either an **OBJECT** or a **CONCRETEVALUE**

CONCRETEVALUE ::= *string*⟨⟨*STRING*⟩⟩ | *integer*⟨⟨ \mathbb{Z} ⟩⟩ | *real*⟨⟨*REAL*⟩⟩

TARGET ::= *object*⟨⟨*OBJECT*⟩⟩ | *concrete*⟨⟨*CONCRETEVALUE*⟩⟩

2 The Substrate

A substrate represents the context of an interpretation. A substrate consists of:

- **c** The set of *SCTIDs* (concepts) that are considered valid in the context of the substrate. **References to any *SCTID* that is not a member of this set MUST be treated as an error.**
- **a** The set of *SCTIDs* that are considered to be valid attributes in the context of the substrate. **Reference to any *ATTRIBUTE* that is not a member of this set MUST be treated as an error.**
- **r** A set of relationship quads (subject, attribute, target, group)
- **descs** The subsumption (ISA) closure from general to specific (descendants)
- **ancs** The subsumption (ISA) closure from specific to general (ancestors)
- **rs** The reference sets within the context of the substrate whose members are members are concept identifiers (i.e. are in *c*). While not formally spelled out in this specification, it is assumed that the typical reference set function would be returning a subset of the *refsetId/referencedComponentId* tuples represented in one or more RF2 Refset Distribution tables.

Open questions:

- While the Core will not contain cycles, and probably NRC Extensions will not, arbitrary extensions (e.g. as a result of handling Expression libraries) may involve cycles (or at least equivalent concepts). This raises questions like should *descendantsOf* (*c1*) include *c1* and *c2* if $c1 \equiv c2$.
- Is it correct to interpret the transitive closure directly against the substrate relationships (*r*) set or is there an implication of a reasoner being invoked somewhere, which could potentially render the transitive closure as logically derived from (*r*). (I'd argue for the former, which is how it is spelled out below.
- Do we want to assert that it is an error to use a non-attribute concept in a attribute position? If not, should we explicitly include a formal definition of the attribute set?

2.1 Substrate Components

Quad Relationships in the substrate are represented a 4 element tuples or “Quads”, which consist of a subject, attribute, target and role group identifier.

Quad

s : *SUBJECT*

a : *ATTRIBUTE*

t : *TARGET*

g : *GROUP*

We will also need to recognize some "well known" identifiers: the *is_a* attribute, the *zero_group* and *linkage_concept*, the parent of all attributes

	<i>is_a</i> : <i>ATTRIBUTE</i>
	<i>zero_group</i> : <i>GROUP</i>
	<i>linkage_concept</i> : <i>SCTID</i>

2.2 Substrate

The formal definition of substrate follows, where *c* and *r* are given and the remainder are derived. The expressions below assert that:

1. All subjects and attributes and targets of type *object* in *r* must be members of the set of concepts, *c*. All attributes are also members of *a*.
2. The *is_a* relation is irreflexive.
3. The *childOf* relationship is defined as the function from a concept in *c* to the set of concepts that are the target of *c* in an *is_a* relationship in the *zero_group*
4. The *parentOf* relationship is defined as a function from a concept *c* to the set of concepts that are the source of *c* in an *is_a* relationship in the *zero_group*
5. The *parentOf* and *childOf* relationships are irreflexive.
6. The ancestors function, *ancs* is the *irreflexive* transitive closure of the *childOf* relationship.
7. The descendants, *descs* is the *irreflexive* transitive closure of the inverse of the *parentOf* relationship.
8. The reference set function is a function from *subset of c* to set of SCTID's in *c*. **An SCTID that is not in the domain of *rs* cannot appear as the target of a memberOf function**

$ \begin{array}{l} \textit{Substrate} \\ c : \mathbb{P} \textit{SCTID} \\ r : \mathbb{P} \textit{Quad} \\ \\ a : \mathbb{P} c \\ \textit{descs} : c \rightarrow \mathbb{P} c \\ \textit{ancs} : c \rightarrow \mathbb{P} c \\ rs : \textit{REFSETID} \rightarrow \mathbb{P} c \\ \\ \textit{childOf} : c \rightarrow \mathbb{P} c \\ \textit{parentOf} : c \rightarrow \mathbb{P} c \end{array} $
$ \begin{array}{l} \forall rel : r \bullet rel.s \in c \wedge rel.p \in c \wedge object\ rel.t \in c \wedge rel.p \in a \\ \forall conc : c; g : \textit{GROUP} \bullet (c, is_a, c, g) \notin r \\ \\ \forall s : c; d : \mathbb{P} c \bullet \textit{childOf} s = \{d : c \mid (s, is_a, d, zero_group) \in r\} \\ \forall d : c; s : \mathbb{P} c \bullet \textit{parentOf} d = \{s : c \mid (s, is_a, d, zero_group) \in r\} \\ \\ \textit{descs} = \textit{childOf}^+ \\ \textit{ancs} = \textit{parentOf}^+ \\ a = \textit{descs linkage_concept} \\ \\ \text{dom } rs \subseteq c \end{array} $

3 Result

The result of applying a query against a substrate is either a (possibly empty) set of SCTID's or an *ERROR*.

$$\begin{aligned}
\textit{ERROR} &::= \textit{unknownOperation} \mid \textit{unknownConceptReference} \mid \\
&\quad \textit{unknownPredicate} \mid \textit{unknownRefsetId} \\
\textit{Result} &::= ok\langle\langle \mathbb{P} \textit{SCTID} \rangle\rangle \mid error\langle\langle \textit{ERROR} \rangle\rangle
\end{aligned}$$

4 Interpretation of Intermediate Constructs

This section carries the interpretation of the intermediate constructs – the various forms of expressions and their combinations.

4.1 expressionConstraint

$$\begin{aligned}
&\text{expressionConstraint} = \\
&(\text{refinedExpressionConstraint} / \text{unrefinedExpressionConstraint})
\end{aligned}$$

4.1.1 Interpretation

The interpretation of `expressionConstraint` the interpretation of the `refinedExpressionConstraint` or the `unrefinedExpressionConstraint`.

expressionConstraint ::=
 ecrec⟨⟨*refinedExpressionConstraint*⟩⟩ |
 ecurec⟨⟨*unrefinedExpressionConstraint*⟩⟩

<i>i_expressionConstraint</i> : <i>Substrate</i> → <i>expressionConstraint</i> → <i>Result</i>
$\forall ss : \text{Substrate}; ec : \text{expressionConstraint} \bullet i_expressionConstraint\ ss\ ec =$ if <i>ecrec</i> ~ <i>ec</i> ∈ <i>refinedExpressionConstraint</i> then <i>i_refinedExpressionConstraint</i> <i>ss</i> (<i>ecrec</i> ~ <i>ec</i>) else if <i>ecurec</i> ~ <i>ec</i> ∈ <i>unrefinedExpressionConstraint</i> then <i>i_unrefinedExpressionConstraint</i> <i>ss</i> (<i>ecurec</i> ~ <i>ec</i>) else <i>error unknownOperation</i>

4.1.2 unrefinedExpressionConstraint

An `unrefinedExpressionConstraint` is either a `compoundExpressionConstraint` or a `simpleExpressionConstraint`

`unrefinedExpressionConstraint` =
 `compoundExpressionConstraint` / `simpleExpressionConstraint`

unrefinedExpressionConstraint ::=
 ucec⟨⟨*compoundExpressionConstraint*⟩⟩ |
 usec⟨⟨*simpleExpressionConstraint*⟩⟩

<i>i_unrefinedExpressionConstraint</i> : <i>Substrate</i> → <i>unrefinedExpressionConstraint</i> → <i>Result</i>
$\forall ss : \text{Substrate}; uec : \text{unrefinedExpressionConstraint} \bullet$ <i>i_unrefinedExpressionConstraint</i> <i>ss</i> <i>uec</i> = if <i>ucec</i> ~ <i>uec</i> ∈ <i>compoundExpressionConstraint</i> then <i>i_compoundExpressionConstraint</i> <i>ss</i> (<i>ucec</i> ~ <i>uec</i>) else if <i>usec</i> ~ <i>uec</i> ∈ <i>simpleExpressionConstraint</i> then <i>i_simpleExpressionConstraint</i> <i>ss</i> (<i>usec</i> ~ <i>uec</i>) else <i>error unknownOperation</i>

4.1.3 refinedExpressionConstraint

```
refinedExpressionConstraint =
  unrefinedExpressionConstraint ":" refinement /
  "(" refinedExpressionConstraint ")"
```

The interpretation of `refinedExpressionConstraint` is the intersection of the interpretation of the `unrefinedExpressionConstraint` and the `refinement`, both of which return a set of SCTID's or an error.

The interpretation of the second option adds no value.

$$\text{refinedExpressionConstraint} == \text{unrefinedExpressionConstraint} \times \text{refinement}$$

$i_refinedExpressionConstraint :$ $Substrate \rightarrow refinedExpressionConstraint \rightarrow Result$
$\forall ss : Substrate; rec : refinedExpressionConstraint \bullet$ $i_refinedExpressionConstraint ss rec =$ $intersect (i_unrefinedExpressionConstraint ss (first rec)) (i_refinement ss (second rec))$

4.2 simpleExpressionConstraint

The interpretation of `simpleExpressionConstraint` is the application of an optional constraint operator to the interpretation of `focusConcept`, which returns a set of SCTID's or an error. The interpretation of an error is the error.

`exclusionIndicator` is ignored on the assumption that it is to be represented as the added binary difference operator.

```
simpleExpressionConstraint =
  [exclusionIndicator] [constraintOperator ] focusConcept
```

$$\begin{aligned} \text{constraintOperator} &::= \text{descendantOrSelfOf} \mid \text{descendantOf} \mid \\ &\quad \text{ancestorOrSelfOf} \mid \text{ancestorOf} \\ \text{simpleExpressionConstraint} &== \text{constraintOperator}[0..1] \times \text{focusConcept} \end{aligned}$$

$i_simpleExpressionConstraint :$ $Substrate \rightarrow simpleExpressionConstraint \rightarrow Result$
$\forall ss : Substrate; sec : simpleExpressionConstraint \bullet$ $i_simpleExpressionConstraint ss sec =$ $i_constraintOperator ss (first sec) (i_focusConcept ss (second sec))$

4.3 compoundExpressionConstraint

compoundExpressionConstraint is interpreted as a sequence of *simpleExpressionConstraint* interpretations joined by compound operators. The sequence is interpreted from left to right

```
compoundExpressionConstraint = simpleExpressionConstraint
  1*( compoundOperator simpleExpressionConstraint) /
  "(" compoundExpressionConstraint ")"
```

```
compoundOperator ::= conjunction | disjunction | difference
compoundEntry  $\hat{=}$  [op : compoundOperator; cons : simpleExpressionConstraint]
compoundExpressionConstraint == simpleExpressionConstraint  $\times$ 
  seq(compoundEntry)
```

$i_compoundExpressionConstraint :$ $Substrate \rightarrow compoundExpressionConstraint \rightarrow Result$
$\forall ss : Substrate; cec : compoundExpressionConstraint \bullet$ $i_compoundExpressionConstraint ss cec =$ $nextCEC ss (i_simpleExpressionConstraint ss (first cec)) (second cec)$

$nextCEC :$ $Substrate \rightarrow Result \rightarrow$ $seq compoundEntry \rightarrow Result$
$evalCEC :$ $Substrate \rightarrow Result \rightarrow compoundOperator \rightarrow$ $simpleExpressionConstraint \rightarrow Result$
$\forall ss : Substrate; lhs : Result; e : seq compoundEntry \bullet$ $nextCEC ss lhs e =$ if $e = \langle \rangle$ then lhs else $nextCEC ss (evalCEC ss lhs (head e).op (head e).cons) (tail e)$ $\forall ss : Substrate; lhs : Result; op : compoundOperator; rhs : simpleExpressionConstraint \bullet$ $evalCEC ss lhs op rhs =$ if $op = conjunction$ then $intersect lhs (i_simpleExpressionConstraint ss rhs)$ else if $op = disjunction$ then $union lhs (i_simpleExpressionConstraint ss rhs)$ else if $op = difference$ then $minus lhs (i_simpleExpressionConstraint ss rhs)$ else $error unknownOperation$

4.4 refinement

`refinement` is the interpretation of one or more `attribute` or `attributeGroups` joined by `compoundOperators`

```
compoundOperator = conjunction / disjunction / minus
refinement = (attribute / attributeGroup / "(" refinement ")")
             [compoundOperator refinement]
```

```
refTarget ::= att⟨attribute⟩ | attg⟨attributeGroup⟩
refinement == refTarget × seq(compoundOperator × refTarget)
```

```
i_refTarget : Substrate → refTarget → Result
i_refinement : Substrate → refinement → Result
```

```
∀ ss : Substrate; rt : refTarget • i_refTarget ss rt =
  if att~rt ∈ attribute
  then i_attribute ss (att~rt)
  else if attg~rt ∈ attributeGroup
  then i_attributeGroup ss (attg~rt)
  else error unknownOperation

∀ ss : Substrate; ref : refinement •
  i_refinement ss ref =
    evalRefinement ss (i_refTarget ss (first ref)) (second ref)
```

```
evalRefinement :
  Substrate → Result → seq(compoundOperator × refTarget) → Result
```

```
∀ ss : Substrate; r : Result; opt : seq(compoundOperator × refTarget) •
  evalRefinement ss r opt =
    if opt = ⟨⟩
    then r
    else if first(head opt) = conjunction
    then evalRefinement ss (intersect r(i_refTarget' ss (second(head opt)))) (tail opt)
    else if first(head opt) = disjunction
    then evalRefinement ss (union r(i_refTarget' ss (second(head opt)))) (tail opt)
    else if first(head opt) = difference
    then evalRefinement ss (minus r(i_refTarget' ss (second(head opt)))) (tail opt)
    else error unknownOperation
```

```
i_refTarget' : Substrate → refTarget → Result
```

4.5 expressionConstraintValue

```
expressionConstraintValue = simpleExpressionConstraint /
    "(" (refinedExpressionConstraint /
        compoundExpressionConstraint) ")"

expressionConstraintValue ::= ecvsec⟨⟨simpleExpressionConstraint⟩⟩ |
    ecvrec⟨⟨refinedExpressionConstraint'⟩⟩ |
    ecvcec⟨⟨compoundExpressionConstraint⟩⟩
[refinedExpressionConstraint']
```

$i_expressionConstraintValue : Substrate \rightarrow$ $expressionConstraintValue \rightarrow Result$
$\forall ss : Substrate; ecv : expressionConstraintValue \bullet$ $i_expressionConstraintValue ss ecv =$ if $ecvsec \sim ecv \in simpleExpressionConstraint$ then $i_simpleExpressionConstraint ss (ecvsec \sim ecv)$ else if $ecvrec \sim ecv \in refinedExpressionConstraint'$ then $i_refinedExpressionConstraint' ss (ecvrec \sim ecv)$ else if $ecvcec \sim ecv \in compoundExpressionConstraint$ then $i_compoundExpressionConstraint ss (ecvcec \sim ecv)$ else $error\ unknownOperation$
$i_refinedExpressionConstraint' :$ $Substrate \rightarrow refinedExpressionConstraint' \rightarrow Result$

4.6 Attributes and Attribute Groups

The interpretation of an attribute. **attributeOperator** and **attributeName** determines the set of possible attributes in the substrate relationship table. **reverseFlag** and **expressionConstraintValue** determine the set of candidate targets (if **reverseFlag** is absent) or **subjects** (if **reverseFlag** is present).

cardinality determines the minimum and maximum matches. In all cases, only a subset of the subjects (targets if **reverseFlag** is present) in the substrate relationship table will be returned in the interpretation.

```
attribute = [cardinality] [reverseFlag] [attributeOperator] attributeName
    (comparisonOperator concreteValue / "=" expressionConstraintValue )
attributeGroup = "{" attributeSet "}"
attributeSet = attribute [compoundOperator attributeSet] / "(" attributeSet ")"
cardinality = "[" nonNegativeIntegerValue to (nonNegativeIntegerValue / many) "]"
constraintOperator = descendantOrSelfOf / descendantOf / ancestorOrSelfOf / ancestorOf
attributeOperator = descendantOrSelfOf / descendantOf
comparisonOperator = "=" / "!=" ws "=" /
    ("n"/"N") ("o"/"O") ("t"/"T") ws "=" / "<=" / "<" / ">=" / ">"
```

4.7 AttributeName and AttributeSubject

i_attributeName verifies that *conceptReference* is a valid concept in the substrate and interprets it as a set of (exactly one) *ATTRIBUTE*

i_attributeSubject interprets the *constraintOperator*, if any in the context of the attribute name and interprets it as a set of *ATTRIBUTE*s

Question: Should we add any additional validation checks?

attributeName == *conceptReference*

attributeOperator == {*descendantOrSelfOf*, *descendantOf*}

attributeSubject == *attributeOperator*[0..1] × *attributeName*

$i_attributeName : Substrate \rightarrow attributeName \rightarrow Result$ $i_attributeSubject : Substrate \rightarrow attributeSubject \rightarrow Result$
$\forall ss : Substrate; an : attributeName \bullet i_attributeName\ ss\ an = i_conceptReference\ ss\ an$
$\forall ss : Substrate; as : attributeSubject \bullet i_attributeSubject\ ss\ as = i_constraintOperator\ ss\ (first\ as)\ (i_attributeName\ ss\ (second\ as))$

4.8 Attribute

attribute consists of an optional **cardinality** and an **attributeConstraint**.

unlimitedNat ::= *num*⟨⟨ℕ⟩⟩ | *many*

cardinality == ℕ × *unlimitedNat*

$attribute$
$card : cardinality[0..1]$ $attw : attributeConstraint$

attributeConstraint is either an attribute expression constraint or a concrete value constraint.

attributeConstraint ::= *aec*⟨⟨*attributeExpressionConstraint*⟩⟩ | *acvc*⟨⟨*attributeConcreteValueConstraint*⟩⟩

attributeExpressionEonstraint is a combination of a subject constraint (target if reverse flag is true) and an expression constraint value whose interpretation yields a set of SCTID's that filter the target (subject if reverse flag).

[*reverseFlag*]

attributeExpressionConstraint _____

a : *attributeSubject*

rf : *reverseFlag*[0 .. 1]

cs : *expressionConstraintValue*

A concrete value constraint is the combination of a subject constraint and a comparison operator/concrete value whose interpretation yields a set of subject ids. The intersection of the subject and concrete value interpretation is the interpretation of **attributeConcreteValueConstraint**

comparisonOperator ::= *eq* | *neq* | *gt* | *ge* | *lt* | *le*

attributeConcreteValueConstraint _____

a : *attributeSubject*

op : *comparisonOperator*

v : *concreteValue*

The interpretation of an attribute is the interpretation of the cardinality applied against the underlying attribute constraint, producing a set of SCTID's or an error condition

The interpretation of an attribute constraint is the interpretation of either the attribute expression constraint or the concrete expression constraint that produces a set of Quads or an error condition.

The actual interpretations if attribute expression and concrete value are in Section ??

$i_attribute :$ $Substrate \rightarrow attribute \rightarrow Result$ $i_attributeConstraint :$ $Substrate \rightarrow attributeConstraint \rightarrow Quads$ $i_attributeExpressionConstraint :$ $Substrate \rightarrow attributeExpressionConstraint \rightarrow Quads$ $i_attributeConcreteValueConstraint :$ $Substrate \rightarrow attributeConcreteValueConstraint \rightarrow Quads$
$\forall ss : Substrate; a : attribute \bullet$ $i_attribute ss a = i_cardinality a.card (i_attributeConstraint ss a.attw)$ $\forall ss : Substrate; aw : attributeConstraint \bullet i_attributeConstraint ss aw =$ $\text{if } aec \sim aw \in attributeExpressionConstraint$ $\quad \text{then } i_attributeExpressionConstraint ss (aec \sim aw)$ $\text{else if } acvc \sim aw \in attributeConcreteValueConstraint$ $\quad \text{then } i_attributeConcreteValueConstraint ss (acvc \sim aw)$ $\text{else } qerror\ unknownOperation$ $\forall ss : Substrate; aec : attributeExpressionConstraint; eca : expressionConstraintArgs \mid$ $eca.atts = i_attributeSubject ss aec.p \wedge$ $eca.rf = aec.rf \wedge$ $eca.subjOrTarg = i_expressionConstraintValue ss aec.cs \bullet$ $i_attributeExpressionConstraint ss aec = i_attributeExpression ss eca$ $\forall ss : Substrate; awc : attributeConcreteValueConstraint; aca : concreteConstraintArgs \mid$ $aca.atts = i_attributeSubject ss awc.p \wedge$ $aca.op = awc.op \wedge$ $aca.t = awc.v \bullet$ $i_attributeConcreteValueConstraint ss awc = i_concreteAttributeConstraint ss aca$

4.9 AttributeSet and AttributeGroup

An `attributeGroup` is an `attributeSet`. An `attributeSet` consists of a sequence of one or more attribute constraints joined by `compoundOperators`.

$attributeGroup == attributeSet$
 $attributeSet == attribute \times seq(compoundOperator \times attribute)$

$i_attributeGroup : Substrate \rightarrow attributeGroup \rightarrow Result$
$i_attributeSet : Substrate \rightarrow attributeSet \rightarrow Result$
$\forall ss : Substrate; ag : attributeGroup \bullet i_attributeGroup ss ag =$ $i_attributeSet ss ag$
$\forall ss : Substrate; a : attributeSet \bullet i_attributeSet ss a =$ $result (evalCmpndAtt ss (i_groupedAttribute ss (first a)) (second a))$

4.10 Compound attribute evaluation

The left-to-right evaluation of **attributeSet**. The interpretation takes a *lhs* as a function from *SCTID* to *GROUP* and a *rhs* which is sequence of operator/attribute tuples and recursively interprets the *lhs* and interpretation of the head of the *rhs*.

$ \begin{aligned} & \text{evalCmpndAtt} : \\ & \quad \text{Substrate} \rightarrow \text{IDGroups} \rightarrow \text{seq}(\text{compoundOperator} \times \text{attribute}) \rightarrow \text{IDGroups} \\ & \text{gintersect}, \text{gunion}, \text{gminus}, \text{gfirstError} : \\ & \quad \text{IDGroups} \rightarrow \text{IDGroups} \rightarrow \text{IDGroups} \end{aligned} $
$ \begin{aligned} & \forall ss : \text{Substrate}; lhs : \text{IDGroups}; rhs : \text{seq}(\text{compoundOperator} \times \text{attribute}) \bullet \\ & \text{evalCmpndAtt } ss \text{ lhs } rhs = \\ & \text{if } rhs = \langle \rangle \\ & \quad \text{then } lhs \\ & \text{else if } \text{first}(\text{head } rhs) = \text{conjunction} \\ & \quad \text{then } \text{evalCmpndAtt } ss (\text{gintersect } lhs (\text{i_groupedAttribute } ss (\text{second}(\text{head } rhs)))) (\text{tail } rhs) \\ & \text{else if } \text{first}(\text{head } rhs) = \text{disjunction} \\ & \quad \text{then } \text{evalCmpndAtt } ss (\text{gunion } lhs (\text{i_groupedAttribute } ss (\text{second}(\text{head } rhs)))) (\text{tail } rhs) \\ & \text{else if } \text{first}(\text{head } rhs) = \text{difference} \\ & \quad \text{then } \text{evalCmpndAtt } ss (\text{gminus } lhs (\text{i_groupedAttribute } ss (\text{second}(\text{head } rhs)))) (\text{tail } rhs) \\ & \quad \text{else } \text{gerror unknownOperation} \\ & \forall a, b, r : \text{IDGroups} \mid \\ & \quad r = \text{if } \text{gerror} \sim a \in \text{ERROR} \vee \text{gerror} \sim b \in \text{ERROR} \text{ then } \text{gfirstError } a \text{ } b \\ & \quad \text{else } gv (gv \sim a \cap gv \sim b) \bullet \\ & \quad \text{gintersect } a \text{ } b = r \\ & \forall a, b, r : \text{IDGroups} \mid \\ & \quad r = \text{if } \text{gerror} \sim a \in \text{ERROR} \vee \text{gerror} \sim b \in \text{ERROR} \text{ then } \text{gfirstError } a \text{ } b \\ & \quad \text{else } gv (gv \sim a \cup gv \sim b) \bullet \\ & \quad \text{gunion } a \text{ } b = r \\ & \forall a, b, r : \text{IDGroups} \mid \\ & \quad r = \text{if } \text{gerror} \sim a \in \text{ERROR} \vee \text{gerror} \sim b \in \text{ERROR} \text{ then } \text{gfirstError } a \text{ } b \\ & \quad \text{else } gv (gv \sim a \setminus gv \sim b) \bullet \\ & \quad \text{gminus } a \text{ } b = r \end{aligned} $
$ \begin{aligned} & \text{i_groupedAttribute} : \\ & \quad \text{Substrate} \rightarrow \text{attribute} \rightarrow \text{IDGroups} \end{aligned} $
$ \begin{aligned} & \forall ss : \text{Substrate}; a : \text{attribute} \bullet \\ & \quad \text{i_groupedAttribute } ss \text{ } a = \text{i_groupCardinality} (\text{i_attributeConstraint } ss \text{ } a.\text{attw}) \text{ } a.\text{card} \end{aligned} $

4.11 Group Cardinality

The interpretation of cardinality within a group impose additional constraints:

- $[0..n]$ – the set of all substrate concept codes that have at least one group (entry) in the substrate relationships and, at most n matching entries in the same group

- $[0..0]$ – the set of all substrate concept codes that have at least one group (entry) in the substrate relationships and *no* matching entries
- $[1..*]$ – (default) at least one matching entry in the substrate relationships
- $[m_1..n_1]op[m_2..n_2]...$ – set of substrate concept codes where there exists at least one group where all conditions are simultaneously true

The interpretation of a grouped cardinality is a function from a set of SCTID's to the groups in which they were qualified.

The algorithm below partitions the input set of Quads by group and validates the cardinality on a per-group basis. Groups that pass are returned

TODO: the *gresult* function seems to express what is described below more simply

$i_groupCardinality :$ $Quads \rightarrow cardinality[0..1] \rightarrow IDGroups$
$\forall quads : Quads; oc : cardinality[0..1]; uniqueGroups : \mathbb{P} GROUP;$ $quadsByGroup : GROUP \rightarrow \mathbb{P} Quad \mid$ $uniqueGroups = \{q : qids\ quads \bullet q.g\} \wedge$ $quadsByGroup = \{g : uniqueGroups; q : \mathbb{P} Quad \mid$ $q = \{e : qids\ quads \mid e.g = g\} \bullet g \mapsto (evalCardinality\ oc\ q)\} \bullet$ $i_groupCardinality\ quads\ oc =$ $gv \{sctid : SCTID; groups : \mathbb{P} GROUP \mid sctid \in \{q : \bigcup(\text{ran } quadsByGroup)\} \bullet$ $\text{if } qidd\ quads = subjects\ \text{then } q.s\ \text{else } q.t\} \wedge$ $groups = \{g : \text{dom } quadsByGroup \mid (\exists q : quadsByGroup\ g \bullet$ $sctid = \text{if } qidd\ quads = subjects\ \text{then } q.s\ \text{else } q.t)\} \bullet$ $sctid \mapsto groups\}$

4.12 Cardinality

Interpretation: *cardinality* is tested against a set of quads with the following rules:

1. Errors are propagated
2. No cardinality or passing cardinality returns the subjects / targets of the set of quads
3. Otherwise return an empty set

$i_cardinality :$ $cardinality[0..1] \rightarrow Quads \rightarrow Result$
$\forall oc : cardinality[0..1]; qr : Quads \bullet$ $i_cardinality\ oc\ qr =$ $\text{if } qerror \sim qr \in ERROR\ \text{then } result\ (gresult\ qr)$ $\text{else } result\ (gresult\ (qv\ (evalCardinality\ oc\ (qids\ qr),\ qidd\ qr)))$

evalCardinality Evaluate the cardinality of a set, returning the set if it meets the constraints, otherwise return the empty set.

$evalCardinality : cardinality[0 \dots 1] \rightarrow \mathbb{P} Quad \rightarrow \mathbb{P} Quad$
$\forall oc : cardinality[0 \dots 1]; s : \mathbb{P} Quad \bullet evalCardinality oc s =$ if $\#oc = 0$ then s else if $(\#s \geq first(head\ oc)) \wedge$ $(second(head\ oc) = many \vee num^\sim(second(head\ oc)) \leq \#s)$ then s else \emptyset

5 Substrate Interpretations

This section defines the interpretations that are realized against the substrate.

5.1 AttributeExpressionConstraint

expressionConstraintArgs carries the arguments necessary to evaluate an attribute expression

- **rf** – If present, return subjects matching attribute/target subjects
- **subjOrTarg** – Set of subject or target SCTIDS (or error)
- **atts** – Set of attributes to evaluate

$expressionConstraintArgs$
$rf : reverseFlag[0 \dots 1]$ $subjOrTarg : Result$ $atts : Result$

Evaluate an attribute expression, returning the set of quads in the substrate relationship table with the supplied subject / attribute if *rf* is absent and attribute / target if *rf* is present. The interpretation returns an error there is an error in either the subject/targets or attributes.

$i_attributeExpression :$ $Substrate \rightarrow expressionConstraintArgs \rightarrow Quads$
$\forall ss : Substrate; args : expressionConstraintArgs;$ $sti : \mathbb{P} TARGET; atts : \mathbb{P} ATTRIBUTE \mid$ $sti = v \sim args.subjectOrTarg \wedge atts = v \sim args.attrs \bullet$ $i_attributeExpression ss args =$ if $error \sim (bigunion\{args.subjectOrTarg, args.attrs\}) \in ERROR$ then $qfirstError\{args.subjectOrTarg, args.attrs\}$ else if $\neg (sti \cup atts) \subseteq ss.c$ then $qerror\ unknownConceptReference$ else if $\#args.rf = 0$ then $qv(\{s : SUBJECT; t : sti; a : atts; g : GROUP; re : ss.r \mid$ $re.t = t \wedge re.p = a \bullet re\}, subjects)$ else $qv(\{s : sti; a : atts; t : TARGET; g : GROUP; re : ss.r \mid$ $re.s = t \wedge re.p = a \bullet re\}, targets)$

5.2 ConcreteAttributeConstraint

```

concreteValue = QM stringValue QM / "#" numericValue
stringValue = 1*(anyNonEscapedChar / escapedChar)
numericValue = decimalValue / integerValue
integerValue = ( ["-"/"+"] digitNonZero *digit ) / zero

```

The mechanism for interpreting *concreteValue* is not fully specified at this point. Much of the rest of the machinery is focused around interpreting constraints in the context of quads - subject, attribute, target and group, so the function is currently defined as returning a *Quads*, but another type or interpretation may be in order.

$concreteValue ::= stringValue \mid integerValue \mid realValue$

concreteConstraintArgs The arguments to the concrete value function.

- **atts** – list of attributes to test
- **op** – operator
- **t** – value to test against

Note: *reverseFlag* has no meaning for concrete constraints.

$concreteConstraintArgs$
$atts : Result$ $op : comparisonOperator$ $t : concreteValue$

```

i_concreteAttributeConstraint :
  Substrate → concreteConstraintArgs → Quads

```

5.3 ConstraintOperator

Issue: What is the behavior on a transitive closure that has a cycle? At the moment we always remove the focus concept on *descendantsOf* and *ancestorsOf*

Interpretation: Apply the substrate forward (*descs*) or reverse (*ancs*) functions to a set of SCTID's in the supplied *Result*. Error conditions are propagated.

```

i_constraintOperator :
  Substrate ↔ constraintOperator[0..1] ↔ Result ↔ Result

completeFun : (SCTID ↔  $\mathbb{P}$  SCTID) → SCTID →  $\mathbb{P}$  SCTID

∀ ss : Substrate; oco : constraintOperator[0..1]; rs : Result •
i_constraintOperator ss oco rs =
  if error ~ rs ∈ ERROR ∨ #oco = 0
  then rs
  else if head oco = descendantOrSelfOf
    then v(∪{id : ids rs • completeFun ss.descs id ∪ {id}})
  else if head oco = descendantOf
    then v(∪{id : ids rs • completeFun ss.descs id \ {id}})
  else if head oco = ancestorOrSelfOf
    then v(∪{id : ids rs • completeFun ss.ancs id ∪ {id}})
  else if head oco = ancestorOf
    then v(∪{id : ids rs • completeFun ss.ancs id \ {id}})
  else error unknownOperation

∀ f : (SCTID ↔  $\mathbb{P}$  SCTID); id : SCTID • completeFun f id =
  if id ∈ dom f then f id else ∅

```

5.4 FocusConcept

`focusConcept = conceptReference / (memberOf focusConcept)`

Issue: The recursive `memberOf` operator should be replaced with two options:

- One deep – return all the direct descendants of the focus concept
- Transitive Closure – return the transitive closure of the recursive evaluation of the concept

Interpretation:

1. *conceptReference* – *i_conceptReference* *conceptReference*

2. *memberOf conceptReference* – apply the substrate *vs* function if the SCTID of *conceptReference* is in the domain of the function, otherwise return an error.
3. *closure conceptReference* – apply the substrate *tcvs* function if the SCTID of *conceptReference* is in the domain of the function, otherwise return an error

valueSetOptions ::= *memberOf* | *closure*
focusConcept $\hat{=}$ [*opts* : *valueSetOptions*[0 .. 1]; *cr* : *conceptReference*]

<i>i_focusConcept</i> : <i>Substrate</i> \rightarrow <i>focusConcept</i> \rightarrow <i>Result</i>
$\forall ss : \text{Substrate}; fc : \text{focusConcept} \bullet$ <i>i_focusConcept</i> <i>ss</i> <i>fc</i> = if # <i>fc.opts</i> = 0 then <i>i_conceptReference</i> <i>ss</i> <i>fc.cr</i> else (let <i>sctid</i> == (<i>toSCTID</i> <i>fc.cr</i>) \bullet if <i>sctid</i> \notin dom <i>ss.vs</i> then <i>error unknownConceptReference</i> else if head <i>fc.opts</i> = <i>memberOf</i> then <i>v</i> (<i>ss.vs</i> <i>sctid</i>) else <i>v</i> (<i>ss.tcvs</i> <i>sctid</i>))

5.5 ConceptReference

conceptReference = *conceptId* ["|" *Term* "|"]
conceptId = *sctId*

Interpretation: *conceptReference* is interpreted as SCTID if it is in the list of concepts in the substrate, otherwise as an error.

[*conceptReference*]

<i>toSCTID</i> : <i>conceptReference</i> \rightarrow <i>SCTID</i>
<i>i_conceptReference</i> : <i>Substrate</i> \rightarrow <i>conceptReference</i> \rightarrow <i>Result</i>
$\forall ss : \text{Substrate}; c : \text{conceptReference} \bullet i_conceptReference\ ss\ c =$ if (<i>toSCTID</i> <i>c</i>) \in <i>ss.c</i> then <i>v</i> {(<i>toSCTID</i> <i>c</i>)} else <i>error unknownConceptReference</i>

6 Helper Functions

- *Quads* is either a collection of *Quads* or an error condition. If *Quads*, it also carries a direction indicator that determines whether it represents a set of subjects or targets.
- *IDGroups* is a map from SCTID's to their passing groups or an error condition

$$\begin{aligned}
direction &::= subjects \mid targets \\
Quads &::= qv\langle\mathbb{P} Quad \times direction\rangle \mid qerror\langle ERROR\rangle \\
IDGroups &::= gv\langle SCTID \rightarrow \mathbb{P} GROUP\rangle \mid gerror\langle ERROR\rangle
\end{aligned}$$

Results functions

- *ids* – return the set of sctids in a result or the empty set if there is an error
- *quads* – return the set of quads in a quads result or an empty set if there is an error
- *quidd* – return the direction of a quads result. Undefined if error
- *quids* – return the sctid to group map in an id group or an empty map if there is error
- *gresult* – convert a set of quads into a set of id groups
- *result* – convert a set of id groups into a simple result.

$ids : Result \rightarrow \mathbb{P} SCTID$ $qids : Quads \rightarrow \mathbb{P} Quad$ $qidd : Quads \rightarrow direction$ $gids : IDGroups \rightarrow SCTID \rightarrow \mathbb{P} GROUP$ $gresult : Quads \rightarrow IDGroups$ $result : IDGroups \rightarrow Result$
$\forall r : Result \bullet ids\ r =$ if $error \sim r \in ERROR$ then \emptyset else $v \sim r$ $\forall q : Quads \bullet qids\ q =$ if $qerror \sim q \in ERROR$ then \emptyset else $first(qv \sim q)$ $\forall q : Quads \bullet qidd\ q =$ $second(qv \sim q)$ $\forall g : IDGroups \bullet gids\ g =$ if $gerror \sim g \in ERROR$ then \emptyset else $gv \sim g$ $\forall q : Quads \bullet gresult\ q =$ if $qerror \sim q \in ERROR$ then $gerror(qerror \sim q)$ else if $qidd\ q = subjects$ then $gv\ \{s : SCTID \mid (\exists qr : qids\ q \bullet s = qr.s) \bullet$ $s \mapsto \{qr : qids\ q \bullet qr.g\}\}$ else $gv\ \{t : SCTID \mid (\exists qr : qids\ q \bullet t = qr.t) \bullet$ $t \mapsto \{qr : qids\ q \bullet qr.g\}\}$ $\forall g : IDGroups \bullet result\ g =$ if $gerror \sim g \in ERROR$ then $error(gerror \sim g)$ else $v(dom(gv \sim g))$

Definition of the various functions that are performed on the result type.
 $firstError$ is not fully defined – it takes a set of results with one or more errors
and returns an aggregation of all of the errors.

$union, intersect, minus : Result \rightarrow Result \rightarrow Result$ $bigunion, bigintersect : \mathbb{P} Result \rightarrow Result$ $firstError : \mathbb{P} Result \rightarrow Result$ $qfirstError : \mathbb{P} Result \rightarrow Quads$
$\forall x, y : Result \bullet union\ x\ y =$ if $error \sim x \in ERROR$ then x else if $error \sim y \in ERROR$ then y else $v((v \sim x) \cup (v \sim y))$ $\forall x, y : Result \bullet intersect\ x\ y =$ if $error \sim x \in ERROR$ then x else if $error \sim y \in ERROR$ then y else $v((v \sim x) \cap (v \sim y))$ $\forall x, y : Result \bullet minus\ x\ y =$ if $error \sim x \in ERROR$ then x else if $error \sim y \in ERROR$ then y else $v((v \sim x) \setminus (v \sim y))$ $\forall rs : \mathbb{P} Result \bullet bigunion\ rs =$ if $\exists r : rs \bullet error \sim r \in ERROR$ then $firstError\ rs$ else $v(\bigcup \{r : rs \bullet ids\ r\})$ $\forall rs : \mathbb{P} Result \bullet bigintersect\ rs =$ if $\exists r : rs \bullet error \sim r \in ERROR$ then $firstError\ rs$ else $v(\bigcap \{r : rs \bullet ids\ r\})$

Representing optional elements of type T . Representing it as a sequence allows us to determine absence by $\#T = 0$ and the value by $head\ T$.

$$T[0..1] == \{s : seq\ T \mid \#s \leq 1\}$$