The AURA KB Translations - FOPL, TPTP, ASP, SILK, and OWL2

Michael Wessel

March 29, 2013

Introduction

We describe the translations of the AURA Biology KB ("Bio KB 101"). The native format of the KB is KM syntax. We describe how this KB is translated into 5 different KR formats, using a common description framework. At the time of this writing, we are supporting the formats SILK, ASP, FOPL, TPTP, and OWL2.

The SILK translation is joint work with Benjamin Grosof, Paul Fodor, Paul Haley, Mike Dean, and Brett Benyo.

The ASP translation is joint work with Son Tran, Vinay Chaudhri, and Stijn Heymans.

The OWL2 translation is joint work with Vinay Chaudhri.

Rather than basing our work on KM syntax, we start by giving an axiomatic description of the KB content in terms of first-order axiom patterns. A so-called *underspecified KB* is an idealized weaker version of the AURA KB in which co-references between implied existentials (modeled as Skolem function successors) are left implicit. By adding equality axioms to an underspecified KB, we gain a so-called *fully specified KB,* making co-references between implied existentials explicit.

The utility of a fully specified KB is that it captures so-called unifications, and that we inherited content can be factored out from concept axioms, based on a syntactic criterion - a notion of "local" or "asserted" vs. "inherited" atoms in a concept axiom describing necessary conditions.

The fully specified KB is computed from the original AURA KB by means of an algorithm which hypothesizes equality atoms between existentials or, equivalently, Skolem function successors – this is basically a form of abduction, or default reasoning. The details of the algorithm are not important here.

Axiomatic Description of the KB - Underspecified KB

We are using first-order logic with equality. We are using standard syntax, but sometimes make use of the comma to denote conjunctions more succinctly.

An underspecified KB is a set of first-order axioms of certain kinds. Axioms are grouped according to type, e.g., all taxonomic axioms for a concept, all disjointness axioms for a concept, etc. Axioms of a certain type give rise to so-called axiom patterns. For example, all disjointness axioms have the same form and hence give rise to an axiom pattern.

Let us describe the vocabulary (the signature) of the KB first.

Let *CN* be a countably infinite set of *concept names* (e.g., *Cell*), and RN be a countably infinite set of *relation names* (e.g., *has-part*), and let $AN \subseteq RN$ be a countably infinite set of *attribute names* (e.g., *color*, *temperature*). The naming schema for concept names uses hyphens and uppercase characters, e.g., *Animal-Plasma-Membrane.* Relations have lowercase names and hyphens, e.g., *has-part.*

In the following, we are using $C, C_1, C_2, \ldots\ D, D_1, D_2, \ldots, E, E_1, E_2, \ldots, F, F_1, F_2, \ldots$ to denote concepts, and $R, R_1, R_2, \ldots, S, S_1, S_2, \ldots, T, T_1, T_2, \ldots$ to denote relations. *String* denotes a string, e.g. "Cell", and *float* a floating point number (e.g., 22d0).

Let $\{x, y, z, x_1, x_2, \ldots\}$ be a countably infinite set of first-order variables, and, for every $C \in CN$, let $\{ fn_C\#1, fn_C\#2, \ldots \}$ be a countably infinite set of (Skolem) function symbols.

There are three kinds of attributes:

- Cardinal attribute values.
  For example, "t is 43 years" would be represented as

  $age(t, t_1)$, *the-cardinal-value(t_1,43), cardinal-unit-class(t_1,year).*

- Categorial attribute values.
  For example, "t has color green" would be represented as

  color(t, t_1), the-categorical-value(t_1,green).

- Scalar attribute values.
  For example, "*t* is big w.r.t. a house" (where house is a concept) would be represented as

  size(t, t_1), the-scalar-value(t_1,big), scalar-unit-class(t_1,house).

  The so-called *value atoms* (the-cardinal-value, cardinal-unit-class, the-categorical-value, the-scalar-value, scalar-unit-class) will be explained below.

We have the following sets of constants:

- the set of scalar constant values: *SCs* = { *small, big,* … }.
- the set of categorical constant values: *CCs* = { *blue, green*, …}.
- the set of cardinal unit classes: *CUCs* = { *meter, year*, … }.
- in addition, the symbols in *CN* and *RN* are constants as well.

Note that in the original KM KB, those constants are prefixed with a "*", which we are removing here.

We now describe the axiomatic content of the KB as follows.

An *underspecified AURA KB* is a tuple (*CTAs, CAs, RAs*), where *CTAs* is a set of constant type assertions, *RAs* is a set of relation axioms, and *CAs* is a set of concept axioms. Those axioms are described in the following:

- (CTAs) The KB contains, for every $x \in SCs \cup CCs \cup CUCs$, 1 to n type assertions of the form *C(x)*, where $C \in CN$ (the types of the constant).

- (CAs) For every concept name $C \in CN$, it may contain the following kinds of axioms, described by their patterns as follows:

  - (DAs) disjointness axioms:
    $\forall\, x : C(x) \rightarrow \neg\, D(X)$.

  - (DOAs) A set of documentation assertions:
    userdescription(C,string), description(C,string), originalname(C,string).

    These assertions are on the class level and hence need not be associated with every instance of *C*.

  - (LEXAs) A set of lexicon information assertions:
    concept2words(C,string).

    These assertions are on the class level, also.

  - (TAs) taxonomic axioms:
    $\forall\, x : C(x) \rightarrow E(x)$.

  - (NCAs) necessary conditions:
    $\forall\, x : C(x) \rightarrow\ \Phi[x]$,

    where Φ[x] is a conjunction of *unary (concept) atoms and binary (relation) atoms* over terms $\{ x, fn_C\#1(x), fn_C\#2(x), \ldots \}$ - those terms are also called *nodes*, having a free variable *x.*

    There are two special equality relations, namely *equal* and *not-equal*,

which are user asserted equality atoms. The intended semantics is the semantics of first-order equality resp. in-equality. In order to distinguish them from the equalities in *EQAs*, see below, we are using different predicate names here (*equal, not-equal*).

Moreover, $\Phi[x]$ can contain the following *value atoms*:
if *t* is a node, *float* is floating point number (in Common Lisp syntax, e.g., 43.0d0), and *scalar* $\in$ *SCs*, *categorical* $\in$ *CCs*, *cardinal-unit-class* $\in$ *CUCs*, and *scalar-unit-class* $\in$ *CN*, then the following atoms are value atoms:

- *the-cardinal-value(t, float).*
- *the-scalar-value(t, scalar).*
- *the-categorical-value(t, categorical).*
- *cardinal-unit-class(t, cardinal-unit-class).*
- *scalar-unit-class(t, scalar-unit-class).*

In addition, there are so-called *qualified number restrictions*. Due to a lack of counting quantifiers we are simply representing them on the meta-level, by means of quadrary atoms

- *maxCardinality(t, R, n, C),*
- *minCardinality(t, R, n, C),*
- *exactCardinality(t, R, n, C) =*
  maxCardinality(t, R, n, C), minCardinality(t, R, n, C),

where *n* is a non-negative integer, *C* is a concept, and *R* is a relation name. The intended semantics of those expressions if the standard (description logics) semantics, but a weaker semantics, i.e. based on constraint checking and counting and closed-world reasoning, might be given to those expressions.

Under the standard semantics, the atom *maxCardinality(t, R, n, C)* is true in an interpretation $\Im$ of the KB, $\Im \models$ *maxCardinality(t, R, n, C),* if $\left| \{ i \mid \left(t^{\Im}, i\right) \in R^{\Im}, i \in C^{\Im} \} \right| \leq n$, $\Im \models$ *minCardinality(t, R, n, C),* if $\left| \{ i \mid \left(t^{\Im}, i\right) \in R^{\Im}, i \in C^{\Im} \} \right| \geq n$, and consequently, $\Im \models$ *exactCardinality(t, R, n, C),* if $\left| \{ i \mid \left(t^{\Im}, i\right) \in R^{\Im}, i \in C^{\Im} \} \right| = n$.

- (SCAs) sufficient conditions:
  $\forall x : \Theta[x, \ldots] \rightarrow C(x), EQs(x, \ldots),$

  where $\Theta[x, \ldots]$ is a conjunction of unary, binary, value and qualified number restriction atoms over terms $\{ x, x_1, x_2, \ldots \}$, the sufficient conditions, and $EQs(x, \ldots)$is a conjunction of equality atoms of the form $t1 = t2$, where $t1 \in \{ x, x_1, x_2, \ldots \}$, and $t2 \in \{ x, fn_C\#1(x), fn_C\#2(x), \ldots \}$ (hence, *b* is a node in *C*), linking the variables in the antecedence to the Skolem functions in the

consequence of the necessary conditions, $\Phi(x)$.

Obviously, requiring the Skolem functions in the antecedence of the sufficient condition would be a too strong condition and render the sufficient condition inapplicable in many cases.

Also note that $\Theta'[x] \subseteq \Phi[x]$, where $\Theta'[x]$ is the result of substituting the variables $\Theta[x]$ with their respective Skolem terms from $EQs(x, \dots)$, $\Theta'[x] = \Theta[x]_{[t1 \Rightarrow t2 \ \in EQs(x,\dots)]}$. Hence, every sufficient condition is also necessary.

For a given concept name *C*, we refer to the corresponding axioms as *DAs(C), TAs(C)*, etc. We refer to the union of all axioms for *C* as *CAs(C).*

For a concept name *C*, there is at most 1 NCAs(C). There are no other restrictions on the number of axioms per pattern for a concept.

- (RAs) For every relation name $R \in RN$, RAs may contain the following kinds of axioms:

  - (DRAs) relation domain restrictions:
    $\forall x, y : R(x, y) \rightarrow C_1(x) \vee \dots \vee C_n(x)$

  - (RRAs) relation range restrictions:
    $\forall x, y : R(x, y) \rightarrow D_1(y) \vee \dots \vee D_m(y)$

  - (RHAs) simple relation hierarchy:
    $\forall x, y : R(x, y) \rightarrow S(x, y)$

  - (QRHAs) qualified relation hierarchy:
    $\forall x, y : R(x, y) \wedge C(x) \wedge D(y) \rightarrow S(x, y)$

  - (IRAs) inverse relation:
    $\forall x, y : R(x, y) \rightarrow S(y, x)$

  - (12NAs) 1-to-N cardinality (inverse functionality):
    $\forall x, y, z : R(x, y) \wedge R(z, y) \rightarrow x = z$

  - (N21As) N-to-1 cardinality (functionality):
    $\forall x, y, z : R(x, y) \wedge R(x, z) \rightarrow y = z$

  - (TRANSAs) simple transitive closure axioms:
    $\forall x, y, z : R(x, y) \wedge Rstar(y, z) \wedge C(x) \wedge D(y) \wedge E(z) \rightarrow Rstar(x, z)$,
    $Rstar(x, z) = R^*(x, z)$ (a named relation for the transitive closure of R)

- (GTRANSLAs) generalized transitive closure axioms (left composition):
  $\forall\, x, y, z : R(x, y) \wedge S(y, z) \wedge C(x) \wedge D(y) \wedge E(z) \rightarrow Rstar(x, z)$,
  $Rstar(x, z) = R^*(x, z)$ (a named relation for the transitive closure of R)

- (GTRANSRAs) generalized transitive closure axioms (right composition):
  $\forall\, x, y, z : R(x, y) \wedge S(y, z) \wedge C(x) \wedge D(y) \wedge E(z) \rightarrow Sstar(x, z)$,
  $Sstar(x, z) = S^*(x, z)$ (a named relation for the transitive closure of S)

We refer to the axioms for a relation *R* by *DRAs(R)* etc. We refer to the union of all axioms for R as *RAs(R).*

For a relation name *R*, there is at most one in *DRAs(R), RRAs(R),* and *IRAs(R),* as well as at most one *12NAs(R)* and *N21As(R)*. There are no other restrictions on the number of axioms per pattern for a relation *R*.

Axiomatic Description of the KB - Fully Specified KB

Let (*CTAs, CAs, RAs*) be an underspecified KB. A *fully specified KB* is a tuple (*CTAs, CAs, RAs, EQAs*), such that for a subset of concepts $C \in CN'$, $CN' \subseteq CN$:

- (EQAs) A *set* of equality atoms for C of the form $t = fn(t')$:

  $\forall\, x : C(x) \rightarrow t = fn(t'), \ldots,$

  where $t, t' \in \{ x, fn_C\#1(x), fn_C\#2(x), \ldots \}$, and $fn \in \{ fn_D\#1, fn_D\#2, \ldots \}$, with $C \neq D$, for some D (D is a concept mentioned in C, or a direct or indirect superconcept of C).

  In addition, we require that the equalities are *admissible*, see below.

  Note that the maximum nesting depth of Skolem functions is 2.

These equalities describe how inherited content from other concepts is mapped into C, and can also describe KM's unification (= merging) of nodes.

We require that the equalities cannot be cyclical, in the following sense. Such a fully specified KB is called *admissible:*

Let *terms(set)* := $\{ t \mid D(t) \in set \text{ or } R(t, t_1) \in set \text{ or } R(t_1, t) \in$ be the set of terms used in set. Moreover, for a conjunction Θ, we assume that terms(Θ) is defined as well, in the obvious way (treat the conjunction as a set of conjuncts / atoms).

For $t \in$ *terms(NCAs(C)),* we define its source terms / nodes recursively as follows:

$$sourcenodes(t,C) := \{ (fn_E\#n(x), E) \mid t = fn_E\#n(t') \in EQAs(C) \} \cup \{ sourcenodes(t',E) \mid (t',E) \in sourcenodes(t,C) \}$$

A fully specified KB is *admissible* if
for all $\boldsymbol{C \in CN, t', t \in terms(NCAs(C))}$: $\boldsymbol{(t', C) \notin sourcenodes(t, C)}$.

The admissibility criterion guarantees that, for every concept name C, the transitive closure of all equality assertions in *EQAs(C)* is irreflexive, in the sense that in C, there can be no node / term which is inherited directly or indirectly from another node / term in *C.*

Given a fully specified KB, we can determine which atoms in the necessary conditions $\Phi[x]$ for C, $\forall\, x : C(x) \rightarrow \Phi[x] \in NCAs(C)$, are *local or asserted*, and which are *inherited*:

A term / node $t \in terms(NCAs(C))$ is called a *local or asserted node* / term in *C* if there is no equality assertion in *EQAs(C)* involving t: $sourcenodes(t, C) = \emptyset$.

Otherwise, t is called *inherited*.

For an atom, those notions are defined as follows.

Let *D(t)*, $R(t_1, t_2) \in \Phi[x]$ be conjuncts in the necessary condition of *C*,
$\forall\, x : C(x) \rightarrow\ \Phi[x] \in$ *NCAs(C)*.

We then say that *D(t)* is *local to / asserted in C* if there is no
(t',E) $\in$ *sourcenodes(t,C)* such that *D(t')* $\in [x]$,
$\forall\, x : E(x) \rightarrow\ \Phi'[x] \in$ *NCAs(E)*.

Analog, $R(t_1, t_2)$ is *local to / asserted in C* if there is no pair (t_1, t_2) with
$(t_1', E) \in sourcenodes(t_1, C) \cup \{ (x, E) \}$,
$(t_2', E) \in sourcenodes(t_2, C)$, such that $R(t_1', t_2') \in$,
$\forall\, x : E(x) \rightarrow \Phi'[x] \in$ *NCAs(E)*.

Some Abstract Examples

Those notions are illustrated with the following examples. Assume that every *C* instance has an *R* successor which is a *D* instance:

$$\forall\, x : C(x) \rightarrow R\big(x, fn_C\#1(x)\big), D(fn_C\#1(x))$$

Here, *x* and $fn_C\#1(x)$ are local. Moreover, all atoms are local to / asserted in *C*.

The subclass *SubC* of C refines this as follows: the inherited *D R*-successor also has an *S*-successor. To refer to the inherited *D R*-successors, we equate the local node in *SubC* with the from *C* inherited *D R*-successor as follows:

$$\forall x : SubC(x) \rightarrow C(x), R\big(x, fn_D\#1(x)\big), D\big(fn_D\#1(x)\big), S\big(fn_D\#1(x), fn_D\#2(x)\big), \\ fn_D\#1(x) = fn_C\#1(x)$$

Note that *x*, and $fn_D\#2(x)$ are local, but $fn_D\#1(x)$ is not. Moreover, $R(x, fn_D\#1(x))$ is not local (it is inherited from *C*), as is $D(fn_D\#1(x))$. However, *C(x)* as well as $S(fN_D\#1(x), fn_D\#2(x))$ are local to *SubC*.

Nested Skolem terms occur in case inheritance does not happen directly from a superclass, as in the previous example, but from instantiation – consider the concept *E*, in which every *E* instance has an *R*-successor of type *C*, and the (from *C*) inherited *R*-successor of type *D* has an *S*-successor:

$$\forall\, x : E(x) \rightarrow R\big(x, fn_E\#1(x)\big), C\big(fn_E\#1(x)\big), R\big(fn_E\#1(x), fn_E\#2(x)\big), D\big(fn_E\#2(x)\big), \\ S\big(fn\_E\#2(x), fn\_E\#3(x)\big), fn_E\#2(x) = fn_C\#1(fn_E\#1(x))$$

Note that *x*, $fn_E\#1(x)$ and $fn_E\#3$ are local, but $fn_E\#2$ is inherited. Consequently, $R(x, fn_E\#1(x))$ and $C(fn_E\#1(x))$ as well as $S(fn_E\#2(x), fn_E\#3(x))$ are local to *E*, and the other atoms are inherited from *C*.

Another use for equalities is the representation of unification – consider the concept *F,* which has two *R*-successors, one has types D and G, and the other type H:

$$\forall\, x : F(x) \rightarrow R\big(x, fn_F\#1(x)\big), R\big(x, fn_F\#2(x)\big), D\big(fn_F\#1(x)\big), G\big(fn_F\#1(x)\big), H(fn_F\#2(x))$$

Assume that in the subclass *SubF*, those two successors have been unified by a user ("merged" or "equated") – this is represented as follows:

$$\forall\, x : SubF(x) \rightarrow F(x), R\big(x, fn_{SubF}\#1(x)\big), D\big(f_{SubF}\#1(x)\big), G\big(fn_{SubF}\#1(x)\big), H\big(fn_{SubF}\#1(x)\big), fn_{SubF}\#1(x) = fn_F\#1(x), fn_{SubF}\#1(x) = fn_F\#2(x)$$

Note that only *x* is local. *F(x)* is local to *SubF*. All other atoms are inherited from *F* and hence not local to *SubF.*

The Major Different Kinds of Exported KBs

Starting point for every exported / translated KB is a fully specified KB which may contain equality axioms (*EQAs(C)).*

There are various export options which apply to all syntaxes. The KB can be exported either as a

a. underspecified or
b. fully specified

KB. In case of a.), the equality axioms are omitted from the export. Hence, such a KB does not contain equality atoms.

Moreover, regarding the atoms in the necessary conditions, we can either export

1. all atoms, or
2. only the local / asserted atoms.

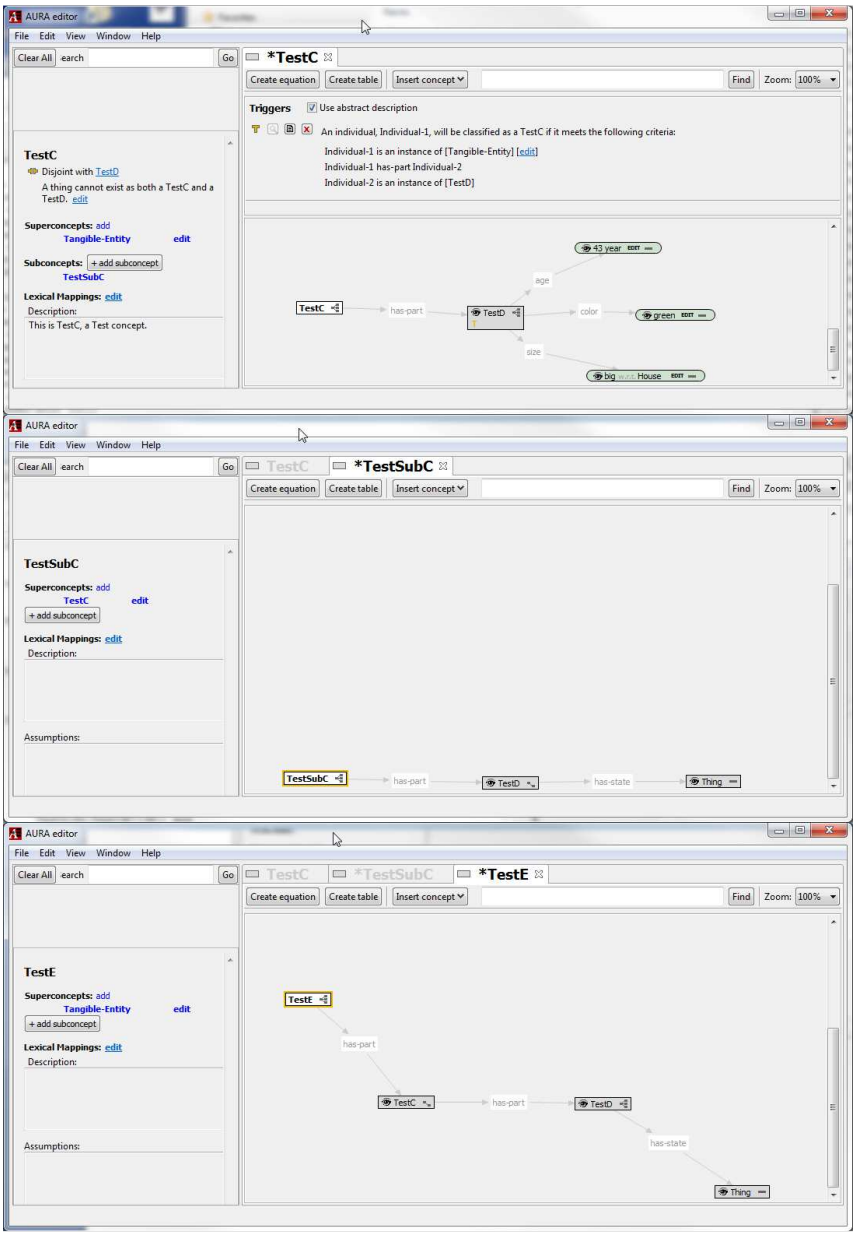
This gives rise to four combinations, a1), a2), b1) and b2). KBs which are exported using option a.) have a "-umap" in their file names, and an "all-triples" for 1.), and "asserted-only" for 2.)

We usually export the KB as one big file that contains all the renderings for the axioms *CTAs, CAs, RAs*, and *EQAs*. Moreover, with the exception of OWL2, we are also creating "one file per concept" versions of the KB. Those KBs reside in directories with appropriate, e.g. "SILK-kb-all-triples-umap" for a1) in SILK, etc. In those directories, there is one file per concept, e.g., `cell.SILK`, one file `relations.SILK` containing all relation (declaration and) axioms *RAs*, and one file `constants.SILK` containing all the constants and their types, the `CTAs.`

The OWL2 translation uses different characterizations which are described in the OWL2 Section.

An Example KB

We are refining the previously discussed examples a little bit. The concepts TestC, TestSubC, and TestE are presented. For TestC, we are adding a disjointness axiom (disjoint with TestD), and some value atoms as well as some documentation atoms and a sufficient condition. Moreover, instead of *R* we are using *has-part*, for *S* we are using *has-state*. Note that has-part is The first-order logic axiom should be clear from the following pictures, and the preceding discussion:



The FOPL Translation

The FOPL translation is the most natural rendering of the original FOPL axioms in the mathematical / abstract description of the KB. It uses a simple intuitive (but non-standard) syntax.

Rendering of axioms:

Each axiom or fact ends with a period. The universal quantifier is dropped from the axioms – it is assumed that all variables are implicitly universally quantified. The Boolean connectives are rendered as `not`, `and`, `or`. The implication arrow → is rendered as `->`.

Rendering of predicates:

The original spelling of the predicates is preserved (case and punctuation etc.) Equality is rendered as =, and inequality as `<>`. The same applies to the equalities in *EQAs*.

Rendering of terms:

Variables are prefixed with a question mark and rendered uppercase. Strings are rendered in double quotes, and \ is used as an escape character in strings, e.g., for double quotes in strings. Skolem functions $fn_C\#n$ are rendered as `fn-C#n`. The original spelling of constants is preserved.

The "KB as one big file" FOPL translations have the following structure:

1. The *CTAs* are rendered. A fact such as *Size-Constant(big)* is rendered as `Size-Constant(big).`

   Note that *Size-Constant* is a concept from the KB and hence, axioms may be rendered it as well (e.g., it is a subconcept of *Constant, etc.*)

2. Relation axioms are written. For a relation *R*, all axioms for *R, RAs(R),* are combined into a single conjunction $\Omega[x, y]$ to produce one axiom of the form
   $$\forall\, x, y : R(x, y) \rightarrow\ \Omega[x, y]$$
   which is then rendered in the obvious way.

3. Next, the axioms *DAs(C), TAs(C), EQAs(C),* as well as *DOAs(C)* and *LEXAs(C)* are combined with the necessary conditions in *NCAs(C)* to produce one axiom of the form
   $$\forall\, x : C(x) \rightarrow\ \Omega[x]$$
   which is then rendered in the obvious way. Note that $\Omega[x]$ is the conjunction of all the atoms in the consequences of the axioms in *DAs(C), TAs(C), EQAs(C),* and also consists of the atoms in the sets *DOAs(C)* and *LEXAs(C).* However, those have the form *R(C,string)* and hence, the *C* argument must be substituted with *x* first.

4. Finally, the sufficient axioms from *SCAs(C)* are rendered, in the obvious way, using the syntax described above for necessary conditions.

The Example KB in FOPL Format

An abridged version of the example KB is given as follows:

```
Color-Constant(green ).

Size-Constant(big).

has-part(?X, ?Y) ->
     Tangible-Entity(?Y).

has-part(?X, ?Y) ->
     Tangible-Entity(?X).

has-part(?X, ?Y) and has-part(?Z, ?Y) ->
     ?X=?Z.

has-part(?X, ?Y) ->
     has-structure(?X, ?Y) and
     related-to(?X, ?Y) and
     has-part-or-unit(?X, ?Y) and
     is-part-of(?Y, ?X).

has-part(?X, ?X1) and Tangible-Entity(?X) and TestD(?X1) ->
     TestC(?X) and
     fn-TestC#1(?X)=?X1.

TestC(?X) ->
     original-name(?X, "TestC") and
     user-description(?X, "This is TestC, a Test concept.") and
     concept2words(?X, "testc") and
     Tangible-Entity(?X) and
     not TestD(?X) and
     TestD(fn-TestC#1(?X)) and
     Size-Value(fn-TestC#2(?X)) and
     Color-Value(fn-TestC#3(?X)) and
     Duration-Value(fn-TestC#4(?X)) and
     the_cardinal_value(fn-TestC#4(?X), 43.0d0) and
     cardinal_unit_class(fn-TestC#4(?X), year) and
     the_categorical_value(fn-TestC#3(?X), green) and
     the_scalar_value(fn-TestC#2(?X), big) and
     scalar_unit_class(fn-TestC#2(?X), House) and
     size(fn-TestC#1(?X), fn-TestC#2(?X)) and
     color(fn-TestC#1(?X), fn-TestC#3(?X)) and
     age(fn-TestC#1(?X), fn-TestC#4(?X)) and
     maxCardinality(?X, has-part, 1, TestD) and
     has-part(?X, fn-TestC#1(?X)).

TestD(?X) ->
     original-name(?X, "TestD") and
     concept2words(?X, "testd") and
     Tangible-Entity(?X) and
     not TestC(?X).

TestE(?X) ->
     original-name(?X, "TestE") and
     concept2words(?X, "teste") and
     Tangible-Entity(?X) and
     Thing(fn-TestE#1(?X)) and
     TestC(fn-TestE#2(?X)) and
     TestD(fn-TestE#3(?X)) and
     TestD(fn-TestC#1(fn-TestE#2(?X))) and
     has-state(fn-TestE#3(?X), fn-TestE#1(?X)) and
     has-part(?X, fn-TestE#2(?X)) and
     has-part(fn-TestE#2(?X), fn-TestE#3(?X)) and
```

```
     fn-TestE#3(?X)=fn-TestC#1(fn-TestE#2(?X)).

TestSubC(?X) ->
     original-name(?X, "TestSubC") and
     concept2words(?X, "testsubc") and
     TestC(?X) and
     Thing(fn-TestSubC#1(?X)) and
     TestD(fn-TestSubC#2(?X)) and
     has-state(fn-TestSubC#2(?X), fn-TestSubC#1(?X)) and
     has-part(?X, fn-TestSubC#2(?X)) and
     fn-TestSubC#2(?X)=fn-TestC#1(?X).
```

The TPTP Translation

The TPTP project defines a standard format for first-order logic. The syntax is described in detail here:
http://www.cs.miami.edu/~tptp/TPTP/SyntaxBNF.html#fof_formula

Like the FOPL format, the FOPL axioms from the abstract KB description are rendered almost exactly in the same form.

Rendering of axioms:

Each axiom or fact *ax* ends with a period. Each fact or axiom is embedded in

`fof(`*id*`, axiom,(`
  ax `)).`

where *id* is some identifier. The universal quantifiers $\forall x : \ldots.$, $\forall x, y : \ldots.$, are rendered as `! [ X ] : ...` and `! [X, Y] : ...`. The Boolean connectives are rendered as `~`, `&`, `|`. The implication arrow → is rendered as `=>`.

Rendering of predicates:

Predicates are converted into lower case. Punctuation characters and parentheses such as `- # / ( ) .` are substituted by underscores, and `* + ` and `,` are substituted by `_star_`, `_plus_`, and `_comma_`. User-asserted equality is rendered as `equal`, and user-asserted inequality as `!=`. The equalities in *EQAs* are rendered using `=`.

Rendering of terms:

Variables are rendered uppercase. Strings are rendered in double quotes. In strings, all newline and linefeed characters are eliminated, as are the characters `* “ | \` and `^M`. The `\` is the espace character in string, e.g., for strings in strings. Skolem functions $fn_C\#n$ are rendered as `fn_C_n`. Constants are rendered in the same way as predicates. Floating point numbers are rendered using the standard scientific notation, e.g., 43.0e0 .

The KB axioms are rendered in the same way and order of sequence as described for the FOPL format. The syntax can be verified under
http://www.cs.miami.edu/~tptp/cgi-bin/SystemB4TPTP
(use “Local file to upload”).

The Example KB in TPTP Format

An abridged version of the example KB is given as follows:

```
fof(a643677,axiom,(
  color_constant(green ))).

fof(a644003,axiom,(
  size_constant(big))).

fof(a644696,axiom,(
```

```
  ! [X, Y] :
    ( ( has_part(X, Y) )
     =>
      ( tangible_entity(Y) ) ))).

fof(a644697,axiom,(
  ! [X, Y] :
    ( ( has_part(X, Y) )
     =>
      ( tangible_entity(X) ) ))).

fof(a644698,axiom,(
  ( ( has_part(X, Y)
    & has_part(Z, Y) )
   =>
    ( X=Z ) ))).

fof(a644699,axiom,(
  ! [X, Y] :
    ( ( has_part(X, Y) )
     =>
      ( has_structure(X, Y)
        & related_to(X, Y)
        & has_part_or_unit(X, Y)
        & is_part_of(Y, X) ) ))).

fof(a646210,axiom,(
  ( ( has_part(X, X1)
    & tangible_entity(X)
    & testd(X1) )
   =>
    ( testc(X)
      & fn_testc_1(X)=X1 ) ))).

fof(a646211,axiom,(
  ! [X] :
    ( ( testc(X) )
     =>
      ( original_name(X, "TestC")
        & user_description(X, "This is TestC, a Test concept.")
        & concept2words(X, "testc")
        & tangible_entity(X)
        & ~ testd(X)
        & testd(fn_testc_1(X))
        & size_value(fn_testc_2(X))
        & color_value(fn_testc_3(X))
        & duration_value(fn_testc_4(X))
        & the_cardinal_value(fn_testc_4(X), 43.0e0)
        & cardinal_unit_class(fn_testc_4(X), year)
        & the_categorical_value(fn_testc_3(X), green)
        & the_scalar_value(fn_testc_2(X), big)
        & scalar_unit_class(fn_testc_2(X), house)
        & size(fn_testc_1(X), fn_testc_2(X))
        & color(fn_testc_1(X), fn_testc_3(X))
        & age(fn_testc_1(X), fn_testc_4(X))
        & maxCardinality(X, has_part, 1, testd)
        & has_part(X, fn_testc_1(X)) ) ))).

fof(a646212,axiom,(
  ! [X] :
    ( ( testd(X) )
     =>
      ( original_name(X, "TestD")
        & concept2words(X, "testd")
        & tangible_entity(X)
        & ~ testc(X) ) ))).

fof(a646213,axiom,(
  ! [X] :
    ( ( teste(X) )
     =>
```

```
      ( original_name(X, "TestE")
        & concept2words(X, "teste")
        & tangible_entity(X)
        & thing(fn_teste_1(X))
        & testc(fn_teste_2(X))
        & testd(fn_teste_3(X))
        & testd(fn_testc_1(fn_teste_2(X)))
        & has_state(fn_teste_3(X), fn_teste_1(X))
        & has_part(X, fn_teste_2(X))
        & has_part(fn_teste_2(X), fn_teste_3(X))
        & fn_teste_3(X)=fn_testc_1(fn_teste_2(X)) ) ))).

fof(a646218,axiom,(
  ! [X] :
    ( ( testsubc(X) )
     =>
      ( original_name(X, "TestSubC")
        & concept2words(X, "testsubc")
        & testc(X)
        & thing(fn_testsubc_1(X))
        & testd(fn_testsubc_2(X))
        & has_state(fn_testsubc_2(X), fn_testsubc_1(X))
        & has_part(X, fn_testsubc_2(X))
        & fn_testsubc_2(X)=fn_testc_1(X) ) ))).
```

The ASP Translation

This format is suitable for consumption by a standard Answer Set Programming (ASP) reasoner.

Rendering of axioms:

Each implication axiom which has more than one conjunct in its consequence is split into multiple implication axioms $\forall\, x : C(x) \rightarrow\ \Phi[x]$ with $\Phi[x] = (\Phi_1, \ldots, \Phi_n)\ [x]$, then the axiom is replaced with a set of axioms $\forall\, x : C(x) \rightarrow\ \Phi_1, \ldots, \forall\, x : C(x) \rightarrow\ \Phi_n$ (note that x may or may not be a free variable in $\Phi_1, \ldots, \Phi_n$). Concept atoms such as *C(x)* are rendered using the `instance-of` predicate: `instance_of(X,c).` Relation atoms such as $R(x, fn_C\#1(x))$ are rendered using the `relation_value` predicate: `relation_value(R,X,fn-c-1(X)).` The direction of the implication arrow is reversed, e.g. $\forall\, x : C(x) \rightarrow\ \Phi_n$ is turned into $\forall\, x : \Phi_n \leftarrow C(x)$, and $\leftarrow$ is reversed and rendered as `:-` as it is standard in logic programming. Moreover, the universal quantifier is dropped. Facts are rendered in the standard way. Every fact and axiom is terminated with a period. For reasons to be explained below, there are no negated atoms in the rules (disjointness information is expressed on a non-axiomatic / factual level).

Rendering of predicates:

Predicates are converted into lower case. Punctuation characters and parentheses such as `- # / ( ) .` are substituted by underscores, and `* +` and `,` are substituted by `_star_`, `_plus_`, and `_comma_`. Moreover, predicate names are not allowed to start with a digit – in case of a leading digit, the prefix `number_` is added to the name. User-asserted equality is rendered as `eq_user`, and user-asserted inequality as `neq_user`. The equalities in *EQAs* are rendered using `eq`.

Rendering of terms:

Variables are rendered uppercase, without leading question mark. Strings are rendered in double quotes. In strings, all newline and linefeed characters are eliminated, as are the characters `* “ | \` and `^M`. The `\` is the espace character in string, e.g., for strings in strings. Skolem functions $fn_C\#n$ are rendered as `fn_c_n`. Constants are rendered in the same way as predicates. Floating point numbers are rendered as strings.

The “KB as one big file” ASP translations have the following structure:

1. Concept declarations are written as facts; but note that we also export the first-order axioms for the concepts (see below).

   For every $C \in CN$, let C' be the ASP rendered class name, e.g., if
   C = Animal-Plasma-Membrane, then
   `C' = animal_plasma_membrane,`
   and the following ASP facts are written for `C'`:

`class(C');`
A concept (class) declaration.

`subclass_of(C',D');`
for every $\forall x$: $C(x) \rightarrow D(X) \in TAs(C)$

`disjoint(C',D');`
for every $\forall\, x : C(x) \rightarrow \neg\, D(X) \in DAs(C)$

`R(C',string);`
for every *R(C,string)* $\in$ *DOAs(C),* where
$R \in$ { *userdescription, description, originalname }*

`concept2words(C',string);`
for every *concept2words(C,string)* $\in LEXAs(C)$

2. The *CTAs* are rendered. A fact such as *Size-Constant(big)* is rendered as `instance_of(big,size_constant).`

   Note that *Size-Constant* is a concept from the KB and hence, axioms may be rendered it as well (e.g., it is a subconcept of *Constant, etc.*)

3. Relation declarations and axioms are written.

   For every relation $R \in RN$, let R' be the ASP rendered relation name, e.g. if $R =$ *has-part-or-unit*, then
   `R'= has_part_or_unit.`
   The following ASP facts are written for `R'`:

   `relation(R');`
   A relation declaration.

   `domain(R',C');`
   for every $\forall\, x, y : R(x, y) \rightarrow C(x) \in RDAs(R)$

   In case of a disjunctive domains, a superconcept representing the disjunction is introduced, as ASPs cannot deal with disjunctions in heads.

   For example, for $\forall\, x, y : R(x, y) \rightarrow C(x) \vee D(x) \in RDAs(R)$, we create a fresh concept `c-approxor-d` representing the disjunction, as well as appropriate subclass declarations: `subclass_of(c,c-approxor-d), subclass_of(d,c-approxor-d).` The domain restriction becomes `domain(R',c-approxor-d).`

   `range(R',C');`
   for every $\forall\, x, y : R(x, y) \rightarrow D(y) \in RRAs(R)$

Disjunctive ranges are handled analog to disjunctive domains.

`subrelation_of(R',S');`
for every $\forall\, x, y : R(x,y) \rightarrow S(x,y) \in RHAs(R)$

`inverse(R',S');`
for every $\forall\, x, y : R(x,y) \rightarrow S(y,x) \in IRAs(R)$

`cardinality(R',"1-to-N");`
if *12NAs(R)* $\neq \emptyset$

`cardinality(R',"N-to-1");`
if *N21As(R)* $\neq \emptyset$

`transfer(R',S',T') ;`
if $x, y, z : R(x,y) \wedge S(y,z) \wedge C(x) \wedge D(y) \wedge E(z) \rightarrow T(x,z)$
$\in\ TRANSAs(R) \cup GTRANSLAs(R) \cup GTRANSRAs(R)$, and *C(x), D(y), E(z)* are already logically implied by the domain and range restrictions of those relations.

Note that relation axioms are **not** rendered on a first order-level.

4. Next, the necessary conditions *NCAs(C)* are combined with the equality atoms in *EQAs(C)* to produce one big axiom of the form

$$\forall\, x : C(x) \rightarrow\ \Omega[x]\ \wedge\ \Xi[x]$$

which is then rendered as described above under "rendering of axioms". Each atomic conjunct in the consequence of the implication gives rise to one ASP rule; i.e., the conjunction is broken up. Note that $\Omega[x]$ is the conjunction of *EQAs(C)* and *NCAs(C),* and $\Xi(x) = \bigwedge_{t\ \in terms(\Omega[x])}\ has_{root(t,x)}$. The axioms *TAs(C)* and *DAs(C)* are not included here, as they are only rendered on the factual / assertional level (see above). The *has_root* atoms connect every node / term to the root variable *x*, which can be helpful for reasoning purposes. They are rendered as `relation_value(has_root,fn_...,X).`

5. Finally, the sufficient axioms from *SCAs(C),* $\forall\, x :\ \Theta[x, \ldots] \rightarrow C(x)$, $EQs(x, \ldots)$ are rendered. Note that the conjunction in the consequence of the implication is again broken up into atomic consequences, and hence one ASP rule per atomic conjunct in the consequence is produced. The antecedence of the implication, $\Theta[x, \ldots]$, is directly rendered as a conjunction in those rules (complex ASP rule body).

The Example KB in ASP Format

```
class(testc).

original_name(testc,"TestC").

subclass_of(testc,tangible_entity).

disjoint(testc,testd).

user_description(testc,"This is TestC, a Test concept.").

concept2words(testc,"testc").

class(testd).

original_name(testd,"TestD").

subclass_of(testd,tangible_entity).

disjoint(testd,testc).

concept2words(testd,"testd").

class(teste).

original_name(teste,"TestE").

subclass_of(teste,tangible_entity).

concept2words(teste,"teste").

class(testsubc).

original_name(testsubc,"TestSubC").

subclass_of(testsubc,testc).

concept2words(testsubc,"testsubc").


instance_of(green ,color_constant).

instance_of(big,size_constant).


relation(has_part).

cardinality(has_part,"1-to-N").

subrelation_of(has_part,has_structure).

subrelation_of(has_part,related_to).

subrelation_of(has_part,has_part_or_unit).

inverse(has_part,is_part_of).

domain(has_part,tangible_entity).

range(has_part,tangible_entity).


instance_of(X,testc) :- relation_value(has_part,X,X1), instance_of(X,tangible_entity),
instance_of(X1,testd).

eq(fn_testc_1(X),X1) :- relation_value(has_part,X,X1), instance_of(X,tangible_entity),
instance_of(X1,testd).

instance_of(fn_testc_1(X),testd) :- instance_of(X,testc).
```

```
instance_of(fn_testc_2(X),size_value) :- instance_of(X,testc).
instance_of(fn_testc_3(X),color_value) :- instance_of(X,testc).
instance_of(fn_testc_4(X),duration_value) :- instance_of(X,testc).
relation_value(the_cardinal_value,fn_testc_4(X),"43.0d0") :- instance_of(X,testc).
relation_value(cardinal_unit_class,fn_testc_4(X),year) :- instance_of(X,testc).
relation_value(the_categorical_value,fn_testc_3(X),green) :- instance_of(X,testc).
relation_value(the_scalar_value,fn_testc_2(X),big) :- instance_of(X,testc).
relation_value(scalar_unit_class,fn_testc_2(X),house) :- instance_of(X,testc).
relation_value(size,fn_testc_1(X),fn_testc_2(X)) :- instance_of(X,testc).
relation_value(color,fn_testc_1(X),fn_testc_3(X)) :- instance_of(X,testc).
relation_value(age,fn_testc_1(X),fn_testc_4(X)) :- instance_of(X,testc).
maxCardinality(testc,X,has_part,1,testd) :- instance_of(X,testc).
relation_value(has_part,X,fn_testc_1(X)) :- instance_of(X,testc).
relation_value(has_root,fn_testc_2(X),X) :- instance_of(X,testc).
relation_value(has_root,fn_testc_3(X),X) :- instance_of(X,testc).
relation_value(has_root,fn_testc_4(X),X) :- instance_of(X,testc).
relation_value(has_root,X,X) :- instance_of(X,testc).
relation_value(has_root,fn_testc_1(X),X) :- instance_of(X,testc).
instance_of(fn_teste_1(X),thing) :- instance_of(X,teste).
instance_of(fn_teste_2(X),testc) :- instance_of(X,teste).
instance_of(fn_teste_3(X),testd) :- instance_of(X,teste).
instance_of(fn_testc_1(fn_teste_2(X)),testd) :- instance_of(X,teste).
relation_value(has_state,fn_teste_3(X),fn_teste_1(X)) :- instance_of(X,teste).
relation_value(has_part,X,fn_teste_2(X)) :- instance_of(X,teste).
relation_value(has_part,fn_teste_2(X),fn_teste_3(X)) :- instance_of(X,teste).
relation_value(has_root,fn_teste_1(X),X) :- instance_of(X,teste).
relation_value(has_root,X,X) :- instance_of(X,teste).
relation_value(has_root,fn_teste_2(X),X) :- instance_of(X,teste).
relation_value(has_root,fn_teste_3(X),X) :- instance_of(X,teste).
eq(fn_teste_3(X),fn_testc_1(fn_teste_2(X))) :- instance_of(X,teste).
instance_of(fn_testsubc_1(X),thing) :- instance_of(X,testsubc).
instance_of(fn_testsubc_2(X),testd) :- instance_of(X,testsubc).
relation_value(has_state,fn_testsubc_2(X),fn_testsubc_1(X)) :- instance_of(X,testsubc).
relation_value(has_part,X,fn_testsubc_2(X)) :- instance_of(X,testsubc).
relation_value(has_root,fn_testsubc_1(X),X) :- instance_of(X,testsubc).
```

```
relation_value(has_root,X,X) :- instance_of(X,testsubc).
relation_value(has_root,fn_testsubc_2(X),X) :- instance_of(X,testsubc).
eq(fn_testsubc_2(X),fn_testc_1(X)) :- instance_of(X,testsubc).
```

The SILK Translation

SILK is an advanced next-generation rule language.

Rendering of axioms:

Unlike ASP, SILK can deal with complex conjunctions in the implication consequences, so there is no need to split up the axioms into single line implications. The implication arrow is `:-`. No F-Logic syntax is used. The universal quantifier is dropped. Facts are rendered in the standard way. Every fact and axiom is terminated with a semicolon. In SILK, we are exporting axioms on the factual / assertional level, as well as on the first-order axiomatic level. Hence, this format blends the ASP style export with the axiom-oriented first-order export.

Rendering of predicates:

1. They are brought into lowercase.
2. Parentheses are replaced by underscores.
3. Punctuation characters, such as `\/''_@!,.;` are replaced with hyphens.
4. + is substituted with `-plus`, and * is substituted with `–star.`
5. All occurences of `is` are substituted with `'is'`.
6. # is replaced with ___ (two underscores).
7. Hyphenated names are rendered as structured, nested terms, e.g., `Animal-Plasma-Membrane` becomes `animal(plasma(membrane)).`
8. Every function / constant in the structured term is prefixed with `a:` or `aop:` The latter is used for the operators / predicates `class, subclass, disjoint, domain, range, transfer, relation, subrelation, inverse, transitive, functional, cardinality, minCardinality, maxCardinality, exactCardinality, eq, neq, description, userdescription, usercomment, concept2word,` and `originalname` in order to distinguish them from KB predicates (they have a special semantics). Otherwise, `a:` is used. Hence, `Animal-Plasma-Membrane` becomes `a:animal(a:plasma(a:membrane)).`

Rendering of terms:

1. Variables are prefixed with a question mark, and are lowercase.
2. Strings cannot contain unicode characters, line breaks or returns. Those characters are removed from the strings.
3. Constants are rendered in the same way as predicates.
4. Floating point numbers are rendered as strings.
5. Skolem functions such as fn_C#1 are rendered as nested terms, quoted, and prefixed with the `a:`, e.g. `a:'fn(C)(1)'`.

The "KB as one big file" SILK translations have the following structure:

1. The two namespace prefix declarations

   ```
   :- prefix a = <http://vulcan.com/2012/aura#> ;
   :- prefix aop = <http://vulcan.com/2012/aura-operators#> ;
   ```

   are written.

2. The *CTAs* are rendered. A fact such as *size-constant(big)* is rendered as a SILK fact `a:size(a:constant)(a:big) ;`

   Note that *size-constant* is a concept from the KB and hence, axioms may be rendered it as well (e.g., it is a subconcept of *constant, etc.*)

3. Concept declarations are written as facts; but note that we also export the first-order axioms for the concepts (see below).

   For every $C \in CN$, let C' be the SILK rendered class name, e.g., if
   C = *Animal-Plasma-Membrane*, then
   `C' = a:animal(a:plasma(a:membrane))`,
   and the following SILK facts are written for `C'`:

   `aop:class(C');`
   A concept (class) declaration.

   `aop:subclass(C',D');`
   for every $\forall x: C(x) \rightarrow D(X) \in TAs(C)$

   `aop:disjoint(C',D');`
   for every $\forall\, x : C(x) \rightarrow \neg\, D(X) \in DAs(C)$

   `aop:R(C',string);`
   for every $R(C,string) \in DOAs(C)$, where
   $R \in \{$ *userdescription, description, originalname* $\}$

   `aop:concept2words(C',string);`
   for every *concept2words(C,string)* $\in LEXAs(C)$

4. Relation declarations and axioms are written.

   For every relation $R \in RN$, let R' be the SILK rendered relation name, e.g. if
   R = *has-part-or-unit*, then
   `R'= a:has(a:part(a:or(a:unit))).`
   The following SILK facts are written for `R'`:

   `aop:relation(R');`
   A relation declaration.

`aop:domain(R',C');`
for every $\forall\, x, y : R(x, y) \rightarrow C(x) \in RDAs(R)$

Note: disjunctive domains are currently ignored here, but are rendered on the first-order level (see below).

`aop:range(R',C');`
for every $\forall\, x, y : R(x, y) \rightarrow D(y) \in RRAs(R)$

Note: disjunctive ranges are currently ignored here, but are rendered on the first-order level (see below).

`aop:subrelation(R',S');`
for every $\forall\, x, y : R(x, y) \rightarrow S(x, y) \in RHAs(R)$

`aop:inverse(R',S');`
for every $\forall\, x, y : R(x, y) \rightarrow S(y, x) \in IRAs(R)$

`aop:cardinality(R',"1-to-N");`
if $12NAs(R) \neq \emptyset$

`aop:cardinality(R',"N-to-1");`
if $N21As(R) \neq \emptyset$

`aop:transfer(R',S',T') ;`
if $x, y, z : R(x, y) \wedge S(y, z) \wedge C(x) \wedge D(y) \wedge E(z) \rightarrow T(x, z)$
$\in\ TRANSAs(R) \cup GTRANSLAs(R) \cup GTRANSRAs(R)$, and $C(x)$, $D(y)$, $E(z)$ are already logically implied by the domain and range restrictions of those relations.

Now, relation axioms are also rendered on the first order-level. For example, the axiom

$\forall\, x, y, z : element(x, y) \wedge element^*(y, z) \wedge aggregate(x) \wedge thing(y) \wedge thing(z) \rightarrow element^*(x, z) \in TRANSAs(R)$

is rendered as a simple SILK rule in the obvious way:

```
a:element(a:star)(?x, ?z)
     :- a:element(?x, ?y), a:element(a:star)(?y, ?z),
        a:aggregate(?x), a:thing(?y), a:thing(?z);
```

All relation axioms are rendered as SILK rules in the obvious way. Possible disjunctions in *DRAs(R)* and *RRAs(R)* are problematic though, as SILK cannot represent disjunctions in rule bodies / consequences. Disjunction is hence represented on the meta-level, using the `aop:disjunction` atom. For example, the disjunctive range restriction

$$\forall\, x, y : timeContains(x, y) \rightarrow \\ Situation(y) \vee Event(y) \in RRAs(timeContains)$$

is rendered as

```
aop:disjunction(a:situation(?y), a:event(?y))
     :- a:time(a:contains)(?x, ?y);
```

5. Next, the axioms *DAs(C), TAs(C),* and *EQAs(C)* are combined with the necessary conditions in *NCAs(C)* to produce one axiom of the form
$$\forall\, x : C(x) \rightarrow\ \Omega[x]\ \wedge\ \Xi[x]$$
which is then rendered in SILK in the obvious way. Note that $\Omega[x]$ is the conjunction of all the atoms in the consequences of the axioms in *DAs(C), TAs(C), NCA(s)* and *EQAs(C),* and $\Xi(x) = \bigwedge_{t\, \in\, terms(\Omega[x])}\ has\_root(t, x)$. The *has_root* atoms connect every term to the root variable *x*, which can be helpful for reasoning purposes. They are rendered as `a:has(a:root)`….

   Negated atoms (stemming from *DAs(C)*) are rendered using SILK's `neg` operator. The comma denotes conjunction. The equality and inequality atoms in *EQAs(C)* are rendered as `aop:eq` and `aop:neq`, respectively. Note that `aop:neq` is not the `neg` operator, as it resides in a different namespace.

   Qualified number restriction atoms are rendered using `aop:minCardinality`, `aop:maxCardinality`, `aop:exactCardinality`.

6. Finally, the sufficient axioms from *SCAs(C)* are rendered, in the obvious way, using the syntax described above for necessary conditions.

The Example KB in SILK Format

```
:- prefix a = <http://vulcan.com/2012/aura#> ;
:- prefix aop = <http://vulcan.com/2012/aura-operators#> ;

a:color(a:constant)(a:green) ;

a:size(a:constant)(a:big) ;


aop:class(a:testc) ;

aop:originalname(a:testc, "TestC") ;

aop:subclass(a:testc, a:tangible(a:entity)) ;

aop:disjoint(a:testc, a:testd) ;

aop:userdescription(a:testc, "This is TestC, a Test concept.") ;

aop:concept2words(a:testc, "testc") ;

aop:class(a:testd) ;

aop:originalname(a:testd, "TestD") ;
```

```
aop:subclass(a:testd, a:tangible(a:entity)) ;

aop:disjoint(a:testd, a:testc) ;

aop:concept2words(a:testd, "testd") ;

aop:class(a:teste) ;

aop:originalname(a:teste, "TestE") ;

aop:subclass(a:teste, a:tangible(a:entity)) ;

aop:concept2words(a:teste, "teste") ;

aop:class(a:testsubc) ;

aop:originalname(a:testsubc, "TestSubC") ;

aop:subclass(a:testsubc, a:testc) ;

aop:concept2words(a:testsubc, "testsubc") ;


aop:relation(a:has(a:part)) ;

aop:cardinality(a:has(a:part), "1-to-N") ;

aop:subrelation(a:has(a:part), a:has(a:structure)) ;

aop:subrelation(a:has(a:part), a:related(a:to)) ;

aop:subrelation(a:has(a:part), a:has(a:part(a:or(a:unit)))) ;

aop:inverse(a:has(a:part), a:'is'(a:part(a:of))) ;

aop:domain(a:has(a:part), a:tangible(a:entity)) ;

aop:range(a:has(a:part), a:tangible(a:entity)) ;


a:tangible(a:entity)(?y)
     :- a:has(a:part)(?x, ?y) ;

a:tangible(a:entity)(?x)
     :- a:has(a:part)(?x, ?y) ;

a:'is'(a:part(a:of))(?y, ?x)
     :- a:has(a:part)(?x, ?y) ;

a:has(a:structure)(?x, ?y),
a:related(a:to)(?x, ?y),
a:has(a:part(a:or(a:unit)))(?x, ?y)
     :- a:has(a:part)(?x, ?y) ;

aop:eq(?x, ?z)
     :- a:has(a:part)(?x, ?y), a:has(a:part)(?z, ?y) ;


a:testc(?x),
aop:eq(a:'fn(testc)(1)'(?X), ?x1)
     :- a:has(a:part)(?x, ?x1), a:tangible(a:entity)(?x), a:testd(?x1) ;

a:tangible(a:entity)(?x),
neg (a:testd(?x)),
a:testd(a:'fn(testc)(1)'(?x)),
a:size(a:value)(a:'fn(testc)(2)'(?x)),
a:color(a:value)(a:'fn(testc)(3)'(?x)),
a:duration(a:value)(a:'fn(testc)(4)'(?x)),
a:the(a:cardinal(a:value))(a:'fn(testc)(4)'(?x), "43.0d0"),
a:cardinal(a:unit(a:class))(a:'fn(testc)(4)'(?x), a:year),
```

```
a:the(a:categorical(a:value))(a:'fn(testc)(3)'(?x), a:green),
a:the(a:scalar(a:value))(a:'fn(testc)(2)'(?x), a:big),
a:scalar(a:unit(a:class))(a:'fn(testc)(2)'(?x), a:house),
a:size(a:'fn(testc)(1)'(?x), a:'fn(testc)(2)'(?x)),
a:color(a:'fn(testc)(1)'(?x), a:'fn(testc)(3)'(?x)),
a:age(a:'fn(testc)(1)'(?x), a:'fn(testc)(4)'(?x)),
aop:maxCardinality(?x, a:has(a:part), 1, a:testd),
a:has(a:part)(?x, a:'fn(testc)(1)'(?x)),
a:has(a:root)(a:'fn(testc)(2)'(?x), ?x),
a:has(a:root)(a:'fn(testc)(3)'(?x), ?x),
a:has(a:root)(a:'fn(testc)(4)'(?x), ?x),
a:has(a:root)(?x, ?x),
a:has(a:root)(a:'fn(testc)(1)'(?x), ?x)
     :- a:testc(?x) ;

a:tangible(a:entity)(?x),
neg (a:testc(?x))
     :- a:testd(?x) ;

a:tangible(a:entity)(?x),
a:thing(a:'fn(teste)(1)'(?x)),
a:testc(a:'fn(teste)(2)'(?x)),
a:testd(a:'fn(teste)(3)'(?x)),
a:testd(a:'fn(testc)(1)'(a:'fn(teste)(2)'(?x))),
a:has(a:state)(a:'fn(teste)(3)'(?x), a:'fn(teste)(1)'(?x)),
a:has(a:part)(?x, a:'fn(teste)(2)'(?x)),
a:has(a:part)(a:'fn(teste)(2)'(?x), a:'fn(teste)(3)'(?x)),
a:has(a:root)(a:'fn(teste)(1)'(?x), ?x),
a:has(a:root)(?x, ?x),
a:has(a:root)(a:'fn(teste)(2)'(?x), ?x),
a:has(a:root)(a:'fn(teste)(3)'(?x), ?x),
aop:eq(a:'fn(teste)(3)'(?x), a:'fn(testc)(1)'(a:'fn(teste)(2)'(?x)))
     :- a:teste(?x) ;

a:testc(?x),
a:thing(a:'fn(testsubc)(1)'(?x)),
a:testd(a:'fn(testsubc)(2)'(?x)),
a:has(a:state)(a:'fn(testsubc)(2)'(?x), a:'fn(testsubc)(1)'(?x)),
a:has(a:part)(?x, a:'fn(testsubc)(2)'(?x)),
a:has(a:root)(a:'fn(testsubc)(1)'(?x), ?x),
a:has(a:root)(?x, ?x),
a:has(a:root)(a:'fn(testsubc)(2)'(?x), ?x),
aop:eq(a:'fn(testsubc)(2)'(?x), a:'fn(testc)(1)'(?x))
     :- a:testsubc(?x) ;
```

The OWL2 Translation

OWL2 – the Web Ontology Language, 2nd Edition - has become the de-facto standard for reasoning on and in the Semantic Web. The translator produces OWL2 knowledge bases in the functional syntax, which is the syntax used in the main OWL2 specification document: http://www.w3.org/TR/owl2-syntax/. The OWL2 functional syntax has good human readability and is readily processed by most OWL2 reasoners. Moreover, tools such as Protégé 4.2 or RacerPro can be used to translate OWL2 functional syntax into one of the other major OWL2 syntaxes, e.g., OWL2 RDF XML or OWL2 XML. The generated KBs have been tested with Protégé 4.2 plus Fact++ as well as with RacerPro.

The OWL2 exports are quite different from the so-far discussed translations. For example, we don't generate one-file-per-concept versions here. A more notable difference is that OWL2 is a variable-free language, and hence, the graph structures expressed in the necessary and sufficient conditions of the concepts cannot be represented truthfully in all cases. The original graph structures have to be approximated. To do so, we rewrite and export the KB in two flavors:

1. We *unravel* the graph structures up to a certain maximal depth *n*. *Unraveling* is a standard technique from modal logics which is not explained here in detail. It results in an approximation of the original KB which gets the better the larger the value of *n* is.

   The filenames of KBs which were produced using unraveling start with `kb-owl-syntax-unraveled-depth-`*n*`.` By default, we are varying *n* from 0 to 4 and produce all those KBs. With depth 0, the axioms from *NCAs* and *SCAs* are getting ignored.

2. As a second option, we can represent the graph structure by introducing symbolic node identifiers as "node markers" in the OWL2 concept expressions, similar to the use of IDs in languages such as XML which can be used for graph representation. Even though the OWL2 reasoner will be "blind" to the intended semantic meaning of the node IDs, expressing graph structure and co-references of nodes, the original graph structure is at least represented and could, in principle, be exploited for reasoning by some powerful extended future OWL2 reasoner. Note that node IDs are only required in case of non-tree structures. The given example KB is tree-structured and hence, no node IDs can be observed as they are not required to truthfully represent the structure of the concepts.

   The filenames of the respective KBs start with `kb-owl-syntax-coreference-IDs.`

The discussed export options 1. and 2. Can be chosen by means of the `:max-depth` optional keyword argument to the main `render-kb / render-kb-as-`

`owl` export functions. If `nil` is passed here, then option 2. is used, and otherwise a non-negative integer is expected to specify the maximal depth *n*. Moreover, in case of option 2., node IDs can either be rendered as atomic / unstructured concepts (simple names), or introduced as *nominals* . Pass `:use-id-concepts-p nil` for the nominal representation. Note that a similar option is also available for rendering of symbolic property values, as discussed under "Rendering of terms" below.

Rendering of axioms:

The implication operator is rendered as a `SubClassOf` axiom in OWL2. Antecedences as well as consequences of the implications in the necessary and sufficient conditions can be arbitrarily complex (i.e., complex conjunctions are allowed). The Boolean connectives not, and, or, correspond to `ObjectComplementOf`, `ObjectIntersectionOf`, and `ObjectUnionOf`. The former is not needed, as negation does not occur explicitly.

Disjointness axioms *DAs* are represented by means of `DisjointClasses`. The rendering of *DAs* can be suppressed by passing `:render-disjointness-axioms-p nil`. KBs with *DAs* preserved have a `-disjointness` in their file names.

The rendering of *SCAs* (also called "triggers") can be suppressed by passing `:render-triggers-p nil`. Due to a lack of variables and equality atoms, we are omitting the *EQs(x, ...)* from $\forall x : \Theta[x, \dots] \rightarrow C(x), EQs(x, \dots) \in SCAs$. KBs with *SCAs* preserved have a `–triggers` in their file names.

Qualified cardinality restrictions are also available in OWL2 and have a syntax similar to the abstract logical syntax (see example below). The rendering of cardinality constraints in necessary conditions *NCAs* can be omitted by passing `:render-cardinality-constraints-p nil`. Also, we may choose to only export the cardinality constraints with cardinalities 0 and 1 by specifying `:render-cardinality-constraints-p :km-relevant-only` – the other cardinalities are ignored by KM and hence may not be relevant for reasoning with the KB. KBs with cardinality constraints preserved have a `–cardinalities` resp. `–km-relevant-cardinalities` in their file names.

Regarding relation axioms, *RAs*, OWL2 has expressive means for domain and range restrictions, specification of inverse relations, limited forms of role composition ("transfer through axioms"), and offers transitively closed roles, functional roles, as well as role hierarchies. The rendering of relation axioms, *RAs*, can be suppressed by specifying `:render-relation-axioms-p nil`. KBs with relation axioms retained have a `–relation-axioms` in their file names.

Moreover, so-called (class and property) annotation property axioms will be used for representing the *DOAs* and *LEXAs*.

The axioms *EQAs* are currently ignored. However, the user asserted equality and in-equality atoms are retained, and we are using the `:same-as` and `:not-equal` object properties for that purpose.

Rendering of predicates:

Unary predicates are called *classes* in OWL2. Binary relations are called *properties*. There are object as well as data properties. We will also use so-called *annotation properties* for capturing the *DOAs* and *LEXAs*. Predicate names are rendered in original case, but prefixed with a colon. Moreover, punctuation characters, underscores and parentheses in names are substituted by hyphens.

Rendering of terms:

OWL2 is a term-free language, similar to a modal logic. However, there is the anlog of first-order constants, so-called individuals, or nominals, and we may choose to use them for the representation of categorical property values (such as "green") and scalar symbolic property values (such as "big"). A categorical property value such as "green" can either be represented as a type / instance assertion of the form `ClassAssertion(:Color-Constant :green)` and then used as a nominal object property filler in concept sub-expressions such as `ObjectHasValue(:color :green)`, or `:green` might be a special subclass of `:Color-Constant`, `SubClassOf(:green :Color-Constant)`, and then used in an `ObjectSomeValuesFrom(:color :green)`expression to represent the color of some object. However, for string- and float-based property values we need to use a datatype property-based representation, and values will be XSD data literals, e.g. by means of `DataHasValue(:the-cardinal-value "43.0e0"^^xsd:float)`.

The discussed rendering options are available through the main export functions `render-kb` and `render-kb-as-owl`, via the `:use-nominals-for-constants-p` optional keyword argument. If `t` is specified, then the nominal representation is used. KBs using the nominal representation have a `–value-nominals` in their file names, and otherwise `–value-classes.` The rendering of value classes and nominals can be switched of completely by specifying `:render-value-classes-p nil.`

The OWL2 KBs are generated as follows:

1. A preamble including the namespace prefix declaration is written:
   `Prefix(:=<http://www.projecthalo.com/aura#>)`
   `Ontology(<http://www.projecthalo.com/aura>`

2. If `render-value-classes-p = t,` then the corresponding value or class assertion axioms for the nominals are written, depending on the value of `use-nominals-for-constants–p.`

3. If `use-id-concepts-p = t`, then concepts for the representation of node IDs are introduced, Node-ID-*concepts,* by means of `SubClassOf(Node-ID-concept Node-ID-Marker-Concept)` axioms. We also generate some cross-referencing information for those concepts, so that the original concept can be identified in which that node was introduced, by generating annotation assertions of the form `AnnotationAssertion(in-concept Node-ID-concept concept)`. This means that the node with that ID was introduced in this `concept`.

4. If `render-relation-axioms = t,` then we are outputting *RAs* as follows. For every relation *R,* and the corresponding `R'` OWL2 property:

    a. A number of auxiliary property, domain and range declarations are rendered, e.g., for annotation properties such as `:user-description, :description`, the `:value` datatype property, the property `:has-categorical-value`, etc. This can be considered as a preamble, and the exact details can be inspected in the exported KBs.

    b. The axioms *TRANSAs*, *GTRANSLAs*, *GTRANSRAs* are analyzed. If an axiom can be truthfully encoded as an OWL2 complex role inclusion axiom, then it is represented via a complex property chain inclusions axiom in the OWL file. If a relations R turns out to be transitive, then this is declared by means of of `TransitiveObjectProperty(R')` axiom.

    c. *RDAs(R)* are rendered as `ObjectPropertyDomain(R',C')`, for every $\forall\, x, y : R(x, y) \rightarrow C(x) \in RDAs(R)$.

    Disjunctive domains are not a problem in OWL2. `ObjectUnionOf` is used to express disjunctions.

    d. *RRAs(R)* are rendered as `ObjectPropertyRange(R',C')`, for every $\forall\, x, y : R(x, y) \rightarrow D(y) \in RRAs(R)$.

    Disjunctive ranges are not a problem in OWL2.

    e. *RHAs(R)* are rendered as `SubObjectProperty(R',S')`, for every $\forall\, x, y : R(x, y) \rightarrow S(x, y) \in RHAs(R)$.

    f. *IRAs(R)* are rendered as `InverseObjectProperties(R',S')`, for every $\forall\, x, y : R(x, y) \rightarrow S(y, x) \in IRAs(R)$.

g. If *N21As(R)* $\neq \emptyset$, then we declare `FunctionalObjectProperty(R'),` and `ObjectProperty(R')` otherwise.

h. If *R* has a user-description *string*, then this is rendered as an `AnnotationAssertion(R' string).`

5. Next, the *CAs* are processed as follows. For every *C* and respective `C':`

   a. The axioms *TAs(C)* and *NCA(C)* are combined into one axiom of the form $\forall\, x : C(x) \rightarrow \Omega[x]$, which is then rendered as a `SubClassOf(C'` Ω') axiom. Ω' is either - up to depth *n* - unraveled version of Ω as an OWL2 concept expression, or the OWL2 concept description using node IDs for representing the graph structure. Note that the *DAs(C)* and *EQAs(C)* are excluded here.

   b. For the sufficient axioms from $\forall\, x : \Theta[x, \ldots] \rightarrow C(x), EQs(x, \ldots)$, we generate a `SubClassOf(`$\Theta'$`C')` axiom, where Θ' is Θ as an OWL2 class expression, unraveled up to depth *n*.

   c. Finally, the *DOAs(C)* and *LEXAs(C)* are rendered as annotation assertions: i.e., `AnnotationAssertion(:user-description C' string), AnnotationAssertion(:concept2words C' string),` etc.

The Example KB in OWL2 Format

```
Prefix(
:=<http://www.projecthalo.com/aura#>)

Ontology(
<http://www.projecthalo.com/aura>

 Declaration(
  AnnotationProperty(:description))

 Declaration(
  AnnotationProperty(:user-description))

 Declaration(
  AnnotationProperty(:concept2words))

 ClassAssertion(:Color-Constant :green)

 ObjectPropertyRange(:has-part :Tangible-Entity)

 ObjectPropertyDomain(:has-part :Tangible-Entity)

 Declaration(
  ObjectProperty(:has-part))

 SubClassOf(
  ObjectIntersectionOf(:Tangible-Entity
   ObjectSomeValuesFrom(:has-part :TestD)) :TestC)

 DisjointClasses(:TestC :TestD)
```

```
 SubClassOf(:TestC
  ObjectIntersectionOf(:Tangible-Entity
   ObjectIntersectionOf(
    ObjectMaxCardinality(1 :has-part :TestD)
    ObjectSomeValuesFrom(:has-part
     ObjectIntersectionOf(:TestD
      ObjectSomeValuesFrom(:age
       ObjectIntersectionOf(:Duration-Value
        ObjectIntersectionOf(
         DataHasValue(:the-cardinal-value "43.0e0"^^xsd:float)
         ObjectHasValue(:cardinal-unit-class :year))))
      ObjectSomeValuesFrom(:color
       ObjectIntersectionOf(:Color-Value
        DataHasValue(:value "*green"^^xsd:string)))
      ObjectSomeValuesFrom(:size
       ObjectIntersectionOf(:Size-Value
        ObjectIntersectionOf(
         ObjectHasValue(:the-scalar-value :big)
         ObjectSomeValuesFrom(:scalar-unit-class :House)))))))))

 AnnotationAssertion(:user-description :TestC "This is TestC, a Test
concept."^^xsd:string)

 DisjointClasses(:TestD :TestC)

 SubClassOf(:TestD :Tangible-Entity)

 SubClassOf(:TestE
  ObjectIntersectionOf(:Tangible-Entity
   ObjectSomeValuesFrom(:has-part
    ObjectIntersectionOf(:TestC
     ObjectSomeValuesFrom(:has-part
      ObjectIntersectionOf(:TestD
       ObjectSomeValuesFrom(:has-state :Thing)))))))

 SubClassOf(:TestSubC
  ObjectIntersectionOf(:TestC
   ObjectSomeValuesFrom(:has-part
    ObjectIntersectionOf(:TestD
     ObjectSomeValuesFrom(:has-state :Thing)))))
```

The Translator Program and API

From the AURA console, the KB translation process can be started as follows.

1. Get the most recent reconstructed AURA KB http://www.ai.sri.com/halo/halobook2010/exported-kb/reconstructed/
2. Get the most recent AURA build and load this KB.
3. After AURA started up, from the console, invoke the following commands:
4. `:pa km`
5. `(read-solutions-from-file <path-to-file-reconstructed-kb.lisp>)` (reading in this data structure takes approx. 5 minutes)
6. Use one of the following functions to initiate the translations: `(export-all)`, `(export-fopl)`, `(export-tptp)`, `(export-asp)`, `(export-SILK)`, `(export-owl)`.

The generate KBs (kb-as-one-file versions) and corresponding subdirectories (containing the one-file-per-concept versions) will be found in / under the subdirectory `aura/the-back-end`.
The above `export`-functions call the main export function, which has the following signature; note that `{ a, b, c }` means specify one of `a, b,` or `c`:

```
(render-kb {:fopl, :tptp, :asp, :silk}
           :inherited-p {t, nil}
           :umap-version-p {t, nil})
```

And for OWL2:

```
(render-kb :owl
           :max-depth {nil, 0, …, n } ;; if nil, don't use unraveling
           :render-value-classes-p {t, nil}
           :render-cardinality-constraints-p {t, nil, :km-relevant-only}
                                             ;; :km-relevant-only exports only
                                             ;; cardinalities 0 and 1! Other
                                             ;; cardinalities are basically
                                             ;; ignored by KM.
           :render-relation-axiom-p {t, nil}
           :render-disjointness-axioms-p {t, nil}
           :render-triggers-p {t, nil}
           :use-id-concepts-p {t, nil}
           :use-nominals-for-constants-p {t, nil}
           :use-atomic-symbol-concepts-p {t, nil}
           :inherited-p {t, nil}
           :umap-version-p {t, nil})
```

There is also an OWL2 export function which does not require a reconstructed KB. This was used in the past for debugging the KB with Protégé 4.2. To invoke it, skip step 5. above and just call `(export-owl-for-p4-debugging)`. This function calls the main function `render-kb-as-owl` which takes the same arguments as `render-kb`, but without the first argument specifying the syntax (the syntax is fixed to OWL2 here), and the `inherited-p` and `umap-version-p` keyword arguments (without a reconstructed KB, we cannot make those distinctions, because the *EQAs* are not available then).