The KM to SILK Translator - Preliminary Design and Implementation

Working Note 38
Peter Clark (peter.e.clark@boeing.com)
June19th 2010

Introduction

The goal of the "level 0" translator is to translate the core part of KM's prototype representations to SILK. This document assumes some familiarity with both of those representational formalisms. In a nutshell, there are two parts to the translation, an easy part and a difficult part:

- The **easy part** is to translate the prototype data structure to a semantically equivalent data structure in SILK. Because prototypes have a simple "forall...exists..." semantics, this translation is straightforward.
- The **difficult part** is the task of specifying the correct coreferences between elements of the different prototypes. A cell may have DNA, and a eukaryotic cell may have DNA, and KM relies on its unification operator (UMap) to heuristically combine (clones of) different prototypes, e.g., the DNA of the cell and eukaryotic cell, together, essentially "guessing" coreference. In the SILK version, we propose to assert the intended coreferences by reuse of Skolem terms, and thus make that combination a deductive (and hence sound) process. We believe that it is (largely) possible to automatically identify the intended coreferences from KM's knowledge base by tracing how different prototypes were created in the first place. The underlying conjecture is that if a participant of prototype 2 was originally cloned from a participant of prototype 1, then the user intends them to be coreferential, and hence they should be assigned the same Skolem terms to make the explicit and deductive in SILK. The bulk of this document expands on this idea.

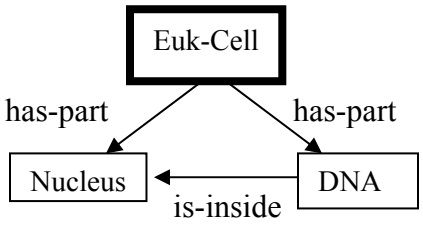
As background to this issue of UMap vs. shared Skolem terms, we elaborate a bit more here. In KM, if the user queries an instance for the values of one of its slots, then that instance will acquire all the properties specified by the prototypes of that instance's classes (via first cloning that prototype, then unifying the cloned root with the instance - a "clone and unify" operation). If multiple prototypes apply, then multiple clones will be unified with the queried instance, and KM relies heavily on its unification operator (UMap) to combine the different parts of the clones together. If one thinks of a prototype as a graph of relationships, then this operation can be visualized as merging (copies of) the graphs together. During this process KM is making heuristic "guesses" about the likely coreferences between (copies of) participants in different prototypes. SILK does not have an equivalent UMap operator, and analysis in 2009 suggests that a declarative formalization of it would be somewhat difficult. However, there is another avenue for handling this - if we can identify the coreferences that the user *intended*, then we can replace unification with equivalence through the use of shared Skolem terms. One might view UMap as a mechanism to heuristically try and recreate the coreferences the user intended, but were not recorded in the KB. But if we can recreate the intended coreferences, we can make them explicit in SILK and thus not need some UMap equivalent in SILK. And it appears that, for the most part, such intensions can be identified, as KM fortuitously records the way that its prototypes were created. We elaborate further below, but first let's deal with the simple part!

In addition, for the special case of "functional' predicates (slots), i.e., ones that take at most one value (or at most one value of a particular type), there is an additional mechanism available for enforcing coreference, namely "at most 1 *class*" cardinality constraints. For example, if a cell has

at most one nucleus, an "at most 1 Nucleus" axiom in SILK will enforce that all implied nuclei of a cell are coreferential, even if their Skolem names differ (the axiom will enforce equality between Skolems). In AURA users sometimes omit such cardinality constraints either because UMap already performs the required inferences or because they are not universally true (e.g., a cell *can*, occasionally, have more than 1 nucleus), but if such constraints can be identified and expressed in SILK, this would provide an additional mechanism for enforcing appropriate equalities in certain cases. We do not discuss this option further in this document as the mechanism of shared Skolem names is more general and seems adequate. However, it might be a useful additional mechanism to bear in mind, and might contribute to the simplicity and/or computational efficiency of the translated KBs in SILK.

Basic Translation to SILK

Example

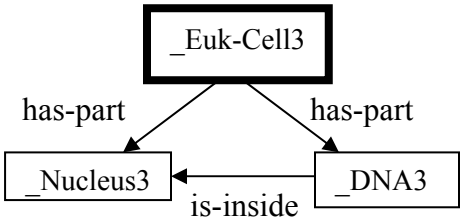


Let's consider the simplest case of a Euk-Cell (eukaryotic cell) that has a DNA and a Nucleus, and the Nucleus encloses the DNA. This would be created in AURA as the graph below:
This graph states that for all Euk-Cells, it has a nucleus and DNA part, and that the DNA is inside the Nucleus. When saved, the following file Euk-Cell.km would be automatically synthesized from the graph:

```
;;; Prototype of Euk-Cell
(_Euk-Cell3 has
   (instance-of (Euk-Cell))
   (prototype-of (Euk-Cell))
   (prototype-scope (Euk-Cell))
   (prototype-participants (_Euk-Cell3 _DNA3 _Nucleus3))
   (has-part (_DNA3 _Nucleus3))
   (prototype-participant-of (_Euk-Cell3)))

(_DNA3 has
   (instance-of (DNA))
   (is-part-of (_Euk-Cell3))
   (is-inside (_Nucleus3))
   (prototype-participant-of (_Euk-Cell3)))

(_Nucleus3 has
   (instance-of (Nucleus))
   (is-part-of (_Euk-Cell3))
   (encloses (_DNA3))
   (prototype-participant-of (_Euk-Cell3)))
```

The formal semantics of this can be written:

$\forall$c isa(c,EukCell)
$\rightarrow \exists$ n,d isa(n,Nucleus), isa(d,DNA), has-part(c,n), has-part(c,d), is-inside(d,n).



For our purposes here, it's easier to redraw the diagram showing the explicit Skolem numbers in the saved prototype, although the user doesn't see those:

To translate this into SILK we generate a separate SILK axiom for each arc in the graph. We could generate a single, giant axiom for the entire graph, however one axiom per arc seems simpler.

A Quick Note on Inverses

Each arc in the graph really denotes 2 equivalent literals, as each predicate f(x,y) has an inverse f^{-1}(y,x), e.g.,

has-part(_Euk-Cell3, _Nucleus3)

can also be written

is-part-of(_Nucleus3, _Euk-Cell3)

and

is-inside(_DNA3, _Nucleus3)

can also be written

encloses(_Nucleus3, _DNA3)

as has-part and is-part-of are "inverses", and is-inside and encloses are "inverses". The current translator prints out each literal in SILK in one direction only (making an arbitrary choice of which one), and assumes that somewhere else the relation between the predicate and its inverse is expressed in SILK. In AURA, every predicate includes a note of its inverse predicate (on the "inverse" slot in the KM KB). Translating these inverse declarations to SILK is straightforward, and not described here. If these inverse declarations are not translated, the alternative is to modify the translator to print each literal in both forward and inverse directions.

Translation to SILK

As each Skolem constant above denotes a different individual in each clone of the prototype, we use a Skolem term, parameterized by the root instance of the prototype, namely _Euk-Cell3. Then, to convert this into a SILK axiom that applies to *all* Euk-Cells, we replace _Euk-Cell3 with a variable ?x of type Euk-Cell. (Also note syntactically we have to replace "-" with "_", as "-" has special meaning in SILK). The result looks as follows:

```
KM: (load "km")
KM: (load "silk-translator")
KM: (assert-demo1)
```

```
KM: (translate-class '|Euk-Cell|)
// ==========================================
//                SILK REPRESENTATION OF Euk-Cell
// ==========================================

?x[has_part->{_DNA3(?x)#DNA,_Nucleus3(?x)#Nucleus}] :- ?x#Euk_Cell ;

_DNA3(?x)[is_inside->_Nucleus3(?x)#Nucleus] :- ?x#Euk_Cell ;
```

Note that each part of a euk-cell, e.g., the DNA, is parameterized by the euk-cell that is belongs to, i.e., for any euk-cell ?x, the DNA is denoted by the function _DNA3(?x). _DNA3 is a somewhat arbitrary name to use for this Skolem term -- note, though, we couldn't use a function name like dna(?x) or the-cells-dna(?x) because the euk-cell may have more than one DNA, and thus we need a distinct function name for each entity in the prototype. Using the Skolem names from KM seems as good a method for achieving this. In addition, as we describe later, it turns out to be important for specifying coreferences.

Given a new Euk-Cell, e.g., euk-cell01, SILK should infer that:

```
euk-cell01#Euk-Cell[
  has-part->{_Nucleus3(euk-cell01)#Nucleus,
             _DNA3(euk-cell01)#DNA[
                  is-inside->{_Nucleus3(euk-cell01)}]}]
```

For this particular Euk-Cell, its nucleus is denoted by the (instantiated) Skolem term _Nucleus3(euk-cell01). A different Euk-Cell will clearly have a different Nucleus, denoted by the same Skolem function but a different argument value.

From UMap to Skolem Terms

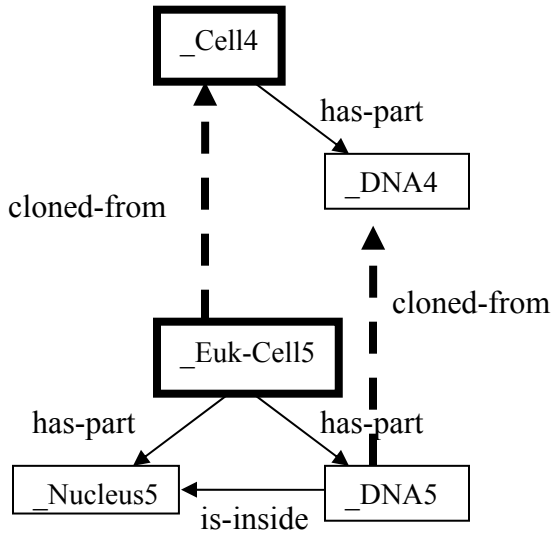
Almost always in AURA, a user creates a more specific version of a concept by a "clone and extend" operation. For example, suppose we have a prototype of a cell with DNA:

```
;;; Prototype of Cell
(_Cell4 has
   (instance-of (Cell))
   (prototype-of (Cell))
   (prototype-scope (Cell))
   (prototype-participants (_Cell4 _DNA4))
   (has-part (_DNA4))
   (has-clones (_Euk-Cell5))
   (has-built-clones (_Euk-Cell5))
   (prototype-participant-of (_Cell4)))

(_DNA4 has
   (instance-of (DNA))
   (is-part-of (_Cell4))
   (has-clones (_DNA5))
   (prototype-participant-of (_Cell4)))
```

Now the user creates a Euk-Cell, namely a Cell with a nucleus. Typically the user will start an instance of Euk-Cell and declare that it is a Cell. During knowledge entry, KM will then create and unify a clone of the Cell graph into the Euk-Cell graph that the user is working on. Suppose he/she then adds a nucleus part, states the DNA is in that nucleus, and then saves it. The saved result will look:

```
;;; Prototype of Euk-Cell
(Euk-Cell has (superclasses (Cell)))

(_Euk-Cell5 has
   (instance-of (Euk-Cell))
   (prototype-of (Euk-Cell))
   (prototype-scope (Euk-Cell))
   (prototype-participants (_Euk-Cell5 _DNA5 _Nucleus5))
   (prototype-participant-of (_Euk-Cell5))
   (cloned-from (_Cell4))
   (cloned-built-from (_Cell4))
   (has-part (_DNA5 _Nucleus5)))

(_DNA5 has
   (instance-of (DNA))
   (is-part-of (_Euk-Cell5))
   (is-inside (_Nucleus5))
   (cloned-from (_DNA4))
   (prototype-participant-of (_Euk-Cell5)))

(_Nucleus5 has
   (instance-of (Nucleus))
   (is-part-of (_Euk-Cell5))
   (encloses (_DNA5))
   (prototype-participant-of (_Euk-Cell5)))
```

Note the cloned-from links here: _Euk-Cell5 is noted as cloned-from _Cell4, and _DNA5 is



cloned-from _DNA4.

Suppose we now ask KM> about the parts of a Euk-Cell:

```
KM> (comments)
KM> (a Euk-Cell)
(_Euk-Cell2)

KM> (the has-part of _Euk-Cell2)
(COMMENT: Cloned _Euk-Cell5 -> _Euk-Cell2 to find (the has-part
of _Euk-Cell1))
```

```
(COMMENT: (_Euk-Cell1 &! _Euk-Cell2) unified to be _Euk-Cell1)
(COMMENT: Cloned _Cell4 -> _Cell5 to find (the has-part of _Euk-
Cell1))
(COMMENT: (_Euk-Cell1 &+ _Cell5) unified to be _Euk-Cell1)
(COMMENT: (_DNA3 &+! _DNA6) unified to be _DNA3)
(COMMENT: (_Euk-Cell1 &! _Cell5) unified to be _Euk-Cell1)
(_DNA3 _Nucleus4)
```

Note the above comments: KM has cloned _Cell4 and _Euk-Cell5 to create a representation of _Euk-Cell2. Also note that _Euk-Cell2 only has one, not two, DNA (namely _DNA3) -- KM has heuristically decided to unify the clones of _DNA4 and of _DNA5, when merging information on the has-part slot, because they are the same type.

Although SILK does not have UMap, we can see from the cloned-from link on _DNA5 -- we'll call this the "derivation history" of _DNA5 -- that it was originally cloned-from _DNA4. Thus this derivation history shows that _DNA5 is really intended to be the same thing as _DNA4. When merging values, KM does not use this derivation history -- in fact, one might say that UMap is a procedure that (in this context at least) is heuristically attempting to recover information that was "lost" during the knowledge base construction. However, it is not in fact completely lost -- the internal bookkeeping cloned-from slots implicitly store it, and for SILK we will attempt to recover it from these connections.

As _DNA5 was cloned-from _DNA4, we can consider that they (or more precisely, their clones for a new instance) are meant to be coreferential:

- In the SILK representation of Cell, a cell's DNA is denoted by _DNA4(?y), where ?y is the instance of that cell.
- In the SILK representation of Euk-Cell, the euk-cell's DNA is denoted by _DNA5(?x), where ?x is the instance of that euk-cell.

For a Euk-Cell ?x, the SILK axioms about Cell (expressed using ?y) apply to ?x under the binding ?y :=: ?x. Thus the coreference can be expressed

```
_DNA5(?x) :=: _DNA4(?y)
```

and as ?y :=: ?x this becomes:

```
_DNA5(?x) :=: _DNA4(?x)
```

The full translation looks as follows:

```
KM: (assert-demo2)
KM: (translate-all)
// ================================================
//                SILK REPRESENTATION OF Cell
// ================================================

?x[has_part->_DNA4(?x)#DNA] :- ?x#Cell ;

// ================================================
//                SILK REPRESENTATION OF Euk-Cell
// ================================================

?x[has_part->{_DNA5(?x)#DNA,_Nucleus5(?x)#Nucleus}] :- ?x#Euk_Cell ;

_DNA5(?x)[is_inside->_Nucleus5(?x)#Nucleus] :- ?x#Euk_Cell ;


_DNA5(?x) :=: _DNA4(?x) :- ?x#Euk-Cell ;
```

Note the equality in the last line expressing the coreference of the DNA in the Euk-Cell with the DNA in the Cell. We can simplify this by substituting _DNA4 for _DNA5, so this becomes simply:

```
KM: (assert-demo2)
KM: (translate-class '#$Euk-Cell)

// ================================================
//                SILK REPRESENTATION OF Euk-Cell
// ================================================

?x[has_part->{_DNA4(?x)#DNA,_Nucleus5(?x)#Nucleus}] :- ?x#Euk_Cell ;

_DNA4(?x)[is_inside->_Nucleus5(?x)#Nucleus] :- ?x#Euk_Cell ;
```

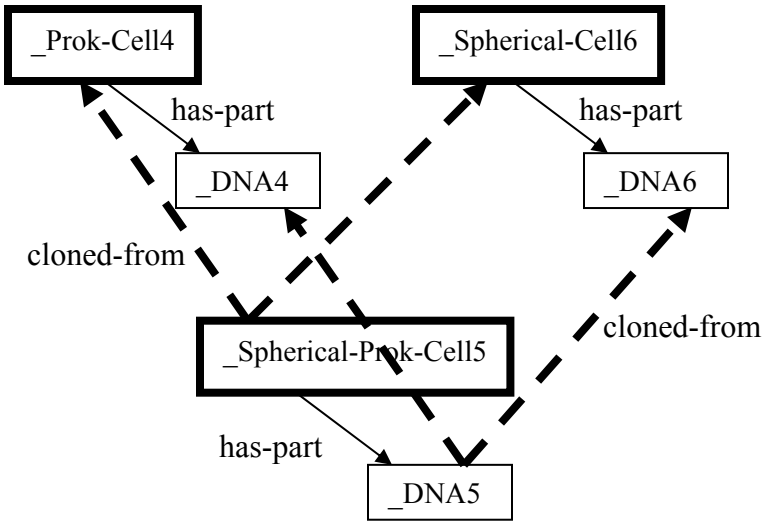
The important thing to note is that both the has-part of Cell and Euk-Cell are now denoted by the Skolem term _DNA4(?x), and thus the equality and _DNA5 is not needed. Thus the implied DNA part from both these axioms will necessarily be the same for any ?x, as the Skolem terms have the same function name.

Note that in AURA this cloning operation may be iterated, e.g., X may clone to Y, which is saved in a prototype and then later Y is cloned to Z, etc. If this happens, then we can trace the chain of cloned-from links to find the *original* KM Skolem constant (X) for Y and Z, and use that as the Skolem term for both (the SILK representation of) Y and Z. This Lisp function doing this trace-back in the code is (node-cloned-from-originally <protoinstance>).

Special Considerations

Multiple Clones onto a Node

Sometimes multiple prototypes can contribute to a single node, e.g.,:



Here a “spherical prokaryotic cell” is defined as both a “prokaryotic cell” and a “spherical cell”. Note that _DNA5 is a clone of _DNA4 and _DNA6 (KM has unified the clones of _DNA4 and _DNA6 to produce _DNA5). In KM this looks:

```
;;; Prototype of Prok-Cell
(_Prok-Cell4 has
   (instance-of (Prok-Cell))
   (prototype-of (Prok-Cell))
   (prototype-scope (Prok-Cell))
   (prototype-participants (_Prok-Cell4 _DNA4))
   (has-part (_DNA4))
   (has-clones (_Spherical-Prok-Cell5))
   (has-built-clones (_Spherical-Prok-Cell5))
   (prototype-participant-of (_Prok-Cell4)))

(_DNA4 has
   (instance-of (DNA))
   (is-part-of (_Prok-Cell4))
   (has-clones (_DNA5))
   (prototype-participant-of (_Prok-Cell4)))

;;; ----------

;;; Prototype of Spherical-Cell
(_Spherical-Cell6 has
   (instance-of (Spherical-Cell))
   (prototype-of (Spherical-Cell))
   (prototype-scope (Spherical-Cell))
   (prototype-participants (_Spherical-Cell6 _DNA6))
   (has-part (_DNA6))
   (has-clones (_Spherical-Prok-Cell5))
   (has-built-clones (_Spherical-Prok-Cell5))
   (prototype-participant-of (_Spherical-Cell6)))

(_DNA6 has
   (instance-of (DNA))
   (is-part-of (_Spherical-Cell6))
   (has-clones (_DNA5))
   (prototype-participant-of (_Spherical-Cell6)))

;;; ----------

;;; Prototype of Spherical-Prok-Cell
(Spherical-Prok-Cell has
   (superclasses (Prok-Cell Spherical-Cell)))

(_Spherical-Prok-Cell5 has
   (instance-of (Spherical-Prok-Cell))
   (prototype-of (Spherical-Prok-Cell))
   (prototype-scope (Spherical-Prok-Cell))
   (prototype-participants (_Spherical-Prok-Cell5 _DNA5))
   (prototype-participant-of (_Spherical-Prok-Cell5))
   (cloned-from (_Prok-Cell4 _Spherical-Cell6))
   (cloned-built-from (_Prok-Cell4 _Spherical-Cell6))
   (has-part (_DNA5 _Nucleus5)))

(_DNA5 has
```

```
   (instance-of (DNA))
   (is-part-of (_Spherical-Prok-Cell5))
   (cloned-from (_DNA4 _DNA6))
   (prototype-participant-of (_Spherical-Prok-Cell5)))
```

This produces SILK as follows:

```
KM: (assert-demo5)
KM: (translate-all)

// ========================================
//                SILK REPRESENTATION OF Prok-Cell
// ========================================

?x[has_part->_DNA4(?x)#DNA] :- ?x#Prok_Cell ;

// ========================================
//                SILK REPRESENTATION OF Spherical-Cell
// ========================================

?x[has_part->_DNA6(?x)#DNA] :- ?x#Spherical_Cell ;

// ========================================
//                SILK REPRESENTATION OF Spherical-Prok-Cell
// ========================================

_DNA5(?x) :=: _DNA4(?x) :- ?x#Spherical_Prok_Cell ;
_DNA5(?x) :=: _DNA6(?x) :- ?x#Spherical_Prok_Cell ;

?x[has_part->{_DNA5(?x)#DNA,
    _Nucleus5(?x)}] :- ?x#Spherical_Prok_Cell ;
```

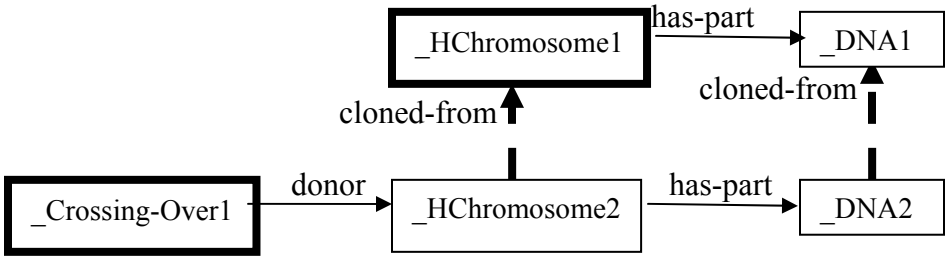
Note that above there are two, not one, equalities, as there are two instances that _DNA5 was independently derived from, and thus _DNA5 should be considered coreferential to both. Because we have more than one equality, we can't remove them all from the final representation by substituting other Skolem names[1]. Rather, the best we can do is remove the first equality by globally substituting _DNA4 for _DNA5 in the representation of Spherical-Prok-Cell, and leave the second equality:

```
KM: (translate-class '#$Spherical-Prok-Cell)

// ========================================
//                SILK REPRESENTATION OF Spherical-Prok-Cell
// ========================================

_DNA4(?x) :=: _DNA6(?x) :- ?x#Spherical_Prok_Cell ;

?x[has_part->{_DNA4(?x)#DNA,
    _Nucleus5(?x)}] :- ?x#Spherical_Prok_Cell ;
```

[1] Of course, it would have been better if the knowledge engineer had made Prok-Cell and Spherical-Cell both inherit their (same) DNA from a common generalization, Cell; if the engineer had done this, then we would not get the two equalities. This example is thus a little contrived, but its point is to show how to handle multiple inheritance.

Cloning onto Non-protoroot Nodes

In AURA, Euk-Cell was created by creating an instance of Euk-Cell, _Euk-Cell5, cloning Cell onto it, adding more info, and then saving _Euk-Cell5 as a new prototype (of Euk-Cell). In this case, Cell was cloned onto the root node of the new prototype, _Euk-Cell5. However, in general cloning can happen onto any instance, and if the cloning is made onto a node that does not later become the root of a new prototype, additional considerations are needed. This is best explained with an example. Let's consider defining that all HChromosomes have a DNA, and then all Crossing-Overs involve an HChromosome (which has a DNA of course). Here, the HChromosome in Crossing-Over is cloned from the prototype of HChromosome:



In KM this looks:

```
;;; Prototype of HChromosome
(_HChromosome1 has
    (instance-of (HChromosome))
    (prototype-of (HChromosome))
    (prototype-scope (HChromosome))
    (prototype-participants (_HChromosome1 _DNA1))
    (prototype-participant-of (_HChromosome1))
    (has-part (_DNA1))
    (has-clones (_HChromosome2))
    (has-built-clones (_HChromosome2)))

(_DNA1 has
    (instance-of (DNA))
    (is-part-of (_HChromosome1))
    (prototype-participant-of (_HChromosome1))
    (has-clones (_DNA2)))

;;; Prototype of Crossing-Over
(_Crossing-Over1 has
    (prototype-of (Crossing-Over))
    (prototype-scope (Crossing-Over))
    (instance-of (Crossing-Over))
    (prototype-participants (_Crossing-Over1 _HChromosome2
                             _DNA2))
    (prototype-participant-of (_Crossing-Over1))
    (donor (_HChromosome2)))

(_HChromosome2 has
    (instance-of (HChromosome))
    (cloned-from (_HChromosome1))
    (cloned-built-from (_HChromosome1))
    (has-part (_DNA2))
```

```
    (prototype-participant-of (_Crossing-Over1)))

(_DNA2 has
    (instance-of (DNA))
    (cloned-from (_DNA1))
    (prototype-participant-of (_Crossing-Over1)))
))
```

which will look in SILK, first without showing any coreferences, as follows:

```
KM: (assert-demo3)
KM: (translate-all)
// ======================================
//                SILK REPRESENTATION OF HChromosome
// ======================================

?x[has_part->_DNA1(?x)#DNA] :- ?x#HChromosome ;

// ======================================
//                SILK REPRESENTATION OF Crossing-Over
// ======================================

?x[donor->_HChromosome2(?x)#HChromosome] :- ?x#Crossing_Over ;

_HChromosome2(?x)#HChromosome[
    has_part->_DNA2(?x)#DNA] :- ?x#Crossing_Over ;
```

KM's cloned-from links show that the protoroot _HChromosome1 was cloned to create _HChromosome2, along with its participant _DNA1 cloned to create _DNA2. If ?x is an instance of Crossing-Over, then the coreference implied by KM's cloned-from links is:

```
_DNA2(?x) :=: _DNA1( argument ) :- ?x#Crossing_Over ;
```

where *argument* is the HChromosome in ?x that _DNA2(?x) is part of. As we can see, *argument* is in fact _HChromosome2(?x), and thus the equality is:

```
_DNA2(?x) :=: _DNA1(_HChromosome2(?x)) :- ?x#Crossing_Over ;
```

Note the doubly nested Skolem term _DNA1(_HChromosome2(?x)); you can read this as "the dna of the hchromosome of ?x". Thus the full translation is:

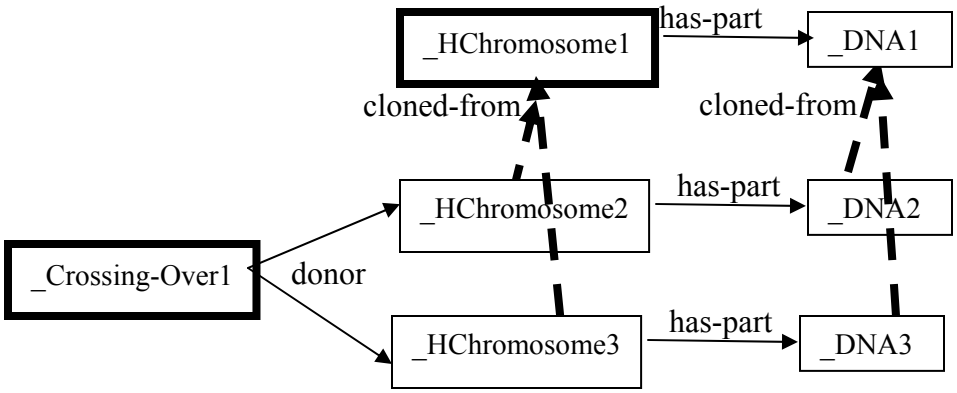
```
KM: (translate-class '#$Crossing-Over)

// ======================================
//                SILK REPRESENTATION OF Crossing-Over
// ======================================

?x[donor->_HChromosome2(?x)#HChromosome] :- ?x#Crossing_Over ;

_HChromosome2(?x)#HChromosome[
    has_part->_DNA2(?x)#DNA] :- ?x#Crossing_Over ;

_DNA2(?x) :=: _DNA1(_HChromosome2(?x)) :- ?x#Crossing_Over ;
```

Note the equality in the last line. Again we can simplify by substituting _DNA1(_HChromosome2(?x)) for _DNA2(?x), removing the equality:

```
KM: (translate-class '#$Crossing-Over)
// ======================================
//                SILK REPRESENTATION OF Crossing-Over
// ======================================

?x[donor->_HChromosome2(?x)#HChromosome] :- ?x#Crossing_Over ;

_HChromosome2(?x)#HChromosome[
    has_part->_DNA1(_HChromosome2(?x))#DNA] :- ?x#Crossing_Over ;
```

If a Prototype is Cloned Multiple Times into Another Prototype

In fact, biologically speaking Crossing-Over involves *two* HChromosome donors, which looks:



and is expressed in KM:

```
;;; Prototype of Crossing-Over
(_Crossing-Over1 has
    (prototype-of (Crossing-Over))
    (prototype-scope (Crossing-Over))
    (instance-of (Crossing-Over))
    (prototype-participants (_Crossing-Over1 _HChromosome2
                             _HChromosome3 _DNA2 _DNA3))
    (prototype-participant-of (_Crossing-Over1))
    (donor (_HChromosome2 _HChromosome3)))

(_HChromosome2 has
    (instance-of (HChromosome))
    (cloned-from (_HChromosome1))
    (cloned-built-from (_HChromosome1))
    (has-part (_DNA2))
    (prototype-participant-of (_Crossing-Over1)))

(_DNA2 has
    (instance-of (DNA))
    (cloned-from (_DNA1))
    (prototype-participant-of (_Crossing-Over1)))

(_HChromosome3 has
    (instance-of (HChromosome))
    (cloned-from (_HChromosome1))
```

```
        (cloned-built-from (_HChromosome1))
        (has-part (_DNA3))
        (prototype-participant-of (_Crossing-Over1)))

    (_DNA3 has
        (instance-of (DNA))
        (cloned-from (_DNA1))
        (prototype-participant-of (_Crossing-Over1)))
    ))
```

and translates into SILK:

```
KM: (assert-demo4)
KM: (translate-all)
// ======================================
//                SILK REPRESENTATION OF HChromosome
// ======================================

?x[has_part->_DNA1(?x)#DNA] :- ?x#HChromosome ;

// ======================================
//                SILK REPRESENTATION OF Crossing-Over
// ======================================

?x[donor->{_HChromosome2(?x)#HChromosome,
    _HChromosome3(?x)#HChromosome}] :- ?x#Crossing_Over ;

_HChromosome2(?x)#HChromosome[
    has_part->_DNA1(_HChromosome2(?x))#DNA] :- ?x#Crossing_Over ;

_HChromosome3(?x)#HChromosome[
    has_part->_DNA1(_HChromosome3(?x))#DNA] :- ?x#Crossing_Over ;
```

Note the two Skolem terms _DNA1(_HChromosome2(?x)) and _DNA1(_HChromosome3(?x)), denoting "the dna of the first hchromosome of ?x" "the dna of the second hchromosome of ?x".

This example is unusual and somewhat complicated, because the *same* prototype (_HChromosome1) is being cloned *twice* into another prototype (_Crossing-Over1), once onto _HChromosome2 (producing _DNA2) and once onto _HChromosome3 (producing _DNA3). Unfortunately the cloned-from links doesn't quite allow the translator to automatically determine whether _DNA2 came from the cloning of _HChromosome1 onto _HChromosome2 or _HChromosome3 -- all the translator can see from the has-clones link is that _HChromosome1 was cloned onto both _HChromosome2 or _HChromosome3 in _Crossing-Over1, but not *which* clone produced _DNA2[2]. As a result, in the SILK the translator can't tell whether:

```
_DNA2(?x) :=: _DNA1(_HChromosome2(?x))
```

or

```
_DNA2(?x) :=: _DNA1(_HChromosome3(?x))
```

In other words, did _DNA2 come from the cloning of _HChromosome1 onto _HChromosome2 or _HChromosome3? We could answer this by some graph matching algorithm, to see whether _HChromosome2 - _DNA2 or _HChromosome3 - _DNA2 are in the same structural relationship as _HChromosome1 - _DNA1. The current implementation does a weaker version of this,

[2] It would be useful to count how often such cases occur; they appear to be fairly rare in the AURA reference biology KB, maybe 5% or less. The proposed heuristic way of "guessing" the appropriate equality appears to work correctly in all the cases I've looked at.

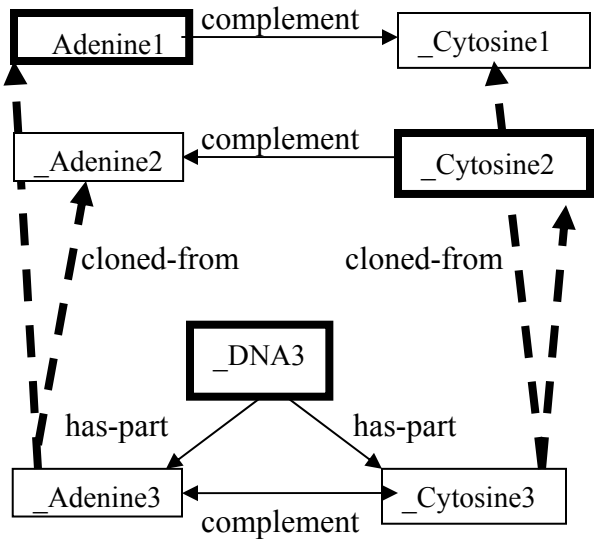
looking to see if _DNA2 is closer (in terms of graph distance) to _HChromosome2 or _HChromosome3, and picks the closest potential source. This is implemented by the function shortest-distance-between in the translator.

Loops in the Skolem Nesting

In some unusual cases, when attempting to simplify one Skolem expression using another, the system can end up looping. This is best illlustrated with an example. Here, all adenines have the complement cytosine, and all cytosines have the complement adenine, and all DNAs have an adenine and cytosine part (which are complements of each other). The adenine and cytosine in the DNA inherit from both adenine and cytosine:

which looks in KM as:

```
;;; Prototype of Adenine
(_Adenine1 has
  (instance-of (Adenine))
  (prototype-of (Adenine))
  (prototype-scope (Adenine))
  (prototype-participants (_Adenine1 _Cytosine1))
  (complement (_Cytosine1))
  (has-clones (_Adenine3))
  (has-built-clones (_Adenine3))
  (prototype-participant-of (_Adenine1)))

(_Cytosine1 has
  (instance-of (Cytosine))
  (complement (_Adenine1))
  (has-clones (_Cytosine3))
  (prototype-participant-of (_Adenine1)))
```




```
;;; ----------

;;; Prototype of Cytosine
(_Cytosine2 has
  (instance-of (Cytosine))
  (prototype-of (Cytosine))
```

```
  (prototype-scope (Cytosine))
  (prototype-participants (_Cytosine2 _Adenine2))
  (complement (_Adenine2))
  (has-clones (_Cytosine3))
  (has-built-clones (_Cytosine3))
  (prototype-participant-of (_Cytosine2)))

(_Adenine2 has
  (instance-of (Adenine))
  (complement (_Cytosine2))
  (has-clones (_Adenine3))
  (prototype-participant-of (_Cytosine2)))

;;; ----------

;;; Prototype of DNA
(_DNA3 has
  (instance-of (DNA))
  (prototype-of (DNA))
  (prototype-scope (DNA))
  (prototype-participants (_DNA3 _Cytosine3 _Adenine3))
  (has-part (_Cytosine3 _Adenine3))
  (prototype-participant-of (_DNA3)))

(_Adenine3 has
  (instance-of (Adenine))
  (complement (_Cytosine3))
  (is-part-of (_DNA3))
  (cloned-from (_Adenine1 _Adenine2))
  (clone-built-from (_Adenine1))
  (prototype-participant-of (_DNA3)))

(_Cytosine3 has
  (instance-of (Cytosine))
  (complement (_Adenine3))
  (is-part-of (_DNA3))
  (cloned-from (_Cytosine1 _Cytosine2))
  (clone-built-from (_Cytosine2))
  (prototype-participant-of (_DNA3)))
```

and which translates to SILK as:

```
KM: (assert-demo6)
KM: (translate-all)
// ======================================
//                SILK REPRESENTATION OF Adenine
// ======================================

?x[complement->_Cytosine1(?x)#Cytosine] :- ?x#Adenine ;

// ======================================
//                SILK REPRESENTATION OF Cytosine
// ======================================

?x[complement->_Adenine2(?x)#Adenine] :- ?x#Cytosine ;

// ======================================
//                SILK REPRESENTATION OF DNA
```

```
// ======================================

_Adenine3(?x) :=: _Adenine2(_Cytosine3(?x)) :- ?x#DNA ;
_Cytosine3(?x) :=: _Cytosine1(_Adenine3(?x)) :- ?x#DNA ;

?x[has_part->{_Cytosine3(?x)#Cytosine,
    _Adenine3(?x)#Adenine}] :- ?x#DNA ;

_Cytosine3(?x)#Cytosine[
    complement->_Adenine3(?x)#Adenine] :- ?x#DNA ;
```

Note the equalities above, repeated below here:

```
_Adenine3(?x) :=: _Adenine2(_Cytosine3(?x)) :- ?x#DNA ;
_Cytosine3(?x) :=: _Cytosine1(_Adenine3(?x)) :- ?x#DNA ;
```

Here, we can't remove either of them through substitution, because each uses the other in its equality. During the translation, the code detects a loop when constructing these Skolem expressions:

```
_Adenine3(?x) = _Adenine2(_Cytosine2(_Adenine2(_Cytosine2(...))))
```

and breaks the loop to produce the equalities shown above, avoiding the infinite recursion.

What Needs to Be Done Next?

This document and implementation has sketched a first round of the SILK translator. There are three areas that need work:

- The above use of cloned-from links is not fool-proof, and may need some debugging and possibly user interaction to correct the links. In particular:
  1. In a few cases, the user has not used a “clone and extend” operation for knowledge formulation, but a “clone and edit” operation of a sibling, e.g., creating a Prokaryotic-Cell by starting with Eukaryotic-Cell, editing it, then saving it. KM will record that the cloned elements of Prokaryotic-Cell were cloned-from Eukaryotic-Cell, although they clearly are not intended to be coreferential as Eukaryotic-Cell and Prokaryotic-Cell are mutually exclusive.

     The specific example encountered in the reference KB was with Uracil and Thymine. It appears that the user created Uracil by first opening Thymine, then saving it as Uracil (as both have similar properties, but are siblings and mutually exclusive). Thus tracing the cloned-from links, Uracil’s participants, e.g., Adenine are recorded as cloned-from Adenine in Thymine, and thus the current translator uses the same Skolem name for that Adenine in both Uracil and Thymine, thus incorrectly implying coreference. It might be possible to spot such inconsistencies in the translator, and have them blocked or have the user queried.
  2. In a few cases, some of the cloned-from links appear to be missing, i.e., the knowledge propogation in AURA has somehow lost a link somewhere. Identifying where there are “holes” in the translation is another area to be dealt with.
  3. Although I haven’t seen cases of this, the user might not have used a “clone and extend” operation to create a new concept and the cloned-from links might be missing and need to be repaired.
- The syntax of the generated SILK may still not be quite right in places
- The whole thing needs to be tested to make sure SILK really can do the reasoning it should do, and the translator fixed or extended when it does not. This might be the biggest task that needs addressing.

- There may be conceptual errors with the design of the translator that means it will not quite work as intended.
- There are some syntactic structures that do not translate to SILK yet, mainly constraint expressions and units of measure. The translator prints out warnings when it encounters them, e.g., in Eukaryotic-Cell:

```
KM: (translate-class 'Animal-Cell)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
//                SILK REPRESENTATION OF Animal-cell
//
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
// DEBUG: Don't know how to translate structure
//          (at-least 1 Mitochondria) to SILK (ignoring)
// DEBUG: Don't know how to translate structure
//          (at-least 10 Ribosome) to SILK (ignoring)
// DEBUG: Don't know how to translate structure
//          (:pair *big Prokaryotic-cell) to SILK (ignoring)
```

These also need to be translated to SILK, as well as the declarations of predicates and their inverses, e.g., has-part(x,y) ↔ is-part-of(y,x).