Reference Resolution and Views
Working Note 25

Peter Clark
Knowledge Systems
Boeing Maths and Computing Technology
peter.e.clark@boeing.com

May 2001

Abstract

A common phenomenon in text understanding is to refer to an entity which was not explicitly introduced earlier in the text, but rather whose existence is indirectly implied (or suggested) from earlier facts. For example, in "He dropped a cup. The handle broke off.", the handle referred to is not explicitly introduced, rather the reader infers that this (probably) means the handle of the cup. This working note gives a simple account for this phenomenon, based on selectively adding background knowledge to the "scene" the text is describing to make sense of it. We describe how this can be implemented in the representation language KM.

1 The Problem

A common phenomenon in text understanding is to refer to an entity which was not explicitly introduced earlier in the text, but rather whose existence is indirectly implied (or suggested) from earlier facts. For example, in "He dropped a cup. The handle broke off.", the handle referred to is not explicitly introduced, rather the reader infers that this (probably) means the handle of the cup. This working note gives a simple account for this phenomenon, based on selectively adding background knowledge to the "scene" the text is describing to make sense of it.

Consider interpreting the sentence (about an airplane's hydraulic system):

> "The hydraulic system supplies power to the thrust reversers."

The issue of interest here is how to resolve the reference to "the thrust reversers". Clearly (to a person) this refers to the thrust reversers on the airplane being talked about. How can we model this recognition computationally?

To understand a paragraph, we assume the computer is gradually building up a "scene" describing the paragraph's contents, where each sentence augments (or modifies) this scene, e.g., in the style of [Zadrozny and Jensen, 1991]. In our representation language KM, a scene is represented by a set of instances and relations, stored in working memory (a KM "context" [Clark and Porter, 1999]). In this particular case, the context will already contain an instance denoting the airplane (introduced in some earlier sentence), but will not contain any instance denoting the thrust reverser. So to resolve the

reference to "thrust reverser,"[1] the computer needs to elaborate the current scene, until something of type `Thrust-Reverser` is found.

A basic operation which KM can do is to query for the values of a particular slot on any object in the current scene. For example, a query for the `parts` of `_Airplane1` (denoting the airplane being talked about) will compute the direct parts of the airplane (e.g., a fuselage and two wings) and add them to the scene. We call this process *elaborating* the scene.

Thus one simple way of finding the `Thrust-Reverser` would be to query for the `parts`[2] of the airplane. This will return a thrust reverser, but will also return the entire list of parts of the airplane. This is rather undesirable for two reasons: First, it is computationally expensive to do this. Second, and more seriously, it will "clutter" the scene with every single aircraft part, making resolution of future anaphoric reference impossible. For example, suppose the next sentence refers to "the pump," meaning the pump in the hydraulic system. If we have elaborated the current scene to include every pump on the aircraft, then suddenly there are many referents of "the pump" (e.g., the fuel pump, the water pump in the coffee maker, etc.). The whole point of using a scene (KM context) was to denote objects actively under consideration, and hence provide focus of attention, but if we introduce every part of the aircraft into the context, we have destroyed any focus which was there.

2 Using Views

2.1 Representing Views

An alternative approach is to break up the representation of an aircraft into a number of *views*, e.g. top level structure, flight control surfaces, fuel system, electrical system, passenger view, cargo view etc. Each view encodes a subset of facts about the aircraft, namely just those relevant to that view. Views may overlap (a fact may be shared between views), and we assume that all views are mutually consistent. The elaboration task then becomes one of selecting a view which introduces the thrust reverser to the scene. As a side effect, this will add some, but not all, other aircraft objects into the scene.

How can we represent views? Our representation language KM has three possible, alternative mechanisms which might support this:

Prototypes: A prototype uses a "graph of instances" to describe a class. If an instance is a member of that class, then the set of facts (which the graph denotes) hold for that instance.

Importantly, the graph does not have to include *every* fact true of class members, just some subset (as chosen by the knowledge engineer). Multiple prototypes can be used to describe different aspects of class members.

Theories: KM's global KB can be partitioned into "theories" using the `(in-theory ...)` mechanism (see KM Release Notes). Theories can be made visible or hidden, and different theories can describe different aspects of the same class.

View Classes: Use classes to represent different views (e.g., have classes `Airplane-Physical-View`, `Airplane-Control-Surface-View`, etc.), and then control application of these

[1] We ignore the singular/plural distinction for this working note

[2] Or, to be fair, we should query all slots, as there is no justification for picking just the `parts` slot.

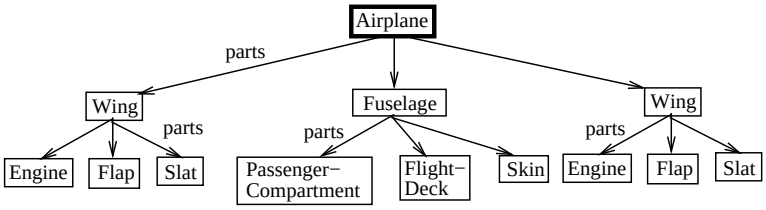
"view classes" by selectively installing "isa" links, or by using an "adjunct instance" approach [Steimann, 2000].

2.2 An Example

For the rest of this note, we will use prototypes to encode views. As a toy example, consider three prototypes, encoding three different views of an airplane:

1. A Structural View

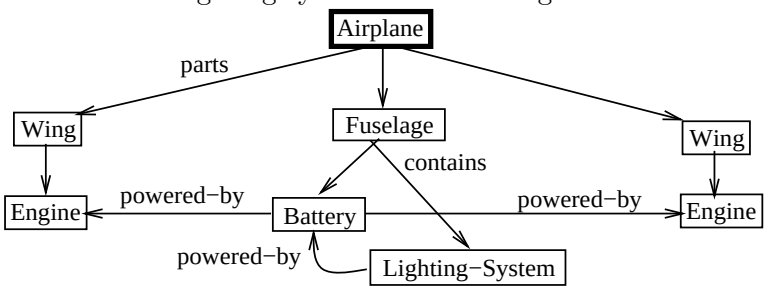
This view encodes a (very) simplified representation of the top-level partonomic structure of an airplane, shown below sketched as a graph and written in KM prototype notation. It simply declares that an airplane has two wings (each with an engine, a flap, and a slat), and a fuselage, and that the fuselage has a passenger compartment, a flight deck, and a skin.




```
(a-prototype Airplane)                    ;;; Structural view

((the Airplane) has
  (view-text ("Structural view of an airplane")))

((the Airplane) has
   (parts ((a Fuselage) (a Wing) (a Wing) (a Tail))))

((the Fuselage) has
  (parts ((a Flight-Deck) (a Passenger-Compartment) (a Skin))))

(forall (every Wing)
        (It has (parts ((a Engine) (a Flap) (a Slat)))))

(end-prototype)
```

2. An Electrical View

This view encodes that a battery, in the fuselage, is powered by the two engines in the airplane, and drives the lighting system of the fuselage.

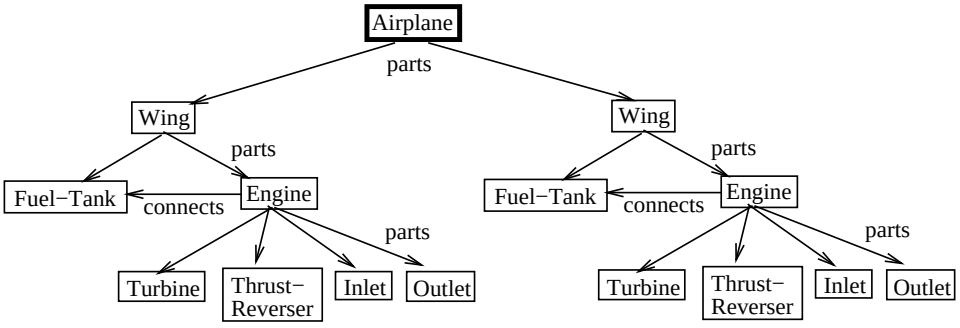



```
(a-prototype Airplane)                    ;;; Electrical view

((the Airplane) has
  (view-text ("Electrical view of an airplane")))
```

```
((the Airplane) has
  (parts ((a Wing) (a Wing)
          (a Fuselage with
             (contains ((a Battery)))))))

(forall (every Wing)
  (It has (parts ((a Engine)))))

((the Battery) has
  (powered-by ((every Engine))))

((the Fuselage) has
   (contains ((a Lighting-System with
                (powered-by ((the Battery)))))))

(end-prototype)
```

3. A Propulsion View

This view encodes details about the airplane engines, namely that each engine's parts are a turbine, a thrust reverser, an inlet, and an outlet, and that the engine is connected to the engine's wing's fuel tank.




```
(a-prototype Airplane)                  ;;; Propulsion view

((the Airplane) has
  (view-text ("Propulsion view of an airplane")))

((the Airplane) has
  (parts ((a Wing) (a Wing))))  ; don't care about fuselage

(forall (every Wing)
  (It has
     (parts ((a Fuel-Tank)
             (a Engine with
                (parts ((a Turbine) (a Thrust-Reverser) (a Inlet) (a Outlet)))
                (connects ((the Fuel-Tank parts of Self))))))))

(end-prototype)
```

2.3 A Scene Elaboration Algorithm

Consider now the original task of elaborating the scene to include a thrust reverser. A simple formulation of the task would be to ask: Are there any prototypes for any objects

in the scene that include a thrust reverser? If so, add them into the scene. An algorithm to do this would look as follows:

```
PROCEDURE: (find-target scene class)
-----------------------------------
GIVEN: a scene
RETURN: the scene, elaborated to include instance(s) of class.

ALGORITHM:
IF the scene contains an instance of class, then STOP
ELSE:
  Forall instances i in the current scene
    Forall prototypes p of i
      Does prototype p contain an instance of class?
      If yes, unify the "instance graph" for p with i
    End forall
  End forall
```

This was implemented as a single Lisp procedure, called `find-target` (the 'scene' parameter is not passed as an argument, rather the scene is the current state of KM's working memory). The following is a trace of this procedure in use (at the third `KM>` prompt below):

```
;;; Start with an empty scene
KM> (new-context)

;;; Now introduce an airplane to the scene (a KM context)
KM> (a Airplane)
(_Airplane54)
```

If we were to graph all the objects in the current context, we would just get a single node:

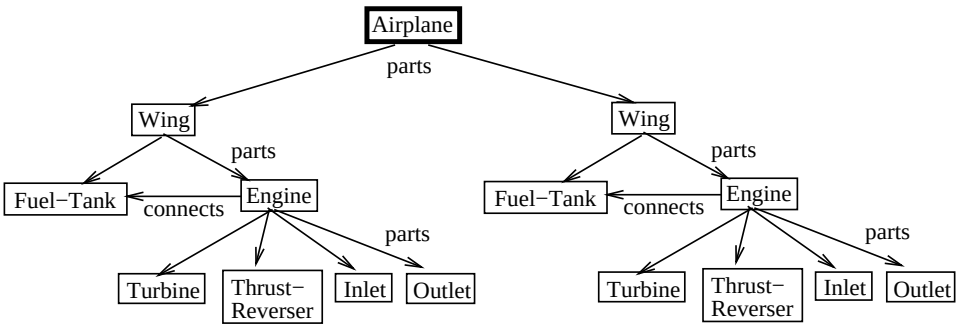
Now, let us state we want to add a thrust reverser into the scene (thus modeling an attempt to find the referent for "the thrust reverser"). We do this by calling the above algorithm, which searches for prototypes of (the classes of) things in the current context (here just `_Airplane54`) which include at least one thrust reverser, and hence would contribute a thrust reverser to the current context if unified in. In this case, just one prototype is found, and the user confirms he/she wants it added. As a result, (a clone of) the prototype is added to the scene.

```
KM> (find-target Thrust-Reverser)
Prototype _ProtoAirplane23 for Airplane
        (("Propulsion view of an airplane")) can supply a Thrust-Reverser!
Add it in (y or n)? y
(COMMENT: Cloned _ProtoAirplane23   -> _Airplane55
                                        to find all info about _Airplane54)
(COMMENT: (_Airplane54 & _Airplane55) unified to be _Airplane54)
(_Thrust-Reverser61 _Thrust-Reverser67)
```

As a result, the new context contains (clones of) all the objects from this prototype, plus the original airplane instance. (The relations between these objects aren't listed here, but can be explored by `(showme ...)` commands).

```
KM> (show-context)
   _Outlet69
   _Inlet68
   _Thrust-Reverser67
   _Turbine66
   _Outlet63
   _Inlet62
   _Thrust-Reverser61
   _Turbine60
   _Engine59
   _Fuel-Tank58
   _Engine65
   _Fuel-Tank64
   _Wing57
   _Wing56
   _Airplane54
```

If the instances and relations were sketched graphically, they would look as follows:



Now consider a second query, to find a fuel tank (modeling resolution of the phrase "the fuel tank"). In this case, as there is already a fuel tank in the scene, no processing is necessary and those fuel tank(s) are returned as the targets.

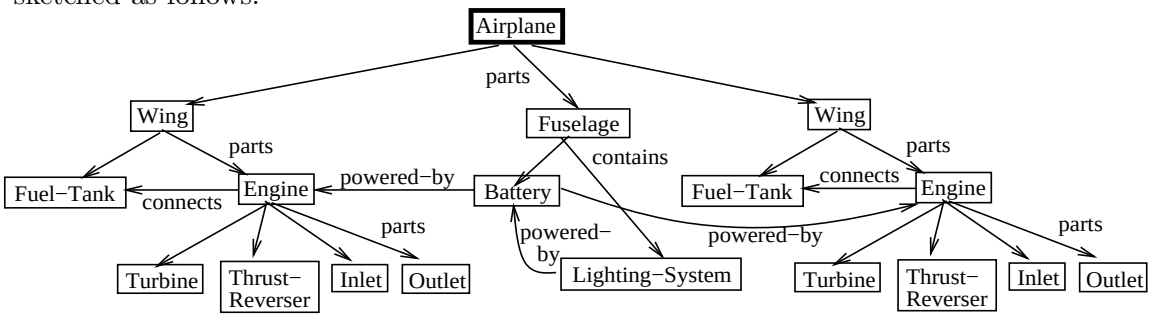
```
KM> (find-target Fuel-Tank)
(_Fuel-Tank74 _Fuel-Tank80)
```

Note that (although not shown here) these two returned fuel tanks are the fuel tanks inside the wings of this airplane, not some arbitrary fuel tank instances.

Finally, consider a third query for a battery (modeling resolution of the phrase "the battery"). There is no battery in the current scene yet, so KM goes looking for prototypes which could be added to the scene to supply one. The prototype representing the electrical view of the airplane can supply this, so it's added in.

```
KM> (find-target Battery)
Prototype _ProtoAirplane15 for Airplane
        (("Electrical view of an airplane")) can supply a Battery!
Add it in (y or n)? y
(COMMENT: Cloned _ProtoAirplane15   -> _Airplane86
                                          to find all info about _Airplane70)
(COMMENT: (_Airplane70 & _Airplane86) unified to be _Airplane70)
(_Battery90)

KM>
```

The final scene is now the unification of the two views (propulsion and electrical) which were added to the scene, plus the original node denoting the airplane. This can be sketched as follows:



3 Discussion

The example in this note is very simple, and obviously needs a lot more development. In particular:

Plausible Reasoning: Given an airplane, the presented algorithm just adds some subset of facts (via prototype selection) known to hold for *all* airplanes. But more generally, we might want to add in sets of facts just known to hold for *some* airplanes. This requires making the abductive inference that the current airplane is of this specialized type.

For example, we know that some types of airplanes have thrust reversers, while others do not. Let us hypothesize that `Jet-Airplane`s have thrust reversers, while `Propeller-Airplane`s do not. So if a reference to "the thrust reverser" appears, the system should guess that the airplane being discussed is probably one of those which has thrust reversers (i.e., is a jet airplane). Implementationally this means that, when searching for applicable prototypes, the system should also look at *specializations* of existing objects' classes in the scene. For example, when searching for prototypes that can contribute a thrust reverser, the system should not just look at prototypes of `Airplane`, but also at `Jet-Airplane` and `Propeller-Airplane`. If one of those prototypes can supply a thrust reverser, then the prototype is a candidate for adding to the scene, along with the plausible inference that the airplane in question is actually a member of that specialization.

Controlling Scene Elaboration: In this note, we treat a prototype as relevant if it can can supply a object of a desired type. In general, however, this is too unconstrained. Consider a reference to "the pump": there are many pumps on an airplane, but as a reader we prefer (given the text is about hydraulics) an interpretation in which the pump is a hydraulic pump, even if there are other possible views which also contain pumps (e.g., the propulsion view of the airplane might include a fuel pump). Some means of assessing relevance is required to control this elaboration process properly.

References

[Clark and Porter, 1999] Clark, P. and Porter, B. (1999). KM – the knowledge machine: Users manual. Technical report, AI Lab, Univ Texas at Austin. (http://www.cs.utexas.edu/users/mfkb/km.html).

[Steimann, 2000] Steimann, F. (2000). On the representation of roles in object-oriented and conceptual modeling. *Data and Knowledge Engineering*, 35(1):83–106.

[Zadrozny and Jensen, 1991] Zadrozny, W. and Jensen, K. (1991). Semantics of paragraphs. *Computational Linguistics*, 17(2):171–209.