Translating from CCALC into KM: An Example

Peter Clark
Boeing Company
peter.e.clark@boeing.com
and
Joohyung Lee, Vladimir Lifschitz and Bruce Porter
University of Texas at Austin
{appsmurf,vl,porter}@cs.utexas.edu

September 15, 1999

KM (Knowledge Machine)[1] and CCALC (Causal Calculator)[2] are knowledge representation systems designed and implemented at the University of Texas at Austin. KM is a frame-based system; CCALC is an implementation of a causal logic closely related to logic programming. Although the two systems are very different from each other, a representation of a collection of facts in one of them can be sometimes translated into the other in a modular, straightforward way.

In this note we show how this can be done for the problem of representing the following collection of assertions:

A VW Rabbit is a VW.
Tom's car is a VW Rabbit.
Dick's car is a VW Rabbit.
A VW has an electrical system.
Part of the electrical system is an alternator.
The alternator is defective on every VW.

This is a modification of an example from [3] (pp. 293–297). Michael Gelfond and Monica Nogueira offered this problem as an exercise to Texas Action Group[3], and Esra Erdem responded with a formalization in the language of CCALC. Figures 1–5 show a "translation" of that formalization into the language of KM, with the original CCALC representation shown in right column as a sequence of comments.

The CCALC representation has a few elements that do not correspond to anything in the English language asser-

[1] `http://www.cs.utexas.edu/users/mfkb/km.html` .
[2] `http://www.cs.utexas.edu/users/mccain/cc` .
[3] `http://www.cs.utexas.edu/tag` .

tions above, such as the sorts `toyota` and `airCondSystem` and the binary predicate constant `isWrong`. (That symbol can be used to ask questions of the form "What's wrong with `X`?" Any minimal defective part of `X` would be an acceptable answer.) These elements have been incorporated in Erdem's formalization in response to questions from Gelfond and Nogueira.

The right column of Figure 1 begins with including the standard file `lp`, because CCALC is used here in the "logic programming mode." That file defines negation as failure in terms of the causal logic from [2]. This directive is followed by the declaration of eight sorts that form a tree structure, with the most general sort `object` as the root. The translation into KM uses `superclasses` as the counterpart of the subsort symbol `>>` of CCALC.

The variable declarations of CCALC are not translated into KM at all—there are no variables in KM, although some constructs available in KM do correspond to the use of bound variables in classical logic. (In particular, the keyword `It` plays the role of a variable.)

At the end of the right column we see three constant declarations. They are translated into KM in a straightforward way, using `instance-of`.

The right column of Figure 2 shows the declarations of three function constants. The symbol `es` represents the function that turns every Volkswagen into its electric system. In KM, it turns into a slot, that is, binary predicate—the graph of the corresponding function. By imposing the condition `cardinality (1-to-1)` we make this predicate the graph of a partial function; by adding that every Volkswagen has an electric system, we make this function total. The declarations of `alt` and `ac` are translated into KM in a similar way.

In Figure 3, declarations of predicate constants—both binary and unary—turn into slot declarations.

The right column of Figure 4 begins with logic programming definitions of the binary predicates `part` (as the transitive closure of the "immediate part" relation `part0` defined in Figure 5) and `isWrong`. As proposed in [1], these rules distinguish between negation as failure (`not`) and classical negation (`-`). The first two rules provide sufficient conditions for `part`, and the third rule expresses the closed world assumption (CWA): `part(X,Y)` is false if there is no evidence to the contrary. The definition of `isWrong` has a similar structure. The CWA for binary predicates is built in the semantics of KM, so that the CWA rules do not need to be translated.

Note the difference between the KM representations of the atoms `defective(X)` and `partsOK(X)` occurring in the logic programming definition of `isWrong`: the former is translated as

$$(\texttt{the defective of It}) \ = \ \texttt{t}, \qquad (1)$$

and the latter as

$$(\texttt{the partsOK of It}) \ / = \ \texttt{f}. \qquad (2)$$

This is related to the difference between the representation methods used here for `defective` and `partsOK`, which is discussed below.

The remaining rules in the right column define two unary predicates: `partsOK` and (in combination with the last rule of Figure 5) `defective`. The definition of `partsOK` differs from the others in that it gives a sufficient condition for the *negation* of the predicate and is followed by the "inverse" closed-world assumption: `partsOK(X)` is *true* if there is no evidence to the contrary.

Our KM formulation reflects this difference by using different representation conventions for the predicates `defective` and `partsOK`. The former is represented by the set of all pairs formed from a defective object and the truth value `t`. The latter is represented by the set of the pairs formed from an object that does not have the property in question and the truth value `f`. This difference explains why we write `= t` in (1) and `/= f` in (2).

The translations of the logic programming definition of `part0` and of the last rule for `defective` in Figure 5 are similar to the translations shown in Figure 4.

Our aim here has been to produce, as far as possible, a "literal" translation of the CCALC formalization in KM. However, the result is not necesarily the best stylistically (although here it seems fairly parsimonious). For example, instead of distributing statements about the car's partonomy among the slots `alt`, `es` and `ac`, we could have gathered them as a collection of nested frames on the `part0` slot (where the nesting reflects the partonomic hierarchy).

```
(every vw has
    (part0 ((a electricSystem with
               (part0 ((a alternator with
                          (defective (t)))))))))
```

Then, the `alt`, `es` and `ac` slots could either contain KM paths pointing to the respective parts on the `part0` slot, e.g., `the electricSystem part0 of Self`, or be omitted entirely.

As a note of interest, this issue is not specific to CCALC and KM, but often arises in any language-to-language translation: literal translations do not necessarily produce results which follow the target language's idiomatic style, are parsimonious and easily comprehensible, and which the target language's inference engine can handle efficiently (or handle at all). This fact has been a major obstacle for defining automatic language-to-language translators [4].

References

[1] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[2] Norman McCain and Hudson Turner. Causal theories of action and change. In *Proc. AAAI-97*, pages 460–465, 1997.

[3] Neil C. Rowe. *Artificial Intelligence Through Prolog.* Prentice Hall, 1988.

[4] Michael Uschold, Michael Healy, Keith Williamson, Peter Clark, and Steve Woods. Ontology reuse and application. In Nichola Guarino, editor, *Proc. Int'l Conf. on Formal Ontology in Information Systems—FOIS'98 (Frontiers in AI and Applications, Vol. 46)*, pages 179–192, Amsterdam, 1998. IOS Press. (http://www.cs.utexas.edu/users/pclark/papers).

```
                                                  ; :- include 'lp'.
                                                  ;
                                                  ; :- sorts
                                                  ;
(car has (superclasses (object)))                 ;   object >> ((car
(vw has (superclasses (car)))                     ;                >> ((vw
(vwRabbit has (superclasses (vw)))                ;                    >> vwRabbit);
(toyota has (superclasses (car)))                 ;                     toyota));
(electricSystem has (superclasses (object)))      ;               electricSystem;
(alternator has (superclasses (object)))          ;               alternator;
(airCondSystem has (superclasses (object)))       ;               airCondSystem).
                                                  ;
                                                  ; :- variables
                                                  ;
                                                  ;         X, Y, Z :: object;
                                                  ;         C1 :: car;
                                                  ;         VW :: vw;
                                                  ;         AC :: airCondSystem;
                                                  ;         ES :: electricSystem;
                                                  ;         Alt ::  alternator.
                                                  ;
                                                  ; :- constants
                                                  ;
(mikesCar has (instance-of (toyota)))             ;         mikesCar :: toyota;
(dicksCar has (instance-of (vwRabbit)))           ;         dicksCar :: vwRabbit;
(tomsCar has (instance-of (vwRabbit)))            ;         tomsCar :: vwRabbit;
```

Figure 1: Volkswagen Domain, Part 1

```
(es has                                         ;        es(vw) :: electricSystem;
   (instance-of (Slot))                         ;
   (cardinality (1-to-1)))                      ;
                                                ;
(every vw has                                   ;
   (es ((a electricSystem))))                   ;
                                                ;
(alt has                                        ;        alt(vw) :: alternator;
   (instance-of (Slot))                         ;
   (cardinality (1-to-1)))                      ;
                                                ;
(every vw has                                   ;
   (alt ((a alternator))))                      ;
                                                ;
(ac has                                         ;        ac(car) :: airCondSystem;
   (instance-of (Slot))                         ;
   (cardinality (1-to-1)))                      ;
                                                ;
(every car has                                  ;
   (ac ((a airCondSystem))))                    ;
```

Figure 2: Volkswagen Domain, Part 2

```
(part0 has                                   ;       part0(object,object) :: atomicFormula;
    (instance-of (Slot)))                    ;
                                             ;
(part has                                    ;       part(object,object) :: atomicFormula;
    (instance-of (Slot)))                    ;
                                             ;
(isWrong has                                 ;       isWrong(object,object) :: atomicFormula;
    (instance-of (Slot)))                    ;
                                             ;
(partsOK has                                 ;       partsOK(object) :: atomicFormula;
    (instance-of (Slot)))                    ;
                                             ;
(defective has                               ;       defective(object) :: atomicFormula;
    (instance-of (Slot)))                    ;
```

Figure 3: Volkswagen Domain, Part 3

```
(every object has                                     ;
    (part ((the part0 of Self)                        ; part(X,Z) <- part0(X,Z).
       (the part of (the part0 of Self))))            ; part(X,Z) <- part0(X,Y), part(Y,Z).
                                                      ;
                                                      ; -part(X,Y) <- not part(X,Y).
                                                      ;
    (isWrong ((allof (the part of Self)               ; isWrong(X,Y) <- part(X,Y),
        where  (((the defective of It) = t)           ;                 defective(X),
               and ((the partsOK of It) /= f)))))     ;                 partsOK(X).
                                                      ;
                                                      ; -isWrong(X,Y) <- not isWrong(X,Y).
                                                      ;
    (partsOK ((if ((the defective of                  ; -partsOK(Z) <- part(X,Z),
                      (the part of Self)))            ;                defective(X).
               then f)))                              ;
                                                      ;
                                                      ; partsOK(X) <- not -partsOK(X).
                                                      ;
    (defective ((the defective of                     ; defective(X) <- part(Z,X),
                   (the part of Self)))))             ;                 defective(Z).
                                                      ;
                                                      ; -defective(X) <- not defective(X).
```

Figure 4: Volkswagen Domain, Part 4

```
(every vw has                                       ; part0(es(VW),VW) <- true.
   (part0 ((the es of Self))))                      ;
                                                    ;
(every vw has                                       ; part0(alt(VW),es(VW)) <- true.
   (es ((a electricSystem with                      ;
            (part0 ((the alt of Self)))))))         ;
                                                    ;
(every car has                                      ; part0(ac(C1),C1) <- true.
   (part0 ((the ac of Self))))                      ;
                                                    ;
                                                    ; -part0(X,Y) <- not part0(X,Y).
                                                    ;
(every vw has                                       ; defective(alt(VW)) <- true.
   (alt ((a alternator with (defective (t))))))     ;
```

Figure 5: Volkswagen Domain, Part 5