# A Brief Note on Implementing Situations in KM

Peter Clark
Knowledge Systems
Boeing Math and Computing Technology
Seattle, USA
peter.e.clark@boeing.com

## 1 Representing Situations

KM's implementation of situations is a special case of using *contexts* [1]. Intuitively, a context (in its general sense) is a theory which is (by default) isolated from other theories – in KM, a context can be thought of as a partition of the knowledge-base which is (by default) isolated from other parts of the knowledge-base. A set of *lifting axioms* (also called *articulation axioms*) then define how axioms from one context can be transferred to other contexts.

Contexts are a very general mechanism which can be used in many ways. For example, contexts might represent an agents' beliefs: for instance, if context $C_{Joe}$ represent "Joe's beliefs", and Joe believes that Sue has Book1, then $C_{Joe}$ would contain the assertion `has(Sue,Book1)`. Lifting axioms would state if/how propositions can be transferred to other contexts, for example, a lifting axiom might state that proposition `P` in $C_{Joe}$ becomes `believes(Joe,P)` in a 'base' context [2].

KM uses contexts to implement the notion of *situations*, as used in situation calculus. A *situation* is a context $C$ subject to two specific lifting axioms:

**importing:** all axioms in $C$'s super-situation(s) also hold in $C$.

**projection:** if no values for a frame's slot can be found in $C$, then that frame's slot acquires the values from the previous situation $C_{prev}$.

where 'super-situation' and 'previous situation' are relations between situations. If we define $\Delta_{LocalC}$ as the axioms explicitly stated in $C$, $\Delta_{SuperC}$ as the axioms in $C$'s super-situation, and $\Delta_{PrevC}$ as the axioms in $C$'s previous situation, then the full set of axioms $\Delta_C$ visible in context $C$ are approximately[1]:

$$\begin{array}{lcll} \Delta_C & = & \Delta_{LocalC} & \\ & \cup & \Delta_{SuperC} & (import) \\ & \cup & \{\ s(f,v) \mid \neg\exists v'\ (\ \Delta_{LocalC} \cup \Delta_{SuperC} \vdash s(f,v')\ )^2 & (if\ no\ values... \\ & & \wedge\ (\ \Delta_{PrevC} \vdash s(f,v)\ )\ \} & ...then\ project) \end{array}$$

These lifting axioms are hard-wired into KM's inference machinery. The arrangement of situations and their relations are depicted in Figure 1.

[1] This is a deliberate simplification: Strictly, when trying to find values for a frame's slot before resorting to projection, the algorithm will use the axiom set $\Delta_{LocalC} \cup \Delta_{SuperC} \cup \Delta_{AlreadyProjected}$, where $\Delta_{AlreadyProjected}$ are the axioms already projected from the previous context.

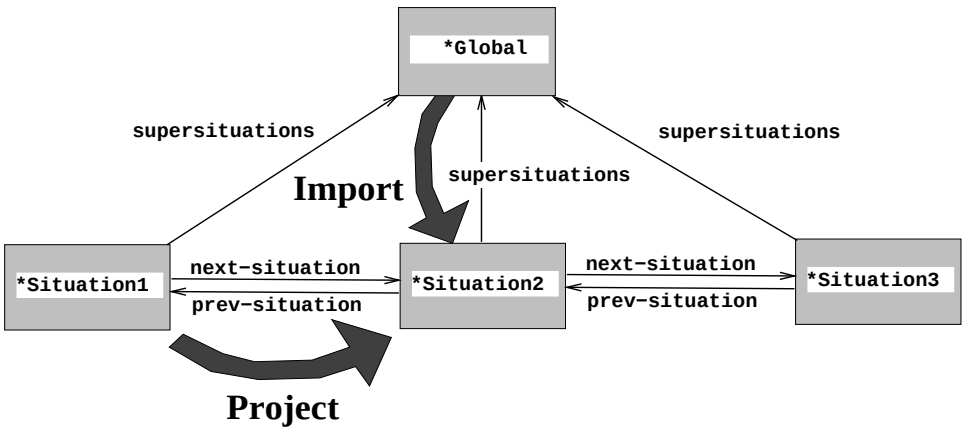


Figure 1: Situations and their relationships in KM.

## 2 Physical Implementation

Despite the many ways of using situations, as reflected by the size of this manual, the implementation of situations in KM is simple and straightforward. We now overview this implementation.

In the global situation, KM stores information about a concept on a property list which we will here call `properties`[3]. Slot-values are read and written to this property list. In a local situation, say `_Situation1`, a situation-specific property list is used instead, say `_Situation1-properties`. To read a frame's slot, KM will first try the information on the `_Situation1-properties` list for that frame, and if no information is found, it will try the global `properties` list for that frame instead (this implements importing from the global to local situation in a demand-driven fashion). To write a frame's slot, KM will always write to the `_Situation1-properties` list. Thus all computation in a local situation is stored on that situation's specific property list, and moving out of that situation will make it inaccessible (as a new situation will be reading and writing from a different property list). In this way, situation-specific information is retained but hidden from the rest of the KB.

Projection is also implemented in a simple way: If no values for a frame's slot can be found in a local situation (including using rules imported from the global situation), then KM will change to the previous situation and try the computation there. If successful, KM will then return to the original situation and assert the result there.

It is important to note that the machinery for importing and projection differ. Importing involves *copying* axioms (ie. slot value expressions) from the global to a local situation, *without* evaluating them in the global situation – they are only evaluated *after* they have been imported, in the local situation. In contrast, projection involves *evaluating* a query in a previous situation – axioms (ie. slot value expressions) are *not* themselves transferred from a previous situation to a next situation, but used to try to answer the query in that previous situation. Only the answers to that query – the computed 'ground facts' – are then copied to the new situation. This distinction is primarily made for efficiency reasons, to avoid redundant computation in multiple situations.

[3]This is a simplification – in reality, four property lists are used, called `own-properties`, `member-properties`, `own-definition`, and `member-definition`, storing assertional properties about a class and its members, and definitional properties about a class and its members, respectively. For the purposes of this description, though, imagine just a single property list is used.

## 3 A Simplified Implementation in Prolog

For Prolog enthusiasts, a simplified version of this algorithm can be concisely expressed in Prolog, and is given in Appendix A.

## References


[1] Univ. Maine. Context in artificial intelligence (web site). (http://cdps.umcs.maine.edu/Context/index.shtml), 1998.

[2] Alessandro Cimatti and Luciano Serafini. Multiagent reasoning with belief contexts ii: Elaboration tolerance. In *Proc AAAI-95 Fall Symposium on Formalizing Context*. AAAI, 1995. (http://www-formal.stanford.edu:80/buvac/95-context-symposium).

## Appendix A: Prolog version of the KM Implementation

For Prolog enthusiasts, a simplified version of KM's situation mechanism can be concisely expressed in Prolog, and is given below:

```
:- dynamic in_situation/2, blocked_projection/1, complete/1.

in_situation(S, (A,B)) :-                       % (A AND B) <- A AND B
        in_situation(S, A),
        in_situation(S, B).
in_situation(S, A) :-                           % A <- (A<-B) AND B
        A \= (_:-_),
        in_situation(S, (A:-B)),
        in_situation(S, B).

in_situation(S, A) :-                                   % PROJECTION
        S \= global,
        simple_term(A),                                 % don't project clauses...
        \+ complete(A),                                 % or indirect effects...
        in_situation(global, prev_situation(S,PrevS)),   % find prev situation
        in_situation(PrevS, A),                         % value in prev situation?
        \+ in_situation(S, blocked_projection(A)),       % projectable?
        writef("TRACE: Projected %w from %w to %w.\n", [A,PrevS,S]).

in_situation(S, A) :-                                   % INCLUSION
        S \= global,
        in_situation(global, A),
        writef("TRACE: Included %w from %w to %w.\n", [A,global,S]).

simple_term(A) :- A \= (_:-_), A \= (_,_), !.

% --------------------

% Form for specifying an action:
% action(+Name, +Del_list, +Add_list).

% Doing an action...
do_action(Action, OldSitn, NewSitn) :-
        action(Action, DelList, AddList),
        assert( in_situation(global, prev_situation(NewSitn,OldSitn)) ),
        del_facts(DelList, NewSitn),
        add_facts(AddList, NewSitn),
        writef("Done!\n"),
        !.

% DELETE is implemented by blocking projection of the facts to the new sitn
del_facts([], _).
del_facts([F|Fs], Sitn) :-
        assert( in_situation(Sitn, blocked_projection(F)) ),
        del_facts(Fs, Sitn).

% ADD asserts the facts in the new situation
```

```
add_facts([], _).
add_facts([F|Fs], Sitn) :-
        assert( in_situation(Sitn,F) ),
        add_facts(Fs, Sitn).

% ========================================================================
%               Domain-specific example
% ========================================================================

% GLOBAL FACTS
in_situation(global, controlled_by(light1,switch1) ).
in_situation(global, ( brightness(L,bright) :-
                        controlled_by(L,S),
                        position(S,up) )).
in_situation(global, ( brightness(L,dark) :-
                        controlled_by(L,S),
                        position(S,down) )).

% BRIGHTNESS is a derived (indirect) effect
complete(brightness(_,_)).

% DEFINE AN ACTION
% action(+Name, +Del_list, +Add_list).
action(switching_on(S), [position(S,down)], [position(S,up)]).

% --------------------
% LOCAL FACTS: INITIAL SITUATION
in_situation(sitn1, position(switch1,down)).

/* Demo:
| ?- do_action(switching_on(switch1), sitn1, sitn2).
| ?- in_situation(sitn2, brightness(light1,X)).
X = bright
*/
```