Appendix D - PROLOG implementation

```
kw_source(definit, html, "file1.html", [autonom, ascript, descript, mean, term, approach] , "file1.html").
kw_source(agent, ol, "file1.html#elm1", [autonom, ascript, descript, mean, sometim, attribut] , "file1.html").
kw_source(agent, html, "file2.html", [characterist, attribut, reactiv, proactiv, model, adapt] , "file2.html").
kw_source(reactiv, ul, "file2.html#elm1", [proactiv, model, adapt, autonom, knowledg, collabor] , "file2.html").
kw_source(autonom, ol, "file2.html#elm3", [intellig, degre, machin, margin, top] , "file2.html").
kw_source(interfac, p, "file2.html#elm4", [interfac, collabor, nana, classif, result, type] , "file2.html").
kw_source(dictionari, p, "file2.html#elm5", [act, behalf] , "file2.html").
kw_source(softwar, html, "file3.html", [interfac, intellig, user, direct, manipul] , "file3.html").
kw_source(intellig, ol, "file3.html#elm1", [user, interfac, agent, cooper, simplifi, distribut] , "file3.html").
kw_source(agent, table, "file3.html#elm2", [interfac, user, proactiv, reactiv, task, search] , "file3.html").
kw_source(debat, html, "file4.html", [mae, user, interfac, ben, adapt, domain] , "file4.html").
kw_source(user, ul, "file4.html#elm1", [agent, interfac, adapt, intellig, model, futur] , "file4.html").
kw_source(agent, ul, "file4.html#elm2", [user, domain, interfac, focus, speech, difficult] , "file4.html").
kw_source(foley, html, "../341/foley.html", [norman] , "../341/foley.html").

lowerList("file4.html", [speech, futur, direct, manipul, design, system]).
lowerList("file4.html#elm1", [design, system, speech, memori, direct]).
lowerList("file4.html#elm2", [softwar, disagr, mainli, due, ben]).


href("file1.html", "file2.html", "attributes").
href("file1.html", "file3.html", "why software agents").
href("file1.html", "file4.html#pattie", "pattie maes").
href("file2.html", "file3.html", "acts on your behalf").
href("file3.html", "file4.html", "direct vs interface agents").
href("file4.html", "file3.html", "software agents ").
href("file4.html", "../341/foley.html", "according to foley").
name("file4.html", "pattie", "pattie").


make_concept :-
     kw_source(ConceptName, _ , _ , _ , _ ) ,
     assert( concept(ConceptName) ) ,
     fail.
     % fail forces backtracking so that every combination is tried

make_concept.
     %when everything is tried and the above rule fails, this one succeeds

%-----------------------------------------------------------------------
% HR-1: Hyper references between two knowledge entities somewhere within
% the domain are relations of type has_deeper_explanation
% between the corresponding concepts.
%---

hr1 :-
     href(From, To, _ ) ,
     kw_source(ConceptA, _ , From, _ , _ ) ,
     kw_source(ConceptB, _ , To, _ , _ ) ,
     assert( relation(has_deeper_explanation, ConceptA, ConceptB) ) ,
     fail.    % forces backtracking

hr1. % ensure success


%-----------------------------------------------------------------------
% member is recursively defined and finds out whether an X is in a list.
% It is used thoroughly in the following.
%---

member(X, [X | Tail]).
member(X, [Head | Tail]) :-
     member(X, Tail).

%-----------------------------------------------------------------------
% The domain is constrained to a list of valid filenames, here exemplified
% with three files only. The rule outsideDomain checks for membership in
% the domain list.
%---

domain( ["file1.html", "file2.html", "file3.html", "file4.html"] ).

outsideDomain(F) :-
     domain(D) ,
     not member(F, D).


%-----------------------------------------------------------------------
% HR-2: External links with destinations outside of the domain
% are slightly different from internal ones, and the corresponding
% relations should be labelled as external.
%---

hr2 :-
href(From, To, _ ) ,
     outsideDomain(To) ,
     kw_source(ConceptA, _ , From, _ , _) ,
     kw_source(ConceptB, _ , To, _ , _) ,
     assert( relation(external, ConceptA, ConceptB) ),
     fail.    % forces backtracking

hr2. % ensure success
```

```
%------------------------------------------------------------------------
% HR-3: All element concepts found within a document are children
% of the document concept.
%---

hr3 :-
     kw_source(DocumentConcept, html, _ , _ , File) ,
     kw_source(ElementConcept, _ , _ , _ , File) ,
     not DocumentConcept = ElementConcept ,
     assert( relation(parent, DocumentConcept, ElementConcept) ) ,
     fail.    % forces backtracking

hr3. % ensure success


%------------------------------------------------------------------------
% member is recursively defined and finds out whether an X is in a list.
% It is used thoroughly in the following.
%---

member(X, [X | Tail]).
member(X, [Head | Tail]) :-
     member(X, Tail).

%------------------------------------------------------------------------
% HR-4: There is a parent relation if a member from a set of
% candidate concepts equals an already found concept.
%---

hr4 :-
     kw_source(ConceptA, _ , _ , CandidateList, _ ) ,
     kw_source(ConceptB, _ , _ , _ , _ ) ,
     not ConceptA = ConceptB ,   %  the two concepts should not be equal
     member(ConceptB, CandidateList) ,
     not relation(parent, ConceptA, ConceptB) , %only consider once
     assert( relation(parent, ConceptA, ConceptB) ) ,
     fail.    % forces backtracking

hr4. % ensure success



%------------------------------------------------------------------------
% Removes bi-directional relations
%---

removeBidirectional :-
     relation(Type, ConceptA, ConceptB) ,
     relation(Type, ConceptB, ConceptA) ,
     retract( relation(Type, ConceptB, ConceptA) ) ,  %remove one
     fail.    % forces backtracking

removeBidirectional. % ensure success




%------------------------------------------------------------------------
% HR-5: If two concepts have some joint members from their
% set of candidates, the concepts are synonymous.
%---

commonMember(List1, List2) :-
     member(Term, List1) ,
     member(Term, List2).

hr5 :-
     kw_source(ConceptA, _ , _ , CandidateListA, _ ) ,
     kw_source(ConceptB, _ , _ , CandidateListB, _ ) ,
     commonMember(CandidateListA, CandidateListB) ,
     not ConceptA = ConceptB ,
     not relation(synonym, ConceptA, ConceptB) ,
     assert( relation(synonym, ConceptA, ConceptB) ) ,
     fail.    % forces backtracking

hr5. % ensure success


make_concept :-
     kw_source(ConceptName, _ , _ , _ , _ ) ,
     assert( concept(ConceptName) ) ,
     fail.
     % fail forces backtracking so that every combination is tried

make_concept.
     %when everything is tried and the above rule fails, this one succeeds

%------------------------------------------------------------------------
% member is recursively defined and finds out whether an X is in a list.
% It is used thoroughly in the following.
%---

member(X, [X | Tail]).
member(X, [Head | Tail]) :-
     member(X, Tail).
```

```
%--------------------------------------------------------------------------
% Lower score terms are listed as PROLOG facts, here illustrated with
% three examples. Note that the lists in lowerList and kw_source
% together constitute all the terms surviving lexical analysis.
%---

lowerList("file4.html", [speech, futur, direct, manipul, design, system]).
lowerList("file4.html#elm1", [design, system, speech, memori, direct]).
lowerList("file4.html#elm2", [softwar, disagr, mainli, due, ben]).

inLowerList(Term, Location) :-
     lowerList(Location, Z) ,
     member(Term, Z).


%--------------------------------------------------------------------------
% The domain specific list contains terms that are regarded important,
% here exemplified with seven terms only.
%---

domainSpecificList( [hci, adapt, iui, user, model, domain, intellig] ).

inDsl(Term) :-
     domainSpecificList(DSL) ,
     member(Term, DSL).


%--------------------------------------------------------------------------
% HR-6: A term in a knowledge source KS_1 which is neither selected as
% concept nor in the upper list of candidates, yet a member of the DSL
% and chosen as concept for another knowledge source KS_2,
% is prerequisite to the concept of the first knowledge source KS_1.
%---

hr6 :-
     inLowerList(Term, Location) ,
     inDsl(Term) ,
     kw_source(Concept , _ , Location, _ , _ ) ,
     concept(Term) ,
         %does the concept of the source equals the term?
     assert( relation(prerequisite, Term, Concept) ) ,
         %if so, then make a relation between the two.
     fail.    % forces backtracking

hr6. % ensure success


%--------------------------------------------------------------------------
% There is a path between two nodes if there is an edge from the first
% node to an intermediate node, which again has a path to the destination
% node. This code also adds the nodes to the head of a list.
%---

path(From, To, [To] ) :-
     href(From, To, _ ).

path(From, To, [Intermediate|Tail] ) :-
     href(From, Intermediate, _ ),
     path(Intermediate, To, Tail).


%--------------------------------------------------------------------------
% This rule asserts prerequisite relations from the intermediate
% members of a path to the last node
%---

addToSet([Head|Tail], ToConcept) :-
     kw_source(FromConcept, _ , Head, _ , _ ) ,
     not FromConcept = ToConcept ,  %write relation if the concepts differ
     assert( relation(prerequisite, FromConcept, ToConcept) ) ,
     addToSet(Tail, ToConcept) ,  %move down the path

addToSet([], _). %trivial case
```

```
%-----------------------------------------------------------------------
% HR-7: In the presence of a relation between two concepts, together with
% an unique, explicitly stated path through interlinked nodes connecting
% the two corresponding knowledge sources, there is a set of prerequisite
% relations between the concepts of each node (but the first one), and
% the destination of the path.
% Note that the procedure removes the original relationship type if
% some prerequisites are found
%---

hr7 :-
     relation(Type, ConA, ConB) , %if any relation exists
     kw_source(ConA , _ , From, _ , _ ) , %we go get the
     kw_source(ConB , _ , To, _ , _ ) ,   %destinations
     path(From, To, Visited) , %is there an explicitly stated path?
     addToSet(Visited, ConB) , %add all relations along this path
     retract( relation(Type, ConA, ConB) ) , %remove original relation
     fail.    % forces backtracking

hr7. % ensure success




%-----------------------------------------------------------------------
% The rule next(Concept) finds a gap and builds a bridge, that is a list
% of concepts that can be visited next. Note that this routine does not
% validate the prerequisites of the concepts in the gap.
%---

deletepossible(Item, _ , [] ).
deletepossible(Item, [Item|Tail] , Tail).
deletepossible(Item, [Y|Tail] , [Y|Tail2] ) :-
     deletepossible(Item, Tail, Tail2).

insert(X, List, BiggerList) :-    %insert is the inverse of delete
     delete(X, BiggerList, List).

findGap( [] , R, R).  %trivial case
findGap( DMList, [Head|Tail], GapList):-
     deletepossible(Head, DMList, IntermediateList) ,
             %deletes common members
     findGap(Tail, IntermediateList, GapList).

all(Concept, ListIn, ListOut) :-   %go get all neighbours
     relation( _ , Concept, Next) ,
     not member(Concept, ListIn) ,
     all(Concept, [Next|ListIn], ListOut).
all(C, List, List). %trivial case

getNeighbour(Concept, DMList) :-  %get neigbours to a concept in DM
     all(Concept, [], DMList).

userKnows(Concept, UMList) :-    %get concepts known to the user.
     all2(Concept, [], UMList).  %almost equal to all, except that
                                 %relation in all is the DM relation,
                                 %and relation i all2 is UM relation.

next(Concept) :-
     getNeighboursDM(Concept, DMList) ,   %list all neighbours from DM
     userKnows(Concept, UMList) ,  % build list of neighb. known to user
     findGap(DMList, UMList, GapList) ,  %subtract UM from DM
```