Towards a Tutor for Knowledge Representation

Adil KABBAJ, Claude FRASSON
Université de Montréal, Département d'informatique
et de recherche opérationnelle
C.P. 6128, Succ. A, Montréal, H3C 3J7

tel : 514 343 7019
fax : 514 343 5834
E-mail : Kabbajad@iro.umontreal.ca
E-mail : Frasson@iro.umontreal.ca

Abstract. We present in this paper a tutorial system for basic concepts in Knowledge Representation. For tutorial purpose, we have chosen a Conceptual Graph Theory, proposed by Sowa as a universal knowledge representation. The tutor is based on visual presentation and use of hypermedia techniques.

1. Introduction

During the last years, several tutoring systems have been designed to teach basic scholar fields such as mathematics (arithmetic, algebra, probability, symbolic integration, ...), physics and elementary programming techniques (procedural programming with Basic and Pascal, functional programming with Lisp, and more recently Logical programming with Prolog) [*** references au livres de ITS : 82, 87, 87, 88, 90, 92].

Few works were done, however, to design tutoring systems on basic concepts of Knowledge Representation, which are (basic) components of artificial intelligence (AI), and their applications. For instance, several algorithms in AI are based on knowledge structures and specific operations (such as matching). The user of these algorithms must know how the operations act on knowledge structures in order to appreciate their impact. Also, since AI techniques are applied now in various areas, not only by the computer science community but also by a wide range of people, a tutor in knowledge representation would be useful. Note that the tutor can also be used for teaching, at collegial level for instance, various cognitive concepts such as Knowledge Structures and Conceptual operations. Fundamental questions of knowledge acquisition could then find a solution through the use of this tutor. For instance, it could help in finding how a knowledge concept is retrieved by matching or merging two structures, how the meaning of a sentence can be clarified using matching operations, etc.. With such demonstrations, students will be more conscious to avoid ambiguities, will clarify their actions and, may be the most important benefit, will be motivated and more adapted to acquire in depth knowledge in trying to structure more adequately their own knowledge.

Building a tutor in knowledge representation seems however 'very hard' since they are many paradigms (Semantic Networks, Production Systems, Frame systems, ..) and many proposals for each paradigm. For instance, in [Rodin, 1992] eight family of Semantic Networks are presented. However, the Conceptual Graph (CG) theory, proposed by Sowa [Sowa, 1992] as a universal knowledge representation, is simple and powerful enough to satisfy our tutorial purpose. Moreover, in the past ten years, many works were done by several research groups to see how the theory might be used in several applications and serve as a basis to reformulate the other paradigms. Thus, several works were done in reasoning with CG in a Prolog-like style, in developing expert systems shells, in building natural language processing systems, or dealing with information systems, using CG (for more details see the seven proceedings on CG, the international conference on CG (Quebec, August 1993) and a collection of papers in [Nagle et al. 1992]). Most of these works are based however on what we can call an algebraic level (or conceptual algebra) of the theory : a structure called CG (that allows to represent several kinds of conceptual structures) and a set of basic operations on CG.

This paper presents a learning environment for this "Conceptual Algebra". Using visual presentations and Hypermedia techniques, a tutor drives the student to understand the meaning of these operations without cumbersome details on a particular implementation of the structure (CG in particular) and the operations. This understanding is useful to distinguish how these operations are combined and when they have to be applied. Also, the tutor can be used by a teacher to express basic concepts of some specific domains (natural langage processing, reasoning, ...) by providing specific examples and integrating the tutor with other tools (such as an inference engine or a semantic interpreter). The next section describes the main operations and components of the tutor and provides several examples that show how it works. We give also some details about the implementation of the Conceptual Algebra, in Prolog .

2. Learning a conceptual algebra

The tutor is a learning environment built for two courses : one on types hierarchy and the second on Conceptual Graphs principles and operations. For each course there is a teaching session (in a "guided-tour" form) and an exploration session (coaching phase) where students are coached while they deal with some examples and exercises. The first section will present briefly a first course and the second section the second course.

2.1 Learning Types Hierarchy

The tutor displays an example of a types hierarchy and introduces (using sound and marquer to show items in the example) some basic notions such as concept *types*, that constitute the nodes of a hierarchy, concept *instances*, *IS-A* link that corresponds to a link between two concepts, and *Instance-of* link that connects an instance to its type. To obtain more information, the student can choose an item (by clicking on it) and activate a window (similar to balloons approach) that contains a textual description. For example, in Figure 1, the student clicks on the IS-A relation, between PLANT and TREE, activates hence an explanation window. The student can also choose some items in the description itself. For example, he can click on the word "transitive" to obtain more information (in fact, all text sequences are composed like an hypertext).

Figure 1 shows the learning environment in which the student can create and modify (add or delete some links and/or nodes) his proper hierarchy by activating the options offered to him : options TYPE, Instance, IS-A and Instance-of, allow respectively a creation and/or a selection of a type, an instance, an Is-a link and an Instance-of link.

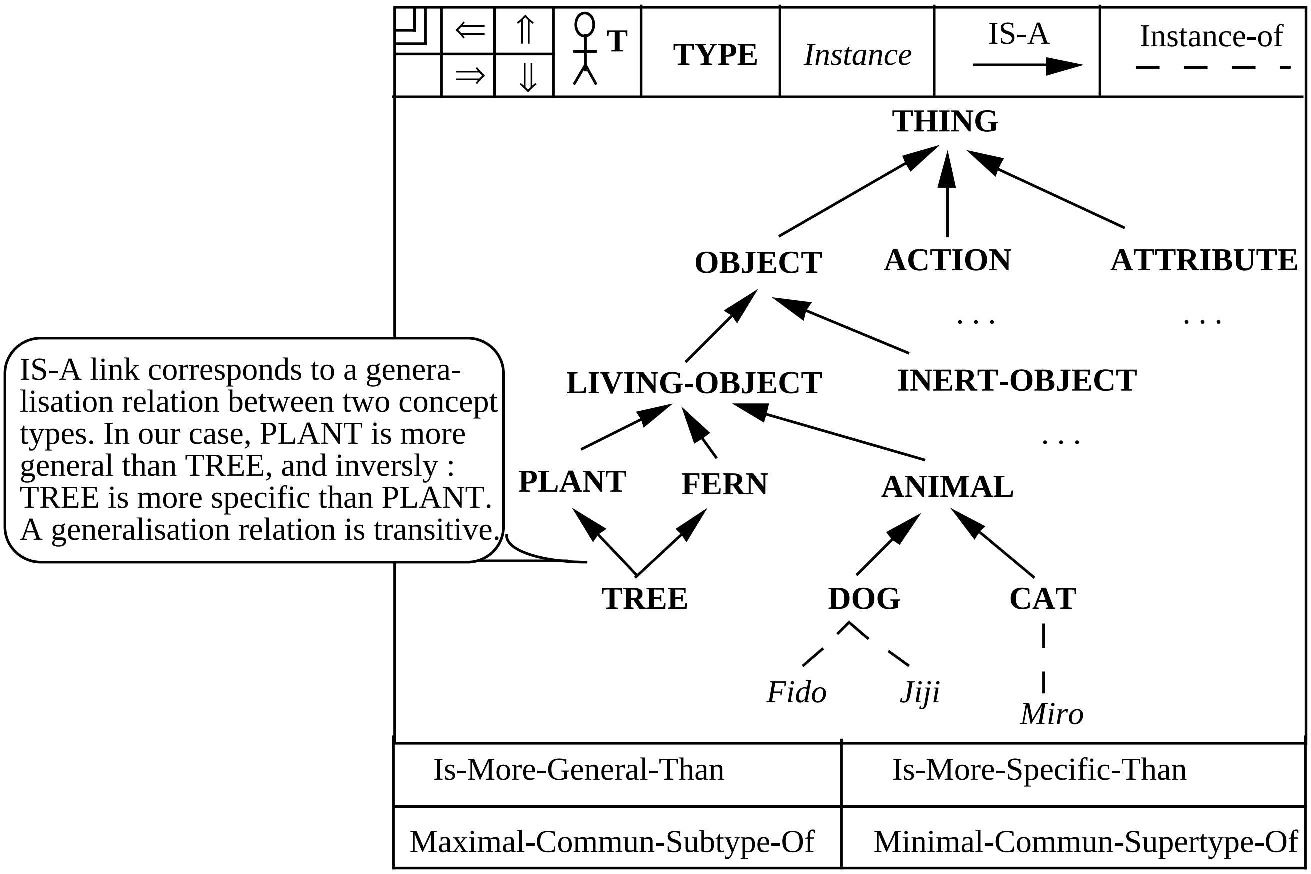

Figure 1 : Environment for the creation and manipulation of types hierarchy

A verification is done by the tutor since there are some restrictions on the relations : IS-A link connects two types and can't be used to link an instance to its type. In the same way, Instance-of link connects an instance to its type and can't be used to link two types.

Operations on type hierarchies can be activated through a pop-up menu located at the bottom of the window :

- Type1 **Is-More-General-Than** Type2 if Type1 is an antecedent of Type2 (Type1 is a supertype of Type2), and reciprocally :
- Type2 **Is-More-Specific-Than** Type1 if Type2 is a descendant of Type1 (Type2 is a subtype of Type1).
- **Maximal-Common-Subtype-Of** Type1 and Type2 is Type3. Type3 is the most specific type common to Type1 and Type2.

- **Minimal-Common-supertype-Of** Type1 and Type2 is Type3. Type3 is the least general type common to Type1 and Type2.

These operations are visually illustrated by showing the propagation of the search through the hierarchy and the exploration of various paths. Also, different use of an operation are demonstrated. For example, if the student specifies the two arguments of the Is-More-Specific-Than operation, a tutor will do just a test, if only the first argument is given then the tutor will show successively all the antecedents of the first argument, in the diagram.

In an exploration session, the student can activate an operation by clicking on it, on the arguments and again on the operation to terminate the activation. For example, the student can click on Is-More-Specific-Than operation, then click on one type and click again on the operation. If the student requires of a solution, he can click on the icon that represents the tutor (figure 1, option at the left of TYPE), otherwise he gives his/her solution (by direct manipulation) and the tutor will verify and correct it if necessary. Exercices are also proposed by the tutor. For example, the tutor can ask the student to give a Maximal-Commun-Subtype-Of of TYPE2 and TYPE6 (Figure 2); if the student gives TYPE7 as an answer, than the tutor will correct him and will show the two paths that lead to TYPE8, which is the correct response.

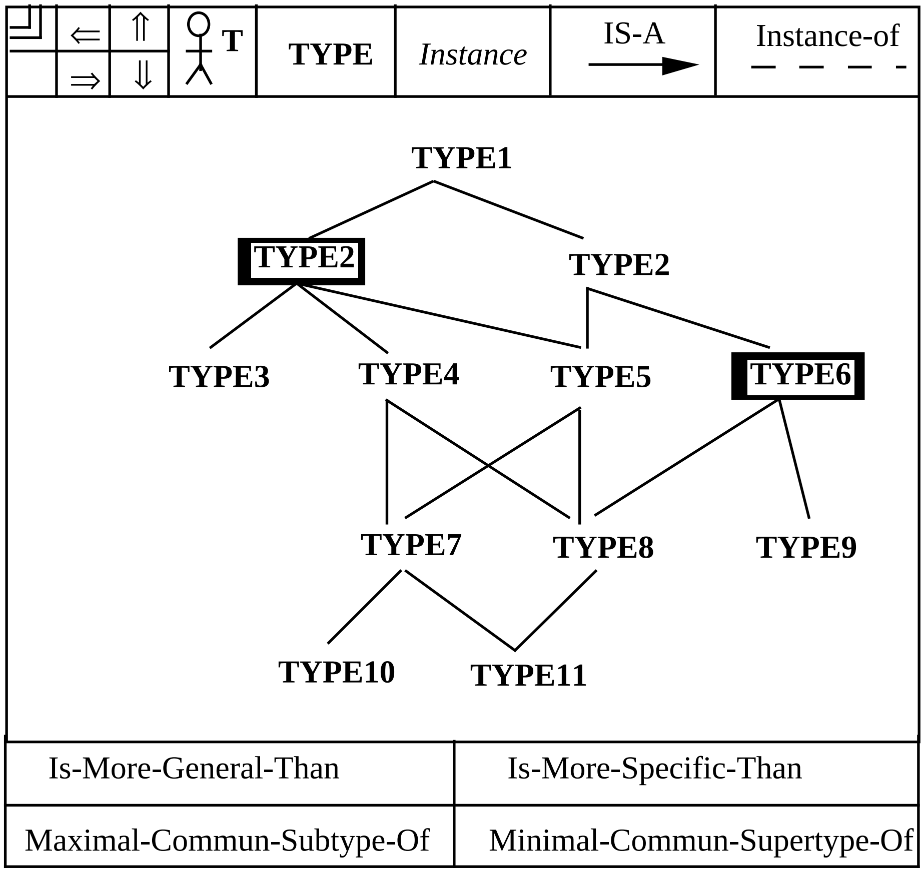

Figure 2 : Operations on Type Hierarchy

2.2 Learning How to Manipulate CG: teaching session

Using the same philosophy, the tutor introduces the concept of Conceptual Graph by showing and commenting an example. A CG is defined as a connected graph with nodes representing concepts and links conceptual relations ; a concept is composed of a type and a referent (optional) that can be a variable or an instance (Figure 3).

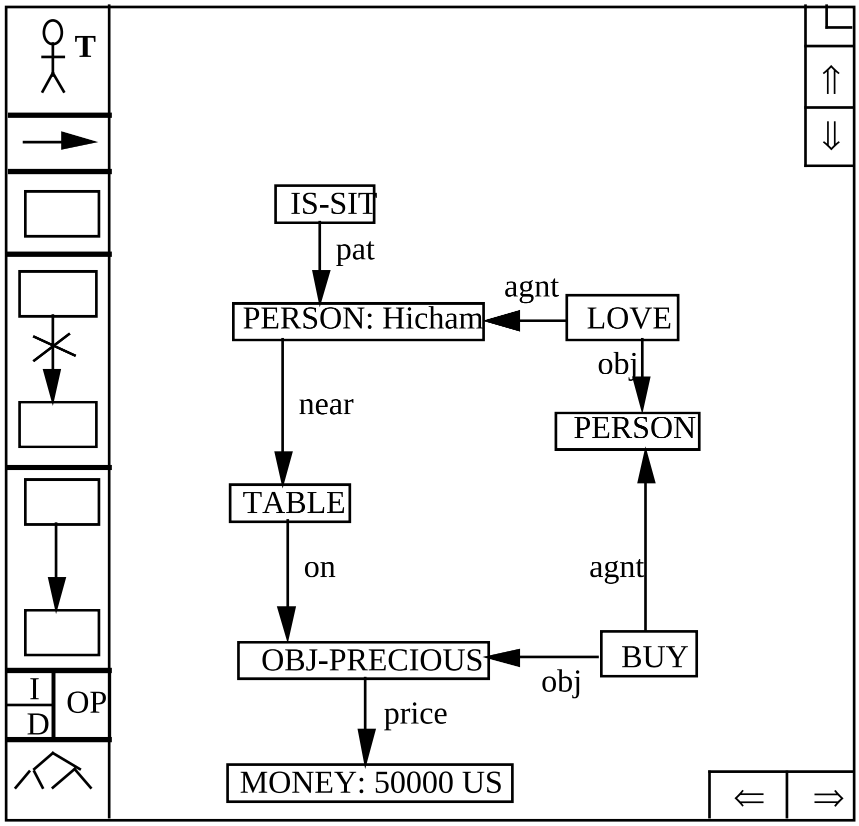

Figure 3 : Environment to manipulate CG

Also, the tutor illustrates with some examples different kinds of restrictions :
- a concept is unique in a CG ; we can't have two concepts in a CG refering to the same instance.
- relations are functional : a concept can't have identical input relations and can't have identical output relations.

First, the tutor gives an oral presentation and a textual description of these basic concepts but the student can ask for more information if needed. For instance, if he does not understand what the tutor means by "input relations" to a concept, he can ask the tutor (by simply clicking on the word) which will display a diagram showing an example of input relations for a concept (Figure 4).

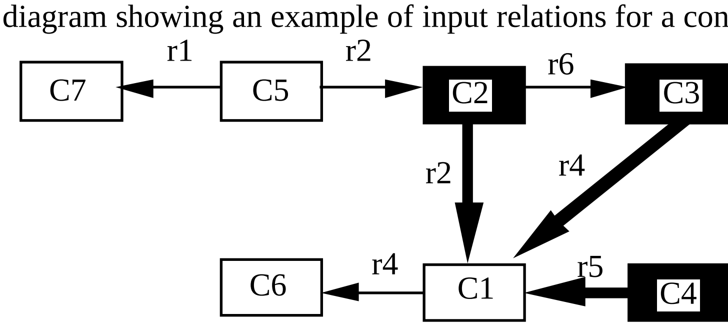

Figure 4 : More information on "input relations"

After an introduction to the basic concepts of CG, the tutor describes the **role** of each option of the environment provided in the menu (left side of window in Figure 3). These options enable the student to create his/her CG and/or modify an existing one. We describe briefly these options :

- *The relation option* (second icon indicated in the upper left side) gives the student the possibility to specify or change a relation.
- The *rectangle option* (which represents a concept) enables the student to edit a concept : changing its type and/or its referent, or to have a definition of the concept type (if the concept type is defined) and/or a description of the instance (yet described by a CG).
- The fourth and fifth options are provided respectively for *eliminating and adding* a relation between two concepts.
- The sixth option allows to *activate an operation* on CG associated with a specific mode: in I mode (for Incremental), the tutor will show each step in the execution of the operation, if the student chooses the D (for Direct) mode, the tutor will show the result only. When a student chooses the fifth option, the tutor displays a hierarchy of operations and the student must choose among them (Figure 5 shows an example with a selection of a match operation).

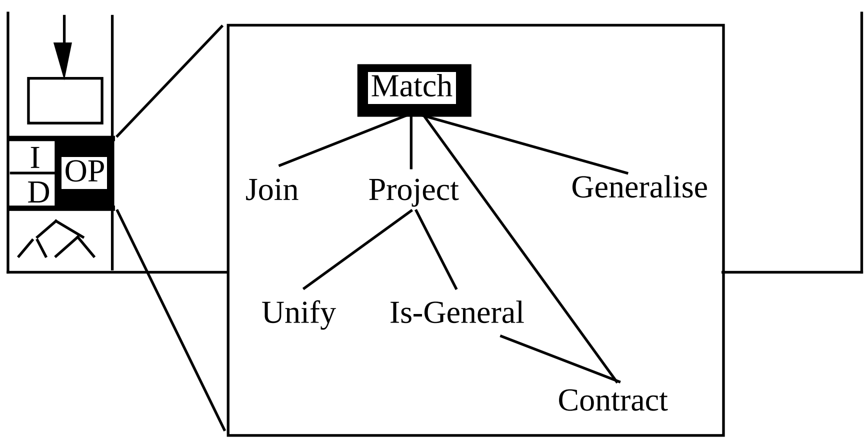

Figure 5 : Part of the menu with the operation hierarchy

- The last option allows to visualize types hierarchy and execution of operations (more detail will be given for this option).

Once the student has edited and created a CG, the tutor will verify that his CG respects the restrictions described above (each concept must be unique in the CG, relations must be functional, ...).

We first give a detailed description of the matching algorithm then we show how the tutor visualizes such a description to the student (but the tutor can give a textual description on request as well). In doing so, we take into account the difference between textual/visual description in one side and a descriptive/procedural description in the other side and their impact on tutoring. We believe that it is more appropriate to expand on a visual-procedural description, showing how a procedure works on some examples in a *visually animated* way.

Textual description of matching operation

Entry points : matching two CG is a generic operation that identifies the structural recovery between two CG and finds the concepts to match. Propagation of the match between two CG, g1 and g2, begins with two concepts c1 and c2 (of g1 and g2 respectively) called "entry points" and can be determined according to four ways :

- provided directly to the operation (by the student or a process that calls the operation),
- determined by the presence of the same referent in g1 and g2 : if g1 has a concept [TYP1: ref] and g2 has a concept [TYP2: ref], with *ref* corresponding to a particular individual, then entry points for g1 and g2 are [TYP1: ref] and [TYP2: ref].
- determined by co-referent links, when g1 and g2 are compound CG. We skip this possibility here but Annex 1 provides an example and [Kabbaj, 1987, 1992a] offers a detailed discussion.
- determined by the structure of g1 and g2 : we have to find in g1 a branch [Cs1] --(rel)--> [Cc1] and in g2 a branch with the same relation rel : [Cs2] --(rel)--> [Cc2], so that Cs1 can match with Cs2, Cc1 with Cc2 and the others outcome branches of Cs1 can match with outcome branches of Cs2.

Definition : matching two CG g1 and g2 consists successively in identifying their entry points c1 and c2, matching c1 and c2, and doing a local matching around c1 and c2 : matching outcome branches of c1 with those of c2 and income branches of c1 with those of c2, i.e., for each outcome branch [c1] --(r)--> [cc1], if c2 has an outcome branch with the same relation : [c2] --(r)--> [cc2], then we have to match cc1 with cc2 and perform a local match around them. The same principle is applied to income branches of c1 and c2.

Hence, a match between two CG begins with the two entry points and propagates by a local match which generates other match and so on until a "closeness" of the operation, i.e., when no new match appears.

Matching two concepts consists in matching their types, referents and co-referent links ; if referents are themselves CG, then we have a recursive call to matching operation.

Explanation by a tutor

To teach the student this operation, the tutor begins by a presentation of two CG, g1 and g2 (Figure 6), then shows gradually how the match is performed by highlighting the corresponding concepts. The next figures illustrate the process.

In figure 6, the tutor proposes the entry points [MAN] for g1, and [PERSON : Hicham] for g2, and shows the creation of a resulting concept (i.e., [MAN/PERSON:Hicham]) of the newly CG gMatch.

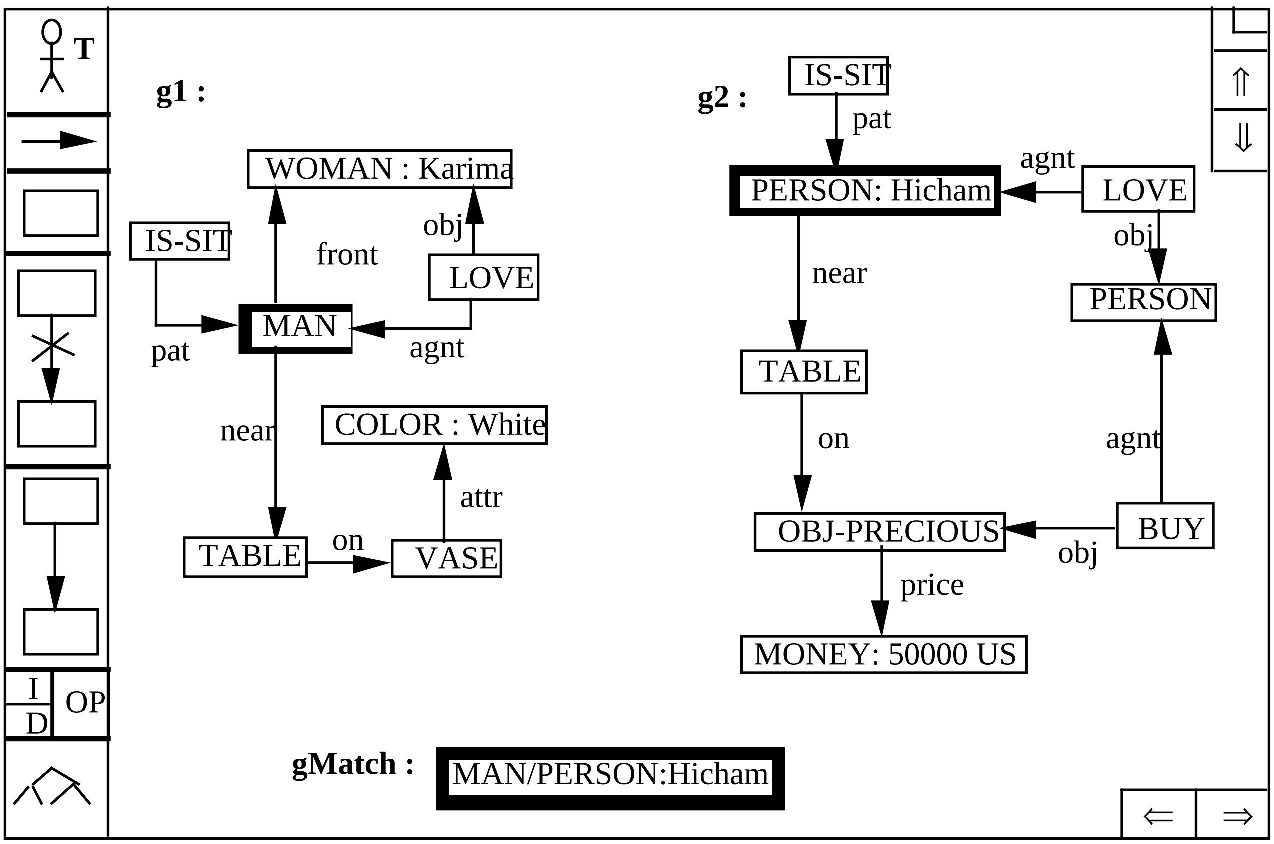

Figure 6 : Matching two CG, first step

The tutor will then construct the resulting CG gMatch by adding gradually common input and output branches to the resulted concept [MAN/PERSON: Hicham]. Figure 7 shows the propagation of the match around [MAN] and [PERSON:Hicham] through the input realtion "agnt", this results in the matching of [LOVE] and [LOVE] and the addition of a branch [MAN/PERSON:Hicham] <--(agnt)-- [LOVE/LOVE] to gMatch.

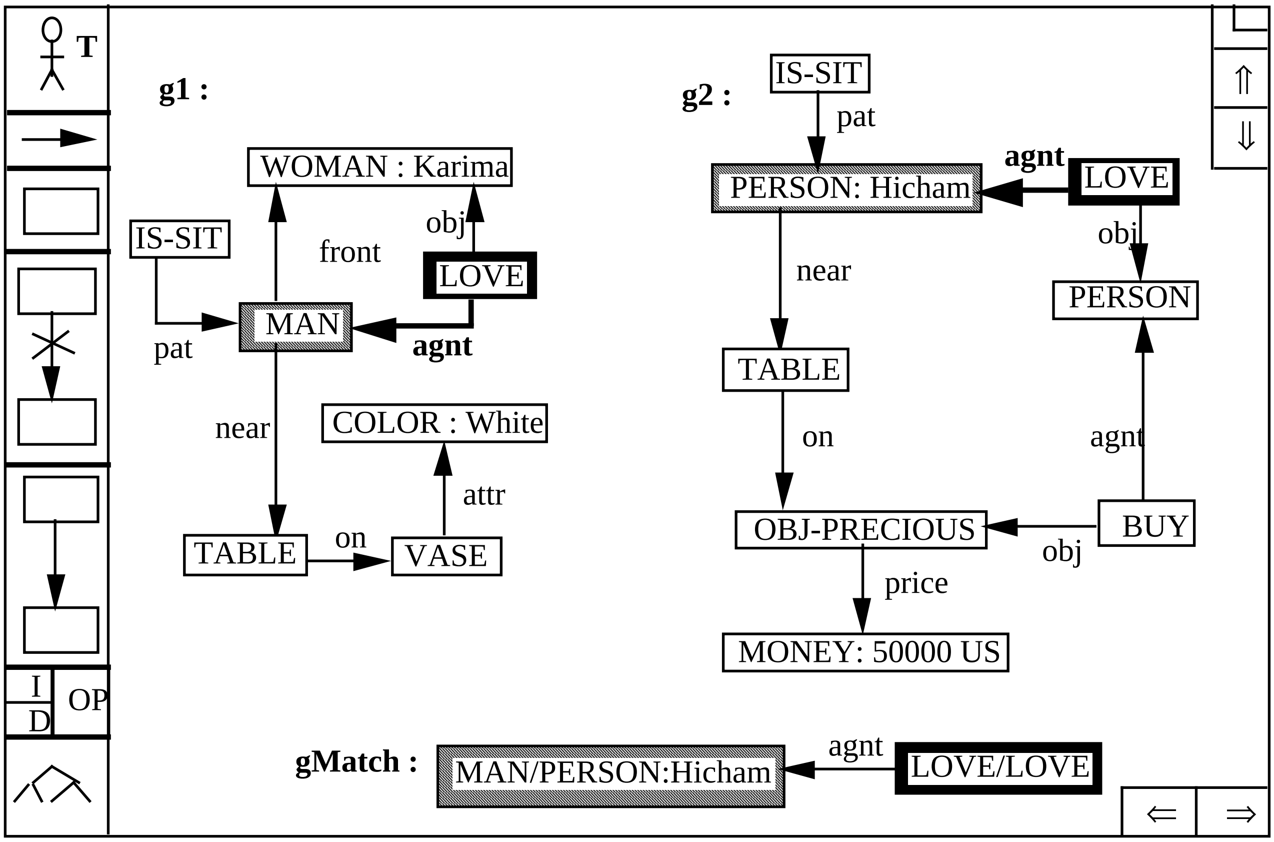

Figure 7 : propagating the match through the input relation "agnt"

Next, the tutor considers the input relation "pat" (figure 8). This results in the matching of [IS-SIT] and [IS-SIT] and the addition of the branch [IS-SIT/IS-SIT] --(pat)--> [MAN/PERSON:Hicham] to gMatch.

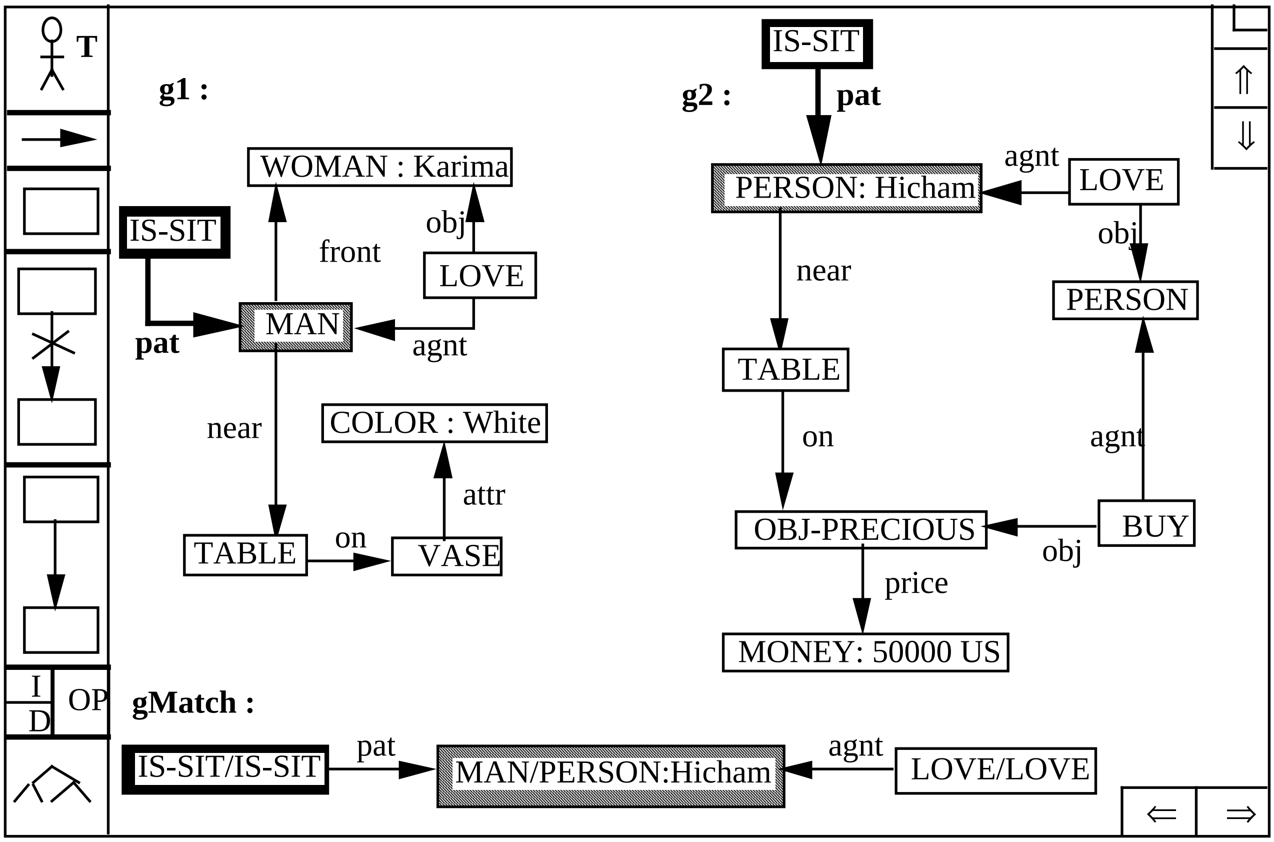

Figure 8 : Propagating the match through the input relation "pat"

The same process is done for the output relation "near". The result is shown in figure 9.

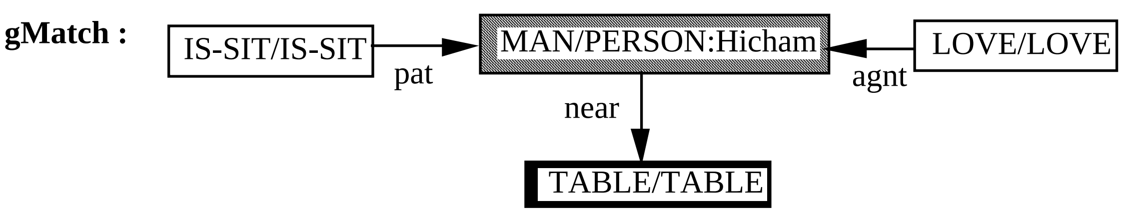

Figure 9 : tResult of the propagation through the output relation "near"

After the local matching around [MAN] and [PERSON:Hicham], the process continues with [TABLE] and [TABLE] as a new candidate to a local matching (figure 10).

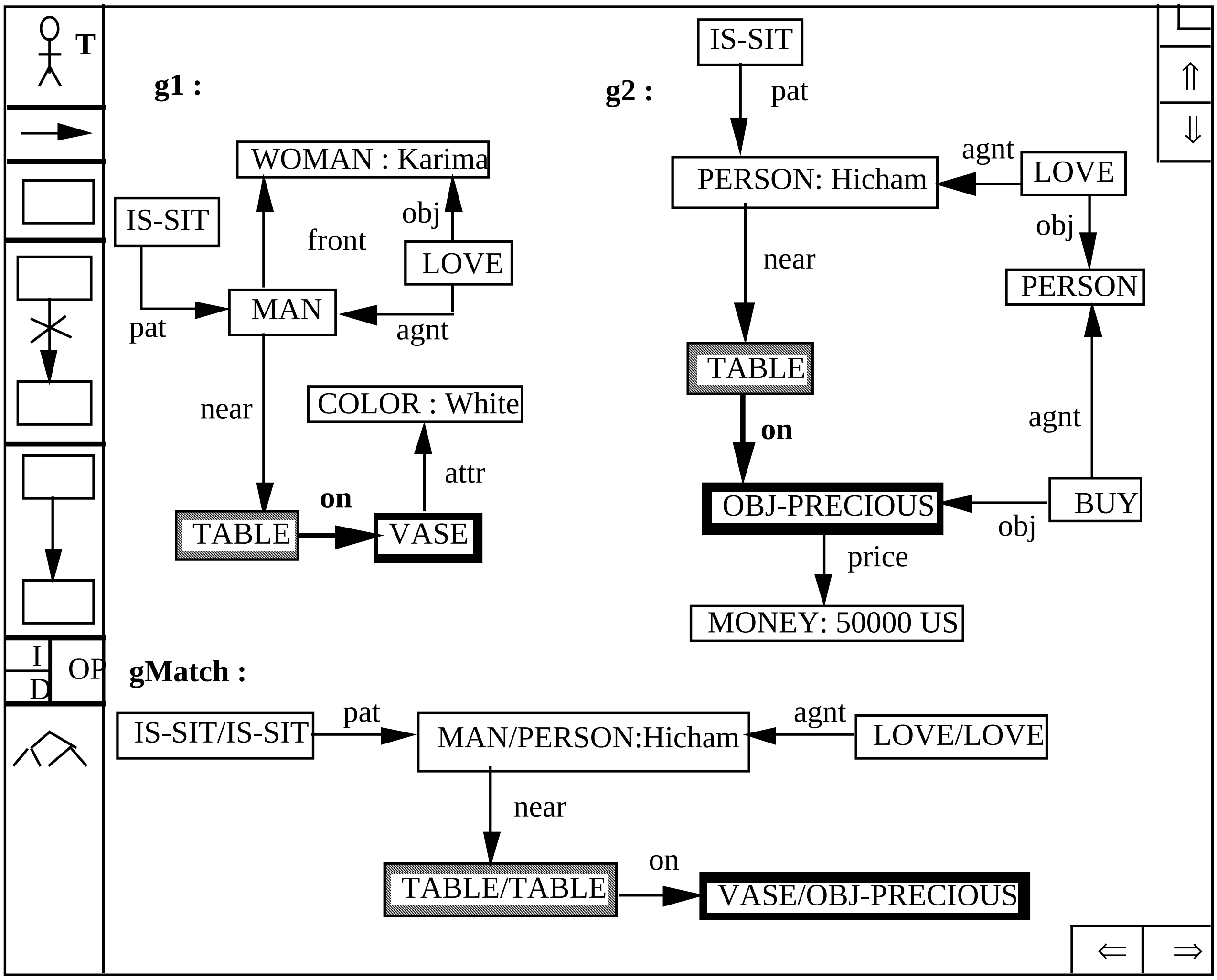

Figure 10 : Local matching around [TABLE] and [TABLE]

The tutor finds that he can't propagate a local matching further through [TABLE/TABLE] since [TABLE] and [TABLE] have no more branches in common. The tutor then attempts a local matching around [VASE/OBJ-PRECIOUS], but since [VASE] and [OBJ-PRECIOUS] have no corresponding branches in common, nothing is added to gMatch. This is also the case for [IS-SIT/IS-SIT]. A local matching around LOVE/LOVE is then attempted which results in the addition of the branch [LOVE/LOVE] --(obj)--> [WOMAN:Karima/PERSON] to gMatch. Now, the next step is to make a local matching around [WOMAN:Karima] and [PERSON] (figure 11).

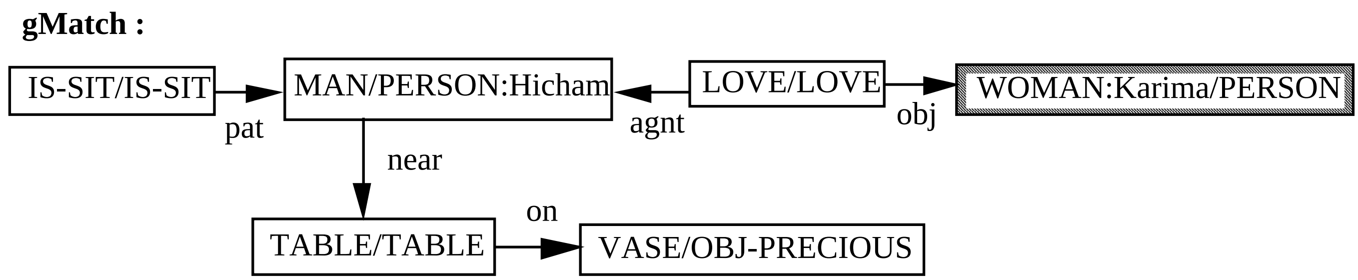

Figure 11 : Making a local match around [WOMAN:Karima/PERSON]

Discovering that nothing is common to [WOMAN:Karima] and [PERSON], the tutor terminates the local matching around these concepts; the box representing [WOMAN:Karima/PERSON] is now without grey background, and the whole matching is terminated.

Note : matching two CG leads to match two concepts and ultimately to match two concept types. The hierarchy option (bottom left in figure 3) provides the student with a possibility to activate the visualization of the search through the hierarchy when a match concerns two concept types, or desactivate the visualization and concentrate only on the matching at the CG level.

As illustrated by figures 6 to 11, a detailed algorithm of each operation is "visible" to the student and its execution is then visually animated. Such an approach enhances the comprehension process and in the same time liberates the student from the detail of a particular implementation and even from a textual description of an algorithm (unless a student asks for such a description). The examples provided by the tutor can be modified by the user : changing some relations, some concepts or a part of the examples. The student can also add or delete some parts of the examples. After that, he/she can ask the tutor to activate an operation and observe the result.

Coaching

In the coaching phase, the tutor can give an exercise to the student, providing some hints on request. For instance, a tutor can show two CG and ask the student to give the result of the match. The student decides when to activate the tutor (by clicking on the corresponding icon, in the upper left corner) to obtain a feedback about the validity of his solution or some help. Help provided by the tutor can be progressively general (a textual description of how to match two concepts or two ouput branches for example), or particular (giving a matching of two current concepts or a match of two current output branches for example).

The same methodology is used to teach the other operations on the CG (Annex1 provides a short definition and examples of these operations).

3. Implementation

In the last ten years, editors of several scientific journals encouraged researchers to give a detailed description of their algorithms and, if possible, examples of code in a high level language such as Lisp and Prolog. We present in this section some details about an implementation of CG and matching operation. See [Kabbaj 1993] for a discussion on the implementation of the tutor.We have chosen Prolog (C-Prolog for instance) as the implementation language.

3.1. Implementation of CG

A CG is described as a list of elementary CG (ECG) each one being defined as a couple of a concept and a list of output branches. An output branch itself is a couple of a relation and a concept. A concept can be an identifier or a couple of type and referent. As an example, consider the following CG and its description according to the above conventions.

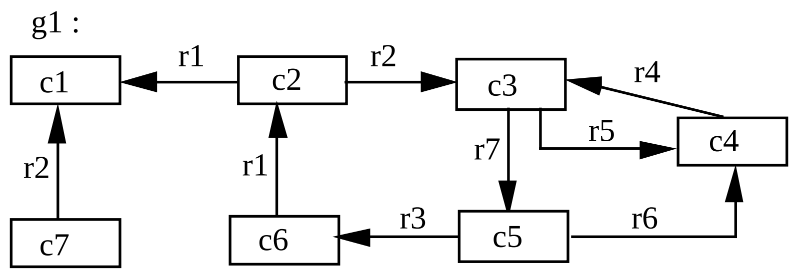

Figure 12 : an example of a CG

[(c2, [(r1, c1), (r2, c3)]),   (c7, [(r2, c1)]),   (c6, [(r1, c2)]),
 (c3, [(r5, c4), (r7, c5)]), (c5, [(r3, c6), (r6, c4)]), (c4, [(r4, c3)])]

To store a CG in a factual database, we can assert it as a fact :   cg(identCG, <descriptionCG>).
| ?- assert(gc(g1, [(c2, [(r1, c1), (r2, c3)]),   (c7, [(r2, c1)]),   (c6, [(r1, c2)]),
(c3, [(r5, c4), (r7, c5)]), (c5, [(r3, c6), (r6, c4)]), (c4, [(r4, c3)])])).

3.2 Implementation of matching

As shown in section 2, the matching operation is generic; it does not resolve a match of concepts (we specify only that [WOMAN:Karima] must match with [PERSON] for example, see figure 6-11). But we can specialize it by providing a particular matching operation for concepts. Thus, the operation will have two CG and a matching concept operation as input , and a CG as output. If a matching concept operation is not defined at the call of the operation, match will operate at the generic level (as shown in section 2) providing a structural recovery of the two input CG and a specification of the concepts to match.

```
/*****  Basic goals that defines the matching operation *********/

matching(MatchConc, GC1, GC2, Gres) :-
   entryPoint(MatchConc, GC1, GC2, G1, G2),  match(MatchConc, G1, G2, Gres), ! .
matching(_, _, _, _) :- eraseTempMemory, ! , fail .

entryPoint(MatchConc, (G1, Conc1), (G2, Conc2), G1, G2) :-  ! , matchConc(MatchConc, Conc1, Conc2, _) .
entryPoint(MatchConc, G1, G2, G1, G2) :-
    enumereConcOfCG(G1, Lconc1),  member(Conc1, Lconc1),   decomposeConc(Conc1, Type1, Ref1),
    enumereConcOfCG(G2, Lconc2),  member(Conc2, Lconc2),   decomposeConc(Conc2, Type2, Ref1),
    ! ,  matchConc(MatchConc, Conc1, Conc2, _).
entryPoint(MatchConc, G1, G2, G1, G2) :-
   branch((ConcSource1, Rel, ConcCible1), G1) ,  branch((ConcSource2, Rel, ConcCible2), G2) ,
   eraseTempMemory,
   matchBranch(MatchConc, (ConcSource1, ConcSource2), (ConcCible1, ConcCible2)) ,
   matchOutBranch(MatchConc, G1, ConcSource1, G2, ConcSource2).

matchBranch(MatchConc, (ConcS1, ConcS2), (ConcC1, ConcC2)) :-
   matchConc(MatchConc, ConcS1, ConcS2, _),   matchConc(MatchConc, ConcC1, ConcC2, _), ! .
matchBranch(_, _, _) :- eraseTempMemory, ! , fail .

match(MatchConc, G1, G2, Gres) :-
   replace(concMatch(Conc1,Conc2,Conc3,0),  concMatch(Conc1,Conc2,Conc3,1)),
   listOutBranch(G1, Conc1, LOutBranch1),    listOutBranch(G2, Conc2, LOutBranch2),   ! ,
   if((LOutBranch1 = [] ; LOutBranch2 = []),  LOutBranchRes = [] ,
          matchOutBranch(MatchConc, LOutBranch1, LOutBranch2, LOutBranchRes)),
   matchInBranch(MatchConc, G1, Conc1, G2, Conc2),
   match(MatchConc, G1, G2, RestGres),
   if(LOutBranchRes = [], Gres = RestGres, Gres = [(Conc3, LOutBranchRes)|RestGres]).
match(_, _, _, []).

listOutBranch(G, Conc, LOutBranch) :- member((Conc, LOutBranch), G).
listOutBranch(G, Conc, []).

matchOutBranch(MatchConc, [(Rel, Conc1)|Rlist1], LOutBranch2, [(Rel, Conc3)|Rlist3])  :-
   member((Rel, Conc2), LOutBranch2), !,
   matchConc(MatchConc, Conc1, Conc2, Conc3),
   matchOutBranch(MatchConc, Rlist1, LOutBranch2, Rlist3), ! .
matchOutBranch(MatchConc, [_|Rlist1], LOutBranch2, Rlist3) :-
   matchOutBranch(MatchConc, Rlist1, LOutBranch2, Rlist3), ! .
matchOutBranch(_, [], _, []).

matchInBranch(MatchConc, G1, Conc1, G2, Conc2) :-
   branch((ConcSource1, Rel, Conc1), G1),
   no(concMatch(ConcSource1,_, _, _)),
   branchD((ConcSource2, Rel, Conc2), G2),
   if(matchConc(MatchConc, ConcSource1, ConcSource2, _), fail, true), ! , fail.
matchInBranch(_, _, _, _, _).
```

```
branchD(Branche, G) :-   branchD(Branche, G), ! .

matchConc(_, Conc1, Conc2, Conc3) :- clause(concMatch(Conc1, Conc2, Conc3, _), _), ! .
matchConc(MatchConc, Conc1, Conc2, Conc3) :-
  nonvar(MatchConc),
  decomposeConc(Conc1, Typ1, Ref1),
  decomposeConc(Conc2, Typ2, Ref2),  ! ,
  But =.. [MatchConc, Typ1, Typ2, Typ3],  But,
  matchRef(MatchConc, Ref1, Ref2, Ref3),
  conforme(MatchConc, Ref3, Typ3),
  composeConc(Conc3, Typ3, Ref3),
  asserta(concMatch(Conc1, Conc2, Conc3, 0)), ! .
matchConc(MatchConc, Conc1, Conc2, Conc3) :-
  var(MatchConc),
  asserta(concMatch(Conc1, Conc2, match(Conc1, Conc2), 0)), ! .
```

The following give an idea of the implementation of the others operations.

/*** join operation is a match with intMax (for Maximal-Commun-Subtype-Of), resulting in a matching operation between concepts and combining specific information of each CG, see Annex 1 for more detail).   ****/

```
join(GC1, GC2, Gres) :- matchWithTreatment(intMax, constructGCjoin, GC1, GC2, Gres).
```

/**** generalize is a match with unionMin (for Minimal-Commun-supertype-Of) as a match operation between concepts *******/

```
generalize(GC1, GC2, Gres) :-  matching(unionMin, GC1, GC2, Gres).

unify(GC1, GC2) :- unifyWithGCres(GC1, GC2, _).

unifyWithGCres(GC1, GC2, Gres) :-  project(intMax, GC1, GC2, Gres).
```

/**** IsGeneral is a projection with moreGeneralThan (for Is-More-General-Than) as a match operation between concepts ********/

```
IsGeneral(GC1, GC2) :-  project(moreGeneralThan, GC1, GC2, _).
```

/****** project is also generic, it is a matching with a restriction : a first CG must be completely recovered by the second CG ; the first CG must be included (structuraly) in the second ****/

```
project(MatchConc, GC1, GC2, Gres) :-
  matchWithTreatment(MatchConc, testInclusion, GC1, GC2, Gres).


matchWithTreatment(MatchConc, SupplTreatment, GC1, GC2, Gres) :-
  matching(MatchConc, GC1, GC2, GresInt),
  if(GC1 = (Gx, Conc1), G1 = Gx, G1 = GC1),
  if(GC2 = (Gy, Conc2), G2 = Gy, G2 = GC2),
```

if(SupplTreatment = testInclusion, (testInclusion(G1, G2), GresInt = Gres), constructGCjoin(G1, G2, Gres)),
eraseMemory, ! .

4. Conclusion

In this paper, we have presented the design of a tutor in Knowledge Representation and shown how the process is conducted alternatively using either a simple explanation or a coaching method. Both methods use a visual environment the student can manipulate, checking the consistency of his/her own construction with the explanations and hints of the tutor.

As a theoretical basis of a tutor, we have chosen the Conceptual Graph theory proposed by Sowa as a universal knowledge representation and also a practical basis ; we have used of a visual presentation with an Hypermedia support.

The realisation of the tutor is under way and we plan to do some experiments about the efficiency of such a tutor, i.e. does it increase and facilitate understanding of the conceptual algebra for different kinds of students and users ? Such an experimental study is similar to the one done by Anderson and ??? [92] on the advantages of data-flow approach over a textual one in teaching Lisp.

References

Kabbaj A.(1992), *Un système de représentation et manipulation des connaissances*, Rapport interne, DIRO, Université de Montréal.

Kabbaj A. (1993), Vers un tuteur pour la representation des connaissances, Rapport interne, DIRO, Université de Montréal.

Sowa J. F. (1992), Conceptual Graphs as a universal knowledge representation, in E. Y. Rodin (ed), *An International Journal computers & mathematics with applications : Special Issue on SEMANTIC NETWORKS IN ARTIFICIAL INTELLIGENCE*, v. 23:2-9.

Annex 1. Definition of other operations on CG

Join operation

Definition : Join a CG g1 with another CG g2 consists in matching them modulo the join of concepts and in adding to a resulted CG of all branches in g1 and g2 that aren't included in the

match. This operation returns a CG g3 that is more specific than g1 and g2. Join two concepts [TYP1: ref1] and [TYP2: ref2] returns a concept [TYP3: ref3] that is more specific than the two concepts, with TYP3 as a minimal common specialization of TYP1 and TYP2, and ref3 as a specialization of ref1 and ref2 [Kabbaj, 92a].
Figure 13 shows a join of two CG g1 and g2, the result is gJoin.

Generalization operation

Definition : Generalize a CG g1 with another CG g2 consists in matching them modulo the generalization of concepts. Branches that aren't included in the match are ignored. This operation returns thus a CG g3 that is more general than g1 and g2. The generalization of two concepts returns a concept that is more general than the two, and correspond in fact to the inverse of the Join on concepts.
Figure 13 shows a generalization of two CG g1 and g2, the result is gGeneral.

The following example (figures 14, 15, 16) shows a join of two compound CG g3 and g4 and a use of co-referent links to indicate entry points for imbedded CG. A co-reference link connects two concepts in two CG that refer to the same individual, it is represented by dashed lines.
Note : Meaning of some abbreviated relations are : pat for patient, attr for attribute, loc for location, cont for content, qty for quantity, dest for destination and hsbd for husband.

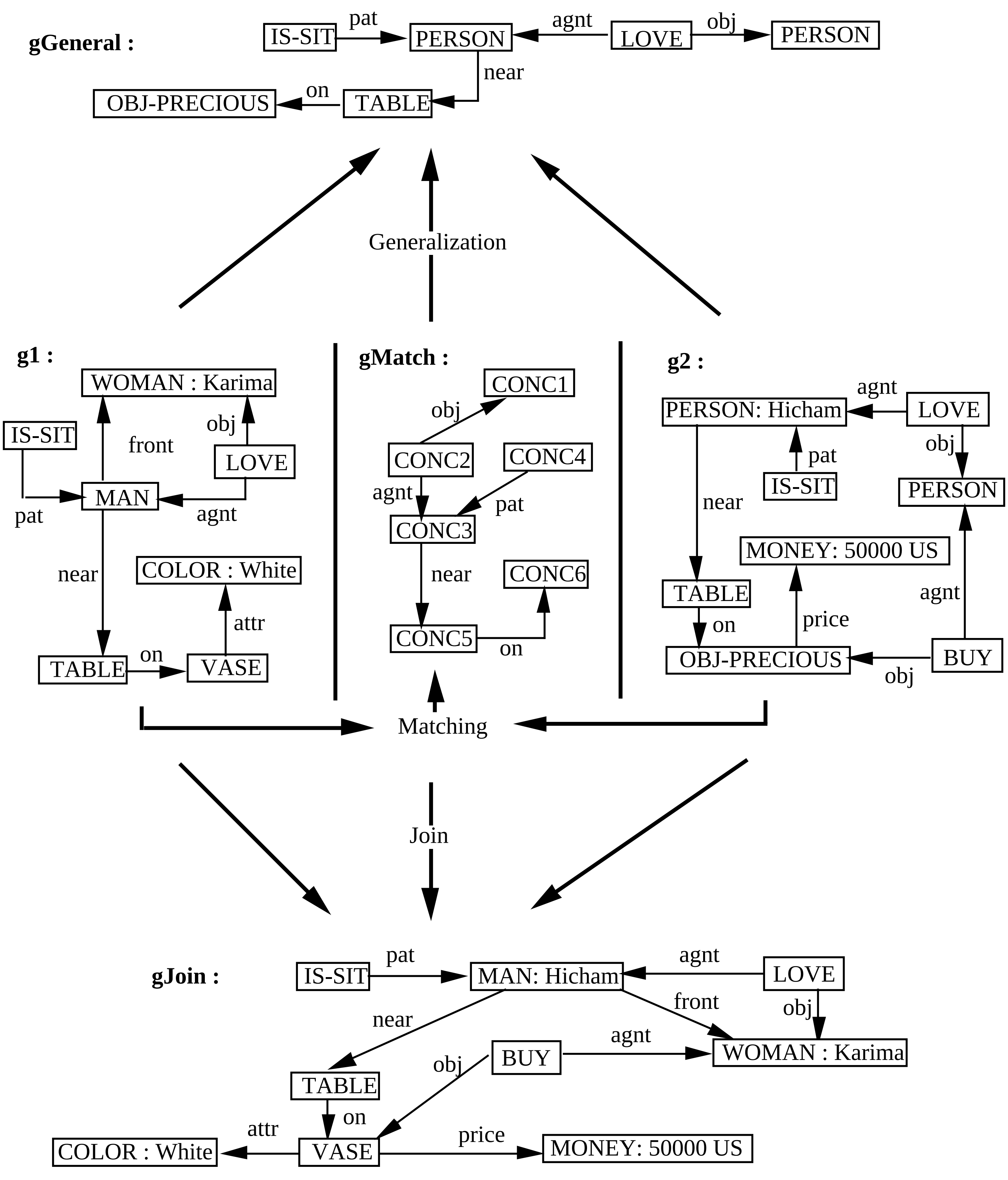

Figure 13 : Match, Join and Generalize operations

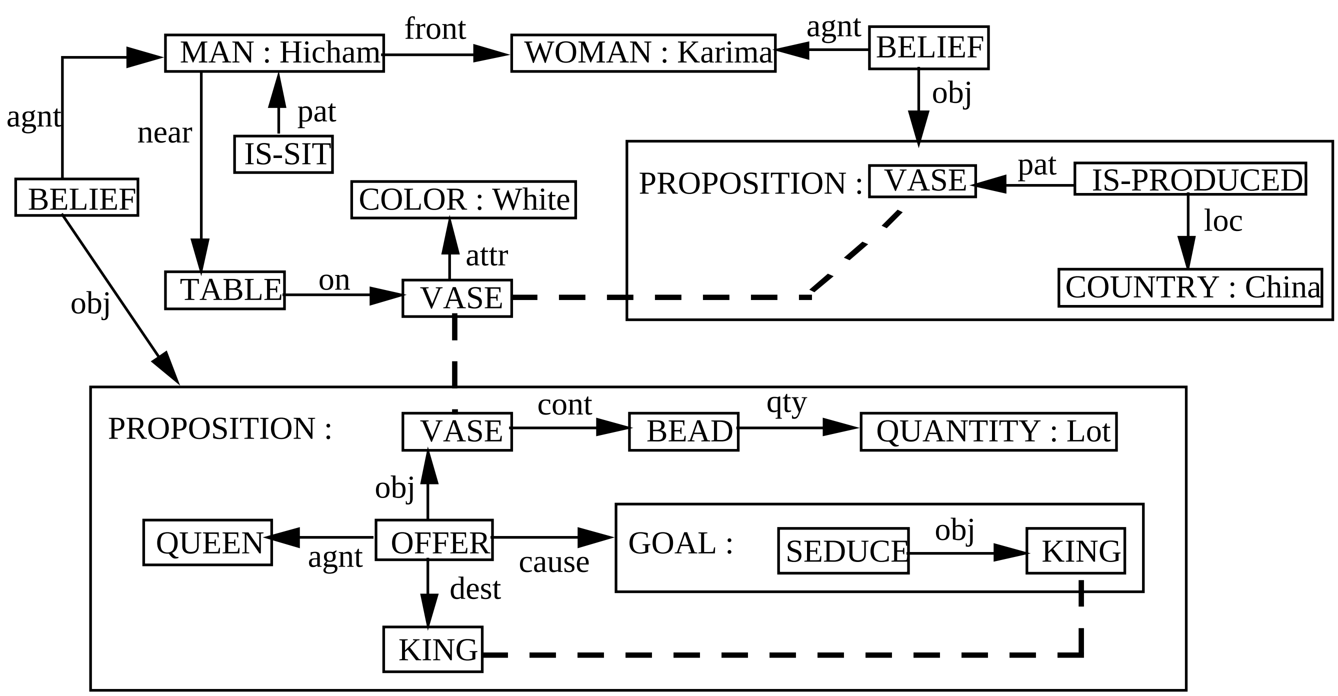

Figure 14 : compound CG g3

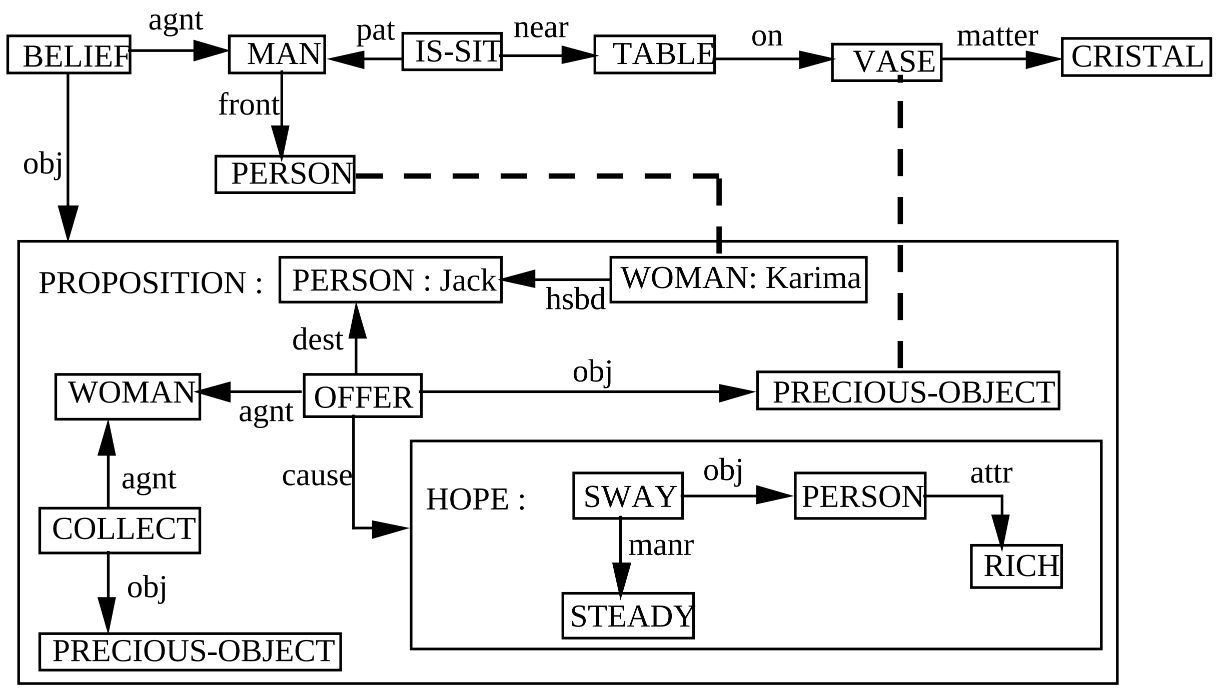

Figure 15 : Compound CG g4

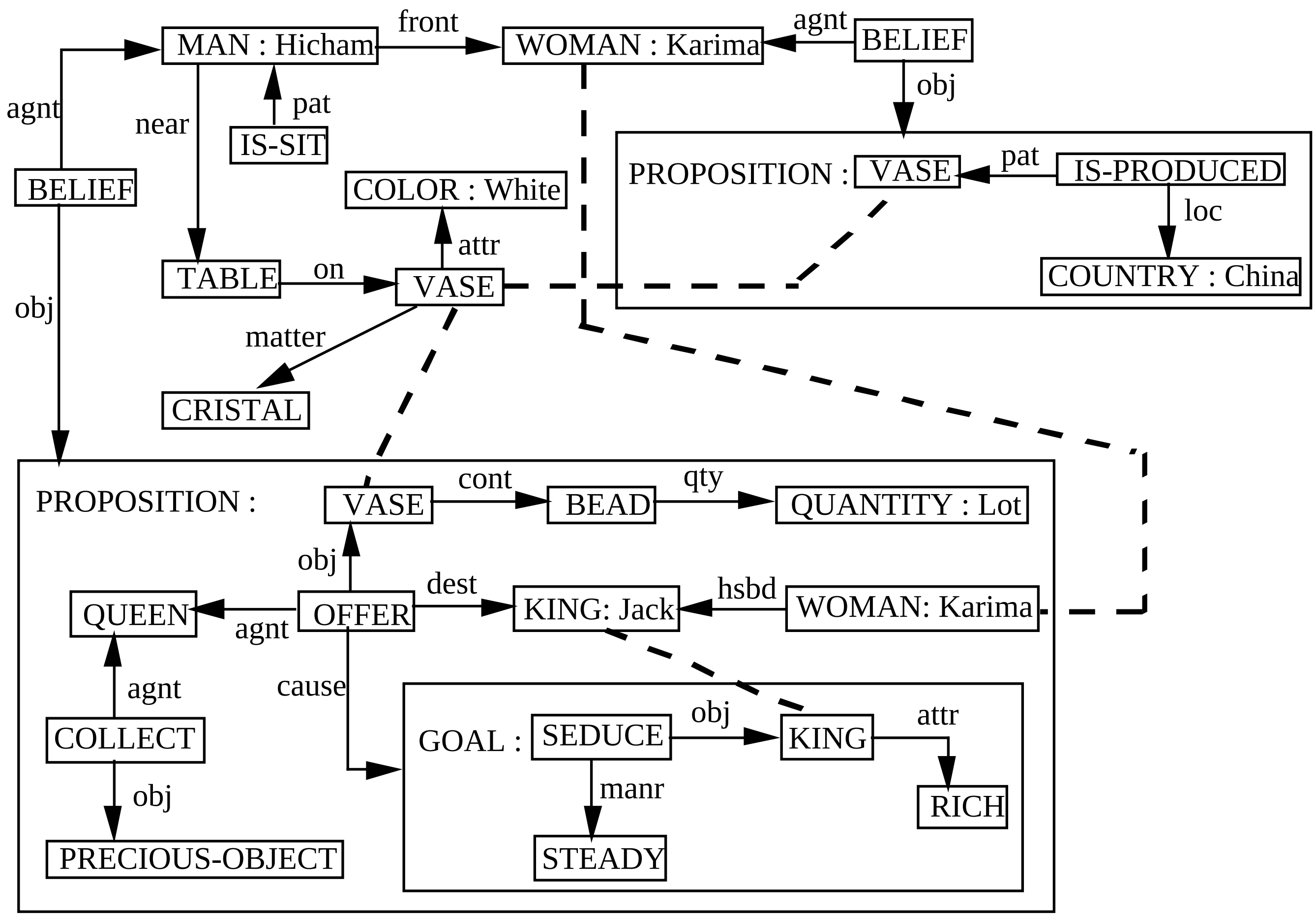

Figure 16 : Join result of the two compound CG g3 and g4

Projection operation

Definition : projecting a CG g1 on another g2 consists in matching them and verifying that g1 is structuraly equivalent to the result of the match.

The following example (figure 17) shows two CG, g1 and g2, with the result of their projection : gProj.

Unification operation

Definition : unify a CG g1 with another g2 consists in a projection of g1 on g2 modulo a join of their concepts.

Figure 17 shows also an unification of the two CG g1 and g2, the result is gUnif.

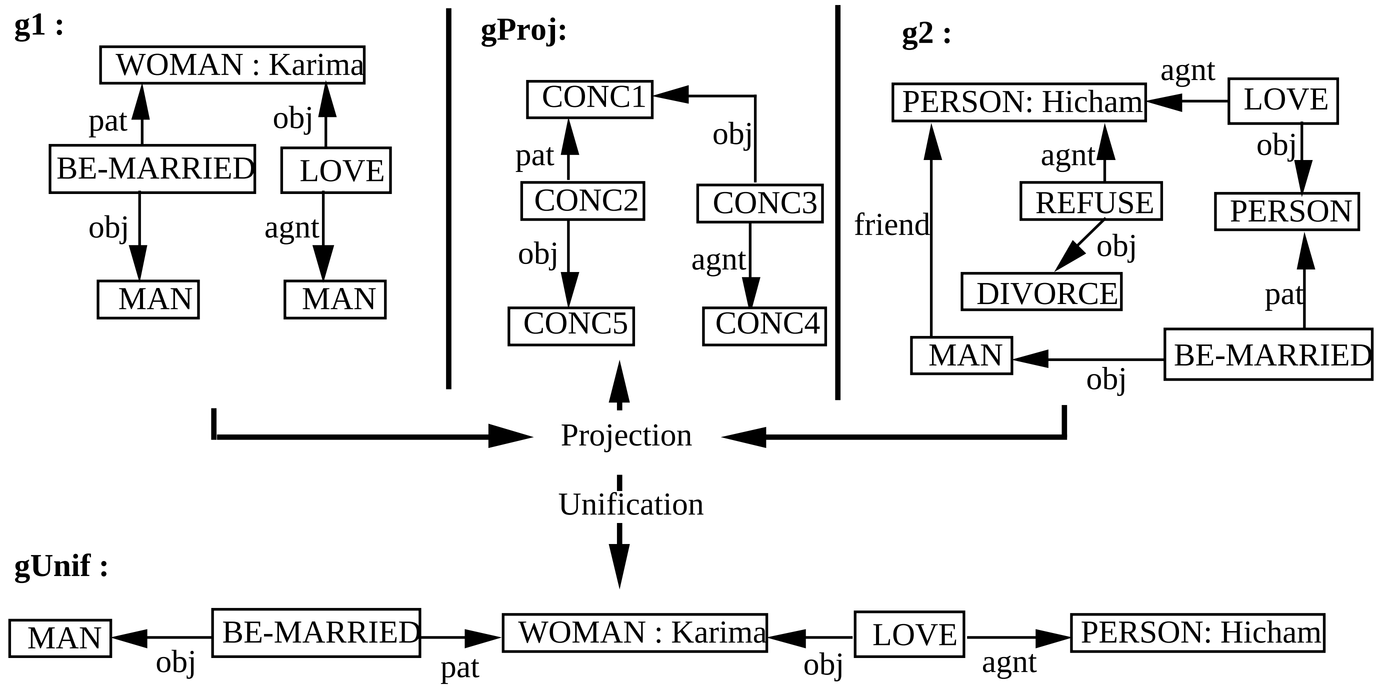

Figure 17 : Projection and unification of CG

Is-General operation

Definition : a CG g1 is more general than another g2 if we can project g1 on g2 modulo a IsGeneralThan operation between concepts of the two CG.

A concept [TYP1 : ref1] is more general than [TYP2 : ref2] if TYP1 is more general than TYP2 and ref1 more general than ref2. For instance, the concept [HUMAN : Ali] is not more general than [MAN] since referent of the first concept is more specific than referent of the second, and [MAN] is not more general than [HUMAN: Ali] since MAN type is more specific than HUMAN.

In the preceding example (figure 17), g1 is not more general than g2 because some concepts of g1 are more specific than their correspondent in g2.

Contraction operation

Definition : contract a CG g1 from a CG g2 consists in eliminating g1 from g2 preserving however a connection of g2. Eliminate g1 from g2 consists in eliminating from g2 all branches and particularly relations that exist in g1 and aren't "cut points" (i.e., if a relation is eliminated, the CG will be disconnected), and eliminates also concepts that are disconnected from the CG.

For a more detailed discussion of that operation and the others see [Kabbaj, 1992].