Interaction Models Group

TULE
Documentation

SEARCH FOR SHORTEST PATHS IN SEMANTIC INTERPRETATION

TULE Internal Report #5, November 2012

Author: Leonardo Lesmo

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF BOXES

LIST OF TABLES

1. INTRODUCTION

This report addresses part of the process of semantic interpretation of NL sentences. The entire process is sketched in Figure 1. It is assumed that the sentence has already been parsed, so the input is a parse tree (in TUT dependency format). The analysis proceeds through three main steps, as outlined in the figure, in order to get a final representation in First Order Logics.

As a first step, the semantic dictionary (lexical semantics) is accessed in order to annotate the parse tree with semantic information which, for content words, consists in references to an ontological KS. On the basis of this annotation, exploiting both the principle of compositionality and the extraction of base structures from the ontology, the interpreter builds an "ontological form" for the input sentence. Finally, this is translated in the final FOL formula.

The 'base structures' mentioned above are paths on the ontology. So, one of the basic operations is the one of searching for short paths that connect two ontology nodes. This, for instance, is needed to 'understand' (i.e. to find a correspondence with my, your, of the system, ontology) "the red ball on the table". This would produce two searches: the first one for 'red ball', the other one for "ball on the table", and the two found paths would later be composed[1] in order to build the complete ontological form for this NP.[2]

This report concerns only this basic step of searching for paths. It also includes a description of the internal format of the ontology and lists and describes some basic reasoning functions.

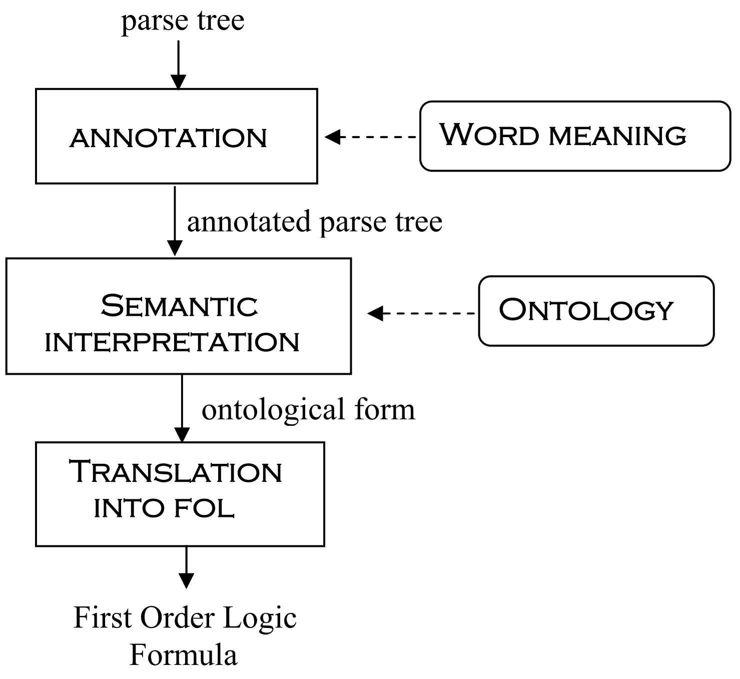


Figure 1 – From a parse tree to a FOL formula

[1] By the way, this could also be useful for disambiguation, since it is easy to imagine a "dance on the table", less easy (but, of course, not impossible) to think about a "red dance".

[2] The NP also include a determiner. It is disregarded in the ontological form. On the contrary, it can be used for building th final FOL formula, but the current implementation does not cover this (fundamental) step

2. THE ONTOLOGY

The ontology (OnTule) is composed of nodes, linked by arcs. The nodes can refer to *concepts* (classes), *relations* (binary), *instances* (individuals), and *relation instances* (individual pairs). The nodes are linked via labelled arcs. A node which has no arcs is not defined correctly. Another type of object present in the ontology are basic data (as an integer). They can only appear as second node of an arc labelled as *value.* It must be noted that this is not a 'true' ontology, since no primitives are provided for writing general axioms.

2.1. Structural Arcs

As all ontologies, OnTule is based on a set of *structural arcs* that express the basic links among objects present in the hierarchy. Most of the links are paired with an inverse link. The labels that identify the inverse link are called *dual* labels, so that:

∀ n1, n2 (arc-with-label (n1, n2) → arc-with-dual-label (n2,n1))

Below, we list the various arcs, together with their graphic representation. In Appendix A, a table lists all arcs[3].

2.1.1. Subclasses and instances

These are the standard taxonomic arcs:

- has-subclass
- subclass-of

- instance
- has-instance

We report here the graphical description we have adopted (Figure 2) and, in Box 1, the external declaration and the internal LISP representation. All concepts have *££entity* as common top-ancestor.

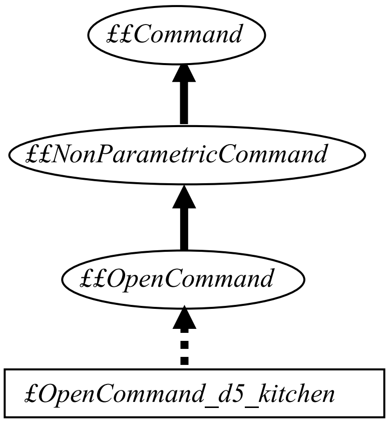


Figure 2 – Two links to superclasses and the *££OpenCommand_d5_kitchen* instance

2.1.2. Relations

The basic relation definition involves the following arcs

- range
  range-of
- domain
  domain-of

[3] For enhancing readability, we adopt the convention that:
- The prefix *££* identifies concepts (classes)
- The prefix *£* identifies instances
- The prefix *&* identifies relation and relation instances
- The prefix *§* identifies some 'special' instances, as *§myself* (the system), *§speaker* (the user)
- The prefix *$* identifies basic datatypes (as *$integer*)
- The prefix – identifies some pseudo-ontological items (as *–intensifier-adv*, used for 'very')

```
                                        external declaration
( subclass-of  ££NonParametricCommand ££Command)
( subclass-of ££OpenCommand ££NonParametricCommand)
( instance £OpenCommand_d5_kitchen ££OpenCommand)

                                        internal representation (LISP property lists)
--- properties of ££Command:
    - subclass-of (££Entity)              !!! not shown in the figure
    - has-subclass (... ££NonParametricCommand ...)
--- properties of ££NonParametricCommand:
    - subclass-of (... ££Command ...)
    - has-subclass (... ££OpenCommand ...)
--- properties of ££OpenCommand:
    - subclass-of (... ££NonParametricCommand ...)
    - has-instance (... £OpenCommand_d5_kitchen ...)
--- properties of £OpenCommand_d5_kitchen:
    - instance (... £OpenCommand ...)
```

Box 1 – *Use of taxonomic and instance links*

The actual representation of a (binary) relation is shown in fig.3.

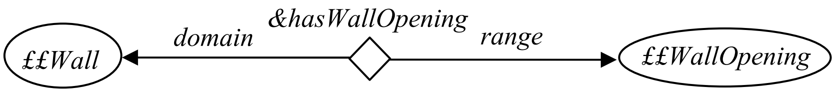


Figure 3 – *Range* and *domain* arcs for the relation (*&hasWallOpening*) that associates a *££Wall* with an opening in it (as a door)

The external declaration involves just the direct arcs (*range* and *domain*). But the internal representation includes the 'dual' links, that can appear in the result of the search for paths. So, given the arcs in Figure 3, the internal representation is shown in Box 2, and the search for a path from *££Wall* to *££WallOpening* will return '*££Wall domain-of &hasWallOpening range ££WallOpening*', while a path from *££WallOpening* to *££Wall* is '*££WallOpening range-of &hasWallOpening domain ££Wall*'. Note that no 'inverse' relation is required in the external declaration of the ontology, but explicit declarations are admitted.

```
                                        external declaration
 ( relation &hasWallOpening
          ( domain ££Wall) (range ££WallOpening) )

                                        internal representation (LISP property lists)
--- properties of &hasWallOpening:
    - range (££Wall)
    - domain (££WallOpening)
--- properties of ££Wall:
    - domain-of (... &hasWallOpening ...)
--- properties of ££WallOpening:
    - range-of (... &hasWallOpening ...)
```

Box 2 – *Range* and *domain* information for *&hasWallOpening*

Other arcs possibly associated with relational concepts are:

- restricts
- restricted-by
- inverse
- union
- one-of

restricts is a binary relation between relations. For instance *&has-mother* restricts *&has-parent*. In the following, I will often use the term *subrelation* for the most specific one; so, *&has-mother* is a subrelation of *&has-parent*. *restricted-by* is the dual arc of *restricts*.

inverse is another binary relation between relations, such that:

$$\forall \text{ r1, r2 (inverse (n1, n2)} \rightarrow \text{(domain (r1)} \equiv \text{range (r2)} \wedge \text{range (r1)} \equiv \text{domain (r2))}$$

Moreover, for each pair of relation instances (see next paragraph), it holds that:

$$\forall \text{ rinst1, rinst2, r1, r2 (relinst (rinst1, r1)} \wedge \text{relinst (rinst2, r2)} \wedge \text{inverse (n1, n2))} \rightarrow$$
$$\text{(argument (rinst1)} \equiv \text{value (rinst2)} \wedge \text{value (rinst11)} \equiv \text{argument (rinst2))}$$

Finally, *union* and *one-of* are not properly arcs, but rather operators for composing complex *range* or *domain* specifications. For instance, in the example of fig.4, it is obviously possible to define a class that has as its sole subclasses *££garage* and *££room*, but if the only use of this class is to act as the *range* of *&floorOf* (and, conversely *domain* of its *inverse &hasFloor),* it seems more reasonable to say that the *union* of the two classes plays that role. Similarly for *one-of*, which is a set of instances.

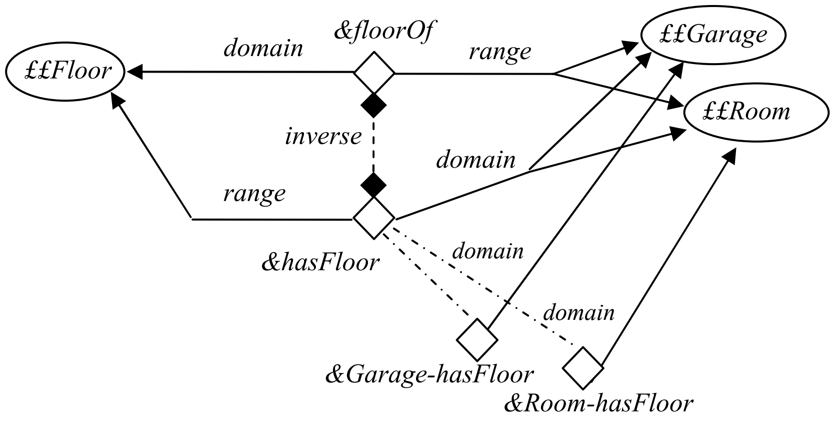


Figure 4 – *Union, restriction* and *inverse* arcs for the relation that associates a *££Floor* with its *environment*

2.1.3 Relation instances

The basic relation definition involves the following arcs

- relinstance
- argument
- arg-of
- value
- value-of

```
                                    external declaration
 ( relation &floorOf
    ( domain ££Floor) (range (union ££Garage ££Room)) )
 ( relation &hasFloor
     (domain (union ££Garage ££Room)) ( range ££Floor) )
( relation &Garage-hasFloor (restricts &hasFloor)
     (domain ££Garage) )
( relation &Room-hasFloor (restricts &hasFloor)
     (domain ££Room) )
(inverse &hasFloor &floorOf)

                                    internal representation (LISP property lists)
--- properties of &floorOf:
    - domain (££Floor)
    - range ((union ££Garage ££Room)))
    - inverse &hasFloor
--- properties of &hasFloor:
    - domain ((union ££Garage ££Room)))
    - range (££Floor)
    - inverse &hasFloor
    - restricted-by (&Garage-hasFloor &Room-hasFloor)
--- properties of &Garage-hasFloor:
    - domain (££Garage)
    - restricts (&hasFloor)
--- properties of &Room-hasFloor:
    - domain (££Room)
    - restricts (&hasFloor)
```

Box 3 – *union*, *inverse* and *restriction* information for *&floorOf*

Relation instances are pairs of individual instances, as exemplified in Figure 5. Apart from the obvious fact that the *argument* of a relation instance must be an instance of the *domain* class of the relation of which it is an instance (as for *value and range*), the relation instance can be viewed as 'named pair' of instances. If we take *&hasWallOpening* as a binary table (from a relational view), a relation instance is just a single tuple (pair) of that table. It may be more efficient to take apart instances (A-Box) from classes (T-Box), but we currently keep them together, as shown in Box 4.

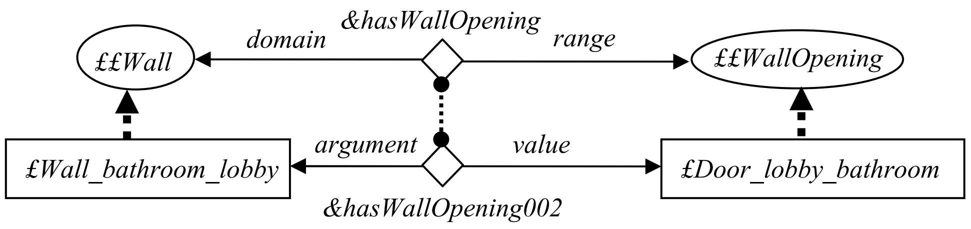


Figure 5 – *The &hasWallOpening002 relation instance*

The declaration of a relation instance is a bit more compact than the one of a relation (see the *relinstance* line in Box 4): after the *relinstance* keyword, there is its identifier (*&hasWallOpening002* in Box 4), then the id of the relation of which it is an instance (*&hasWallOpening*) and finally the two individual instances connected by the relation instance (*£Wall_bathroom_lobby* and *£Door_lobby_bathroom).* It is assumed that the first one is the (first) *argument*, while the second one is the *value* (second argument).

```
                                   external declaration
( subclass-of   ££Door ££WallOpening)             !!!!!! not shown in the figure !!!!!!
( instance      £Wall_bathroom_lobby  ££Wall )
( instance      £Door_lobby_bathroom ££Door)
( relation      &hasWallOpening  (domain ££Wall) (range ££WallOpening) )
( relinstance  &hasWallOpening002 &hasWallOpening £Wall_bathroom_lobby £Door_lobby_bathroom)

                                   internal representation (LISP property lists)
--- properties of &hasWallOpening
  - domain (££Wall)
  - range (££WallOpening)
  - has-relinstance (... &hasWallOpening002 ...)
--- properties of &hasWallOpening002
  - relinstance-of (&hasWallOpening)
  - argument (£Wall_bathroom_lobby )
  - value (£Door_lobby_bathroom)
--- properties of £Wall_bathroom_lobby
  - instance (... ££Wall ...)
  - arg-of (&hasWallOpening002)
--- properties of £Door_lobby_bathroom
  - instance (... ££Door ...)
  - value-of (&hasWallOpening002)
```

Box 4 – Definition of *a relation instance*

2.1 Disjointness

The final declaration appearing in the ontology concerns disjoint concepts. In order to save space, this is provided in terms of "disjoint sets": this type of information is not associated with the involved classes, but rather, all disjoint classes are grouped in a set. Then, the class is declared to belong to the set. So, instead of having:

(cl1 (disjoint-from (cl2 cl3 cl4 cl5)))
(cl2 (disjoint-from (cl1 cl3 cl4 cl5)))
(cl3 (disjoint-from (cl1 cl2 cl4 cl5)))
(cl4 (disjoint-from (cl1 cl2 cl3 cl5)))
(cl5 (disjoint-from (cl1 cl2 cl3 cl4)))

We have the declarations shown in Box 5.

```
                                   external declaration
  ( disjoint-set dsX (cl1 cl2 cl3 cl4 cl5))
  ( cl1 (disjoint-sets (... dsX ...) )
  ( cl2 (disjoint-sets (... dsX ...) )
  ( cl3 (disjoint-sets (... dsX ...) )
  ( cl4 (disjoint-sets (... dsX ...) )
  ( cl5 (disjoint-sets (... dsX ...) )
                                   internal representation (LISP property lists)
--- properties of dsX
      - disj-setdef  (cl1 cl2 cl3 cl4 cl5)
--- properties of cl1
     - disjoint-sets (... dsX ...)
--- properties of cl2
     - disjoint-sets (... dsX ...)
         ...
```

Box 5 – Definition of *disjointness* among concepts

In Box 5, *dsX* is a set of disjoint classes, i.e.

$\forall$ clX, clY (clX $\in$ dsX $\wedge$ clY $\in$ dsX) $\rightarrow$ (clX $\cap$ clY = Φ)

In the external declaration, the existence of a given 'disjoint-set' is provided separately, and then it is specified to which set(s) each class belongs.

3 SEARCH FOR PATHS

The search gets as input the nodes for which a connection path must be found. Both of them can be sets of nodes, in which case a kind of parallel search is carried out.

The search is mono- or bi-directional, according to the presence of constraints. In mono-directional search, the system starts from the first (set of) nodes and moves ahead until the second node (or one in the second set) is found. With respect to constraints, there are two types of them: positive and negative. Positive constraints are requirements of the solution path to pass through a given node (or one in a set of given nodes). Negative constraints force the solution path not to include any of the specified nodes. For positive constraints, the search is split into two mono-directional searches: from the constraint nodes to the start node and from the constraint nodes to the end node (of course, the final solution is obtained by first 'reversing' the first subpath and then appending to it the second subpath. Negative constraints impact on the advancements of the search paths, by blocking some expansions.

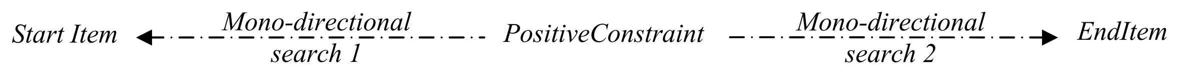


Figure 6 – *Use of (positive) constraints in the search for paths*

After some recursion steps, the first argument of the function that performs the search is a set of paths (initially, paths of length one). If none of them leads to the final node, then all paths are extended in parallel, so a set of longer paths is found and the recursion proceeds. In principle, each new path should be one arc longer[4] than before, but this is true just from a conceptual point of view. What actually happens is that taxonomic links (*subclass-of* and *has-subclass*) do not enter in the count, thus applying inheritance during the search. Apart from this special behaviour for taxonomic links, the search is a standard breadth-first search, but some heuristics, described below, dynamically limit, as far as possible, the branching factor.

The scheme of *find-shortest-path* is reported in Figure 7, where *node1* and *node2* can be either single nodes or list of nodes.

The basic loop for connecting the nodes is in *int-find-sh-path* (see Figure 8). If a positive constraint is present, then the actual function used is *int-2-find-sh-path* (Figure 9). This involves a loop on each current path, and, for each of them an advancement on the ontology. This is accomplished by *ontology-advance*, which is described below and synthetically sketched in Figure 10.

Before advancing on a path, that path is checked for unreasonable situations. Various heuristics are applied, that fall in the classes described in the next subparagraphs.

A first check is made on the input paths in order to exclude some of them (*prun-rels* in Figure 8). In particular, we exclude all paths that lead to a relation that is more general (*restricted-by*) a relation that is the ending point of another input path. The idea is that if a path ends in &open-door, it is useless to go ahead on a path ending in &open, because we already know that what must be opened is a door.

[4] One arc means two items: the arc label and the reached node. For instance <*range-of &hasFloor*>

3.1 Checks made on a single arc to be used for the extension

A first type of check on the expansions of a path concerns the single links that should be added to extend the path. Taking into account note 4 below, this means that these tests concern just the arc label (e.g. *range-of*) and not the reached node.

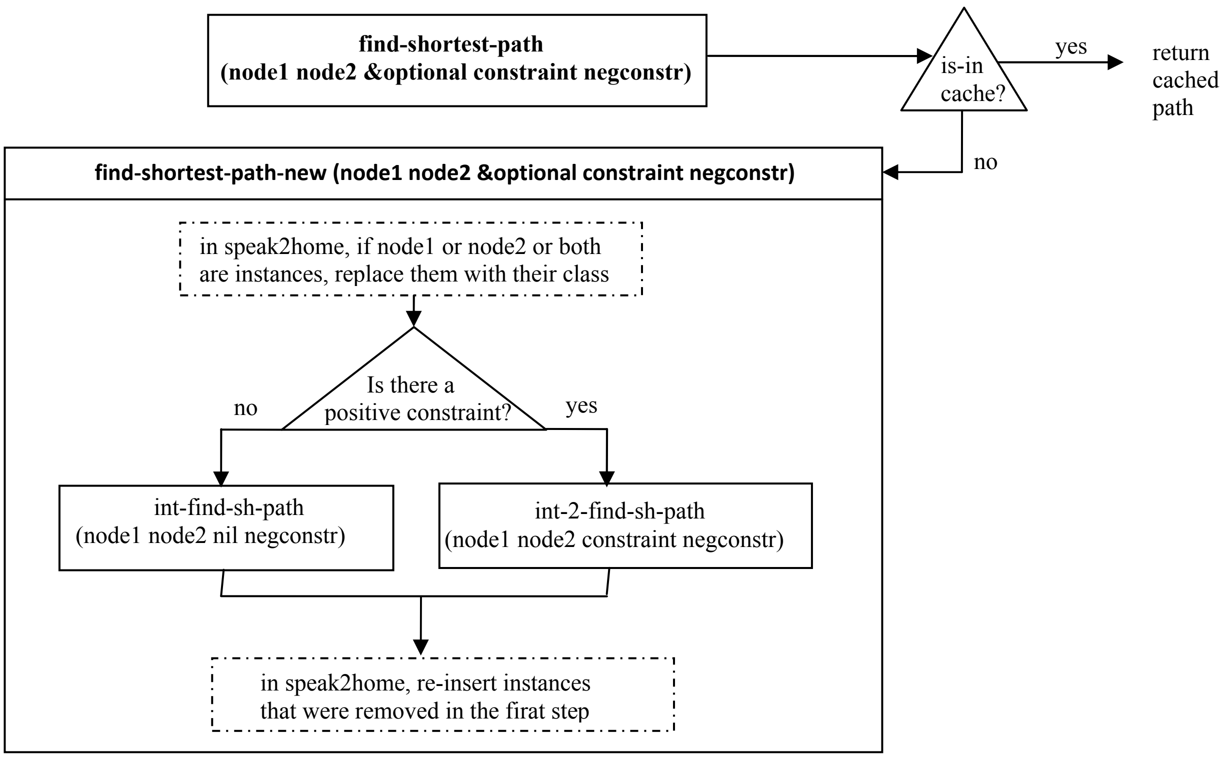
In particular, the extension is not allowed if:

a. The path to extend ends in <*domain &has-info-topic*> and the arc for the extension is *domain-of* (the same if *domain* and *domain-of* are exchanged).
b. The link to use for the extension is either *funct* or *disjoint-from* or *constraints*. These are not real links that can be used to form a path.
c. The path to extend ends in <*domain-of REL*> and the new arc is *domain* (the same for *range-of* and *range*). *Domain* and *range* arcs are 'functional', in the sense that each relation has a single domain and range (of course, they can be a *union* of concepts, but this does not change the matter). So, after the extension, we would get a loop.

3.2 Checks made on the path resulting after adding an arc

In these cases, both the arc-label and the reached node are relevant. For example, for test (d) below, it is not the label *subclass-of* which is relevant, but the fact that through that arc either the concept *££entity* or *££reified-relation* are reached via that arc.

d. The new path ends in <... *subclass-of ££entity*> or in <... *subclass-of ££reified-relation*>. If a path must link two entities semantically close, passing from *££entity* does not help, since anything can be linked in this way. The same for *££reified-relation*.
e. The path passes through *££dialogue-topic*, but does not start from *££dialogue*. We can make dialogues (start from *££dialogue*) but do not talk about them (no start from *££dialogue*).



find-shortest-path
(node1 node2 &optional constraint negconstr)
is-in cache?
yes
return cached path
no
find-shortest-path-new (node1 node2 &optional constraint negconstr)
in speak2home, if node1 or node2 or both are instances, replace them with their class
Is there a positive constraint?
no
yes
int-find-sh-path (node1 node2 nil negconstr)
int-2-find-sh-path (node1 node2 constraint negconstr)
in speak2home, re-insert instances that were removed in the firat step

Figure 7 – Main functions for looking for shortest paths linking two nodes (*speak2home*, appering in the dashed boxes is the name of a specific project where this technology has been used)

```
int-find-sh-path (prevpaths node2 backup-solution constraint &optional negconstr)

prevpaths ← prune-rels (prevpaths)
repeat for all 'path's in prevpaths
    if path ends in node2, and it is not an unreasonable path, add it to found
    if path ends in a superclass of node2, and it is not an unreasonable path,
        add it to found after having added the suitable subclass links
    if path ends in node2, but it is an unreasonable path, set it as backup-solution
            if any of the above applies, end of search, otherwise:
    nxtsteps ← ontology-advance (path nxtsteps prevpaths negconstr)
    nxtsteps ← remove-all (prevpaths nxtsteps)
```

Figure 8 – Actual operations for looking for shortest paths linking two nodes

```
int-2-find-sh-path (lnode1 lnode2 constraint &optional negconstr)

    [for speak2home]
    If constraint = &commandOf
        replace lnode1(which should be a subclass of ££command with a more specialized
         subclass (given the properties of the restrictions of &commandOf)

prevpaths ← prune-rels (prevpaths)
first-half ← int-find-sh-path (constraint lnode 1 nil nil negconstr)
second-half ← int-find-sh-path (constraint lnode 2 nil nil negconstr)
return the result of appending any path in second-half to any path in first-half (reversed)
```

Figure 9 – Actual operations for looking for shortest paths linking two nodes (with a constraint)

f. The new path ends in <*REL1 domain CONC domain-of REL2*> (or in <*REL1 range CONC range-of REL2*>) and *REL1* is a restriction of *REL2*, or viceversa. In this case, the path refers to the 'same' relation (though possibly more or less restricted), so that this is not a useful move.
g. The path ends in <*CONC1 subclass-of CONC has-subclass CONC2*>, but *CONC1* and *CONC2* are disjoint.
h. The path ends in <*SUBREL1 restricts REL restricted-by SUBREL2* >. Since we are looking for 'linguistic' paths between concepts, it is unreasonable that such a path passes for two restrictions of the same relation.

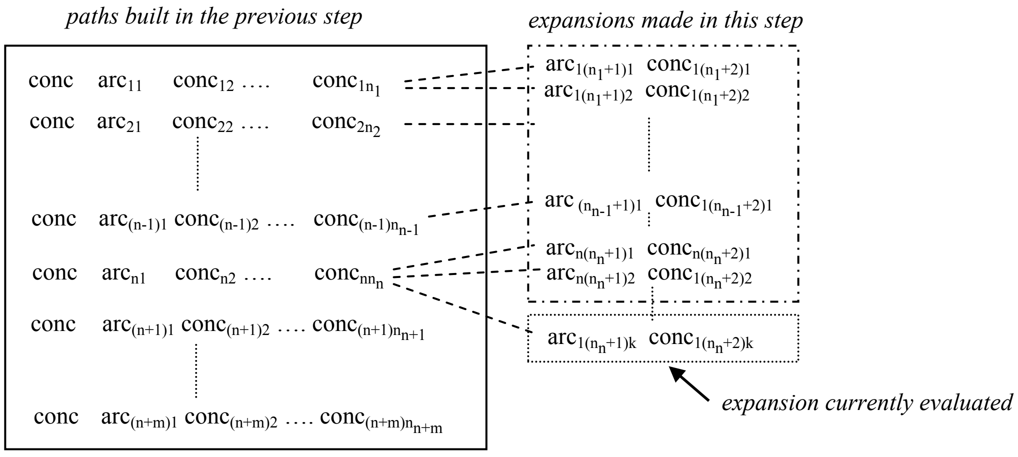
Some heuristics are related with special applications. For instance, in *Speak2Home* (domotics), we have:

i. The path has reached (via a *subclass-of* link) general concepts such as *££Command, ££Vertical, ££Collectable-obj*. The idea is analogous to (a) above: reaching *££Command* via a *subclass-of* link means that we were focused on a more specific command, so that going up to *££Command* is not promising. The same for the other concepts.
j. The path ends in <*... instance-of ££Functionality has-instance*>. Similar as above, but related with single (instance) functionalities.

3.3 Checks made on new path wrt all paths already generated in the same step

For these tests, the new extended path is, per se, acceptable, but it could be worse than some path already found (remember that we are moving ahead, in parallel). So, these tests involve all existing paths, built by extending paths ending in a different node than the current one and the already built extensions starting from the current node.

The situation is described in Figure 10. In the left box there are the paths obtained in the previous recursion step (n+m paths). In the upper right box, there is the result of the work already made in this step: all paths from 1 to n-1 have been expanded in some way. In the lower right box, there is the expansion that is currently attempted. It is an 'internally' good expansion, but it has to be compared with all the ones above (already expanded) and the ones below (waiting for expansion). If any of them is, according to some criteria, better than the new path <*conc* arc_{n1} $conc_{n2}$ …. $conc_{nn_n}$ $arc_{1(n_n+1)k}$ $conc_{1(n_n+2)k}$>, then the new path is not included in the result of this step. In other words, it will no more be inspected for further expansions.

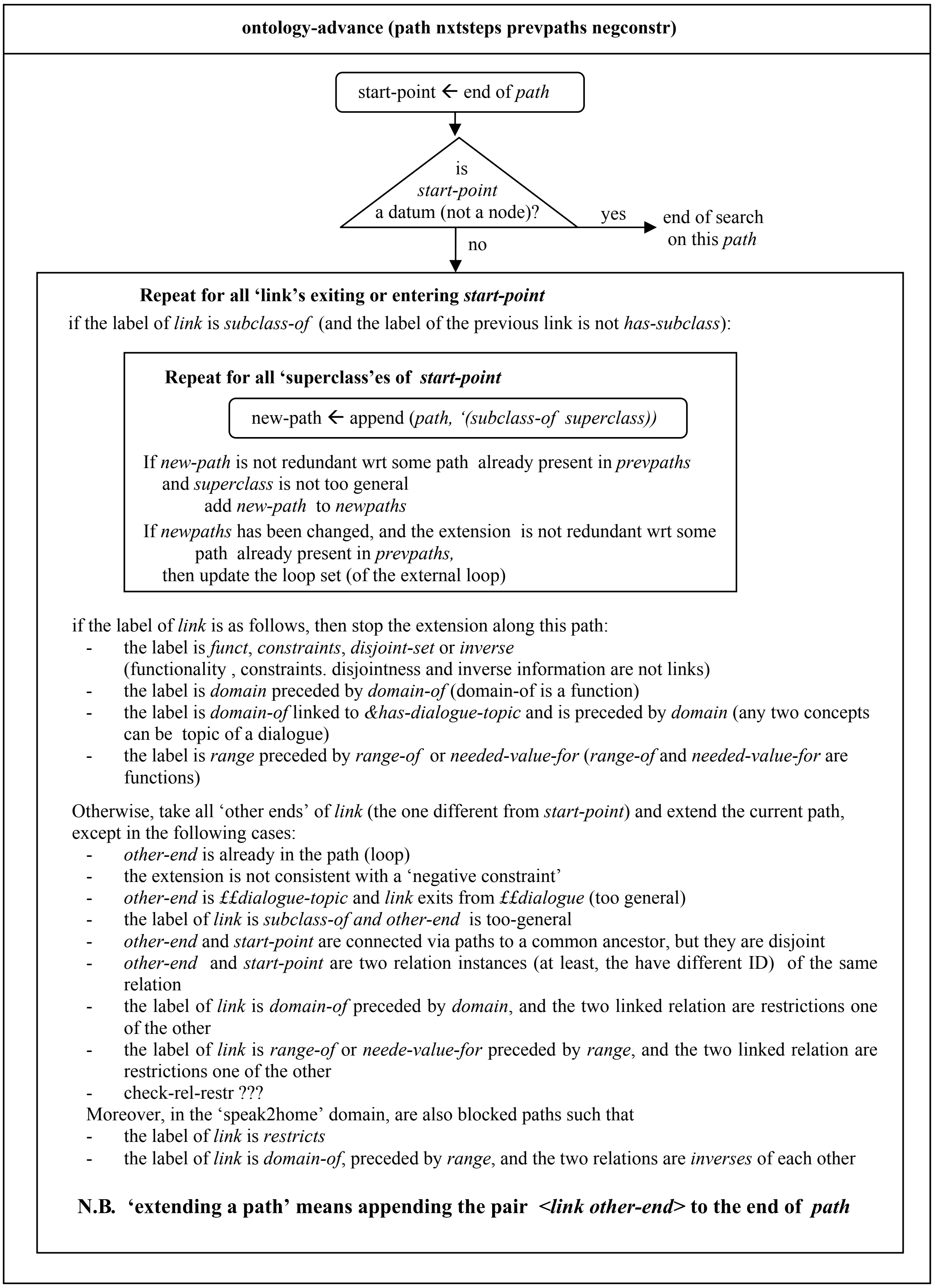


paths built in the previous step
expansions made in this step
conc arc_{11} $conc_{12}$ …. $conc_{1n_1}$
conc arc_{21} $conc_{22}$ …. $conc_{2n_2}$
conc $arc_{(n-1)1}$ $conc_{(n-1)2}$ …. $conc_{(n-1)n_{n-1}}$
conc arc_{n1} $conc_{n2}$ …. $conc_{nn_n}$
conc $arc_{(n+1)1}$ $conc_{(n+1)2}$ …. $conc_{(n+1)n_{n+1}}$
conc $arc_{(n+m)1}$ $conc_{(n+m)2}$ …. $conc_{(n+m)n_{n+m}}$
$arc_{1(n_1+1)1}$ $conc_{1(n_1+2)1}$
$arc_{1(n_1+1)2}$ $conc_{1(n_1+2)2}$
$arc_{(n_{n-1}+1)1}$ $conc_{1(n_{n-1}+2)1}$
$arc_{n(n_n+1)1}$ $conc_{n(n_n+2)1}$
$arc_{n(n_n+1)2}$ $conc_{1(n_n+2)2}$
$arc_{1(n_n+1)k}$ $conc_{1(n_n+2)k}$
expansion currently evaluated

Figure 10 – Comparison of a new path with the ones already existing

The full algorithm that carries out a one-level expansion is in Figure 11. Initially, it checks that the path to expand does not end in a *datum* (i.e. an integr, a string, etc.), in which case it cannot bring to any result (it is clear that a path leading to a string as "ok" must not be expanded: this will lead to any concept associated with an "ok" answer – too general).

The first (outermost) *repeat* refers to the links that can be used for a continuation (i.e. the arcs $arc_{n(n_n+1)1}$, $arc_{n(n_n+1)2}$, …., in Figure 10). For each of them, inheritance is applied (not shown in the figure), by moving up in the ontology (this is the "repeat for all superclass-es" box). After this, we have at disposal all properties of the concept from which the expansion starts (either expressed locally or in one of its superclasses, i.e. inherited). Now, the tests described in §3.2 (*if the label of link is as follows*) and in §3.3 (*Otherwise, take all 'other ends' of link*) are performed. Note that, in this respect, Figure 10 is a simplification, since any 'single' arc $arc_{n(n_n+1)1}$ can have more than one 'ending'. For instance, a *range-of* can end in many concepts, if the starting concept (the end of the current path) is the range of many relations.



ontology-advance (path nxtsteps prevpaths negconstr)

start-point ← end of *path*

is *start-point* a datum (not a node)? — yes → end of search on this *path*

no ↓

Repeat for all ‘link’s exiting or entering *start-point*

if the label of *link* is *subclass-of* (and the label of the previous link is not *has-subclass*):

Repeat for all ‘superclass’es of *start-point*

new-path ← append (*path, ‘(subclass-of superclass))*

If *new-path* is not redundant wrt some path already present in *prevpaths*
and *superclass* is not too general
add *new-path* to *newpaths*

If *newpaths* has been changed, and the extension is not redundant wrt some path already present in *prevpaths,*
then update the loop set (of the external loop)

if the label of *link* is as follows, then stop the extension along this path:

- the label is *funct*, *constraints*, *disjoint-set* or *inverse* (functionality , constraints. disjointness and inverse information are not links)
- the label is *domain* preceded by *domain-of* (domain-of is a function)
- the label is *domain-of* linked to *&has-dialogue-topic* and is preceded by *domain* (any two concepts can be topic of a dialogue)
- the label is *range* preceded by *range-of* or *needed-value-for* (*range-of* and *needed-value-for* are functions)

Otherwise, take all ‘other ends’ of *link* (the one different from *start-point*) and extend the current path, except in the following cases:

- *other-end* is already in the path (loop)
- the extension is not consistent with a ‘negative constraint’
- *other-end* is *££dialogue-topic* and *link* exits from *££dialogue* (too general)
- the label of *link* is *subclass-of and other-end* is too-general
- *other-end* and *start-point* are connected via paths to a common ancestor, but they are disjoint
- *other-end* and *start-point* are two relation instances (at least, the have different ID) of the same relation
- the label of *link* is *domain-of* preceded by *domain*, and the two linked relation are restrictions one of the other
- the label of *link* is *range-of* or *neede-value-for* preceded by *range*, and the two linked relation are restrictions one of the other
- check-rel-restr ???

Moreover, in the ‘speak2home’ domain, are also blocked paths such that

- the label of *link* is *restricts*
- the label of *link* is *domain-of*, preceded by *range*, and the two relations are *inverses* of each other

N.B. ‘extending a path’ means appending the pair *<link other-end>* to the end of *path*

Figure 11 – Scheme of the function that moves a step ahead in the ontology

APPENDIX A – Arc Labels

The table that follows lists the set of arc labels used in the (internal representation) of the ontology. The second column lists the dual labels (see §2.1). The third column list the *other-arg*. This is related with arcs associated with relations. Since all relations are binary, given the label identified one of the arguments (e.g. the *domain* of the relation) it is possible to identify the other argument (i.e. the *range* in case of *domain*).

label	dual label	other-arg label
domain	domain-of	range
domain-of	domain	range-of
range	range-of	domain
range-of	range	domain-of
has-subclass	subclass-of	
subclass-of	has-subclass	
argument	arg-of	value
arg-of	argument	value-of
value	value-of	argument
value-of	value	arg-of
needed-value-for	range	domain-of

Table 1 – The labels of structural arcs, their duals and the 'other-args'

APPENDIX B - Functions

B.1 Basic functions for inspecting the contents of the ontology

B.1.1 Arcs

- **is-a-structural-item (id)**
  INPUT: an identifier (any lisp atom)
  OUTPUT (boolean): true if *id* is a structural arc label (see Appendix A)
- **dual-arc (id)**
  INPUT: an identifier (any lisp atom)
  OUTPUT (boolean): the dual label if *id* is a structural arc label, otherwise *id*
- **other-arg (id)**
  INPUT: an identifier (any lisp atom)
  OUTPUT (boolean): the associated binary label, if *id* is a dom/range label, otherwise *id*

B.1.2 Predicates

- **is-an-instance (id)**
  INPUT: an identifier (any lisp atom)
  OUTPUT (boolean): true if *id* is the identifier of an instance
- **is-a-class (id)**
  INPUT: an identifier (any lisp atom)
  OUTPUT (boolean): true if *id* is the identifier of a class

- **is-a-relinstance (id)**
  INPUT: an identifier (any lisp atom)
  OUTPUT (boolean): true if *id* is the identifier of a relation instance
- **is-a-relation (id)**
  INPUT: an identifier (any lisp atom)
  OUTPUT (boolean): true if *id* is the identifier of a relation. Something is a relation if it has a domain or a range or both
- **is-basic-datatype (id)**
  INPUT: an identifier
  OUTPUT (boolean): true if *id* is a basic datatype (i.e. it is a subclass of *££datatype*)
- **is-subclass-of (down-class-id up-class-id)**
  INPUT: two class identifiers
  OUTPUT (boolean): true if the class *down-class-id* is a subclass (at any level) of the class *up-class-id*. True if *down-class-id* and *up-class-id* are the same class (currently, the same identifier)
- **one-is-subclass-of (down-classes-id up-class-id)**
  INPUT: the identifier or a list of identifiers of classes, and another class identifier
  OUTPUT (boolean): true any of the classes in *down-classes-id* is a subclass (at any level) of the class *up-class-id*.
- **is-subclass-of-one-of (down-class-id up-classes-id)**
  INPUT: the identifier of a class, and another identifier or a list of identifiers of classes
  OUTPUT (boolean): true if the class *down-class-id* is a subclass (at any level) of one of the classes in *up-classes-id* (which could also be a single class)
- **is-instance-of (instance-id class-id)**
  INPUT: the identifiers of an instance and of a class
  OUTPUT (boolean): true if the instance is member (at any level) of the class
- **is-relinstance-of (relinstance-id relation-id)**
  INPUT: the identifiers of a relation instance and of a relation (type)
  OUTPUT (boolean): true if the relation instance is member (at any level) of the relation
- **is-instance-or-subclass-of (object-id class-id)**
  INPUT:
    - o the identifier of an instance, or of a class, or a list of them (possibly nil)
    - o the identifier of a class

  OUTPUT (boolean): true ifany of the items occurring as first argument is an instance or a subclass of the class whose id is *class-id*
- **is-restriction-of (down-rel-id up-rel-id)**
  INPUT: two identifier of relations
  OUTPUT (boolean): true if the relation *down-rel-id* is the same or a restriction of the relation *up-rel-id*
- **one-is-restriction-of (down-rels-ids up-rel-id)**
  INPUT: a list of identifiers of relations and an identifier of a relations
  OUTPUT (boolean): true if any of the relations in the list *down-rels-ids* is the same or a restriction of the relation *up-rel-id*
- **is-restriction-of-one-of (down-rel-id up-rels-ids)**
  INPUT: an identifier of a relations and a list of identifiers of relations
  OUTPUT (boolean): true if the relation *down-rel-id* is the same or a restriction of one of the relations in the list *up-rels-ids*
- **ont-siblings (class-id-1 class-id-2 &optional level)**
  INPUT: two identifier of classes and an optional integer
  OUTPUT (boolean): true if the two classes are siblings. If *level* is 1, the result is always *nil*, since the only level 1 ancestor of a class is the class itself, so it cannot be a sibling of itself. If *level* is *nil*, the result is always *true*, since all classes have *££entity* as common ancestor.
- **check-same-node (nodes-id-1 nodes-id-2)**
  INPUT: two items that can be single ontology nodes or lists of ontology nodes
  OUTPUT: a kind of intersection of *nodes-id-1* and *nodes-id-2*. It cannot be a simple intersection just because either of the them can be a single node
- **are-inverses (rel1 rel2)**
  INPUT: two identifiers of relations

OUTPUT: a kind of intersection of *nodes-id-1* and *nodes-id-2*. It cannot be a simple intersection just because either of the them can be a single node

- **are-disjoint (conc1 conc2)**
  INPUT: two identifiers of concepts
  OUTPUT [boolean]:true *conc1* and *conc2*, or any of their superconcepts, are disjoint

B.1.3 Data retrieval

- **get-link-val (label id)**
  INPUT: an arc label and the identifier of an item in the ontology
  OUTPUT: the ontology item reached via an arc labelled *label*, starting form *id.* Nil if no such arc does exist
- **get-instance-class (instance-id)**
  INPUT: the identifier of an instance
  OUTPUT: the identifier of the 'immediate' class of the instance. Nil if *instance-id* is not the identifier of an instance
- **get-relinstance-rel (relinstance-id)**
  INPUT: the identifier of a relation instance
  OUTPUT: the identifier of the 'immediate' relation type of the instance. Nil if *relinstance-id* is not the identifier of a relation instance
- **get-rel-inverse (rel-id)**
  INPUT: the identifier of a relation
  OUTPUT: the identifier of the 'inverse' relation (if any), nil otherwise
- **get-superclasses (class-id &optional level)**
  INPUT: the identifier of a class and an optional integer
  OUTPUT: all superclasses of class, including *class-id*, going up until *level* levels. If level is less than or equal to 0, returns nil. If *level* is nil, returns the superclasses at any level
- **mult-get-superclasses (classes-ids &optional level)**
  INPUT: as above, but for multiple down classes
  OUTPUT: as above, but for multiple down classes
- **get-superrels (rels-ids)**
  INPUT: a list of *id* of relations
  OUTPUT: all relations that are restricted by (i.e. are super-relations of) a relation in *rels-ids*. Includes the *rels-ids* themselves.
- **find-subclass-path (class-es class)**
  INPUT: a class or a list of classes or a list of lists each of which ends in a class id and a class id
  OUTPUT: a path that leads from any item which is in *class-es* or at the end of a list in *class-es* to *class*
- **find-relation-restriction (relation concept dom-or-range)**
  INPUT: a relation id, a class id and either 'domain' or 'value' (in *dom-or-range*)
  OUTPUT: any subrelation of *relation* that has in its domain or range (according to the value of *dom-or-range*) *concept*. Used to extract the specific subrelation that applies to *concept*.
- **find-ont-relation (concept dom-or-range relation)**
  INPUT: a relation id, a class id and either 'domain' or 'value' (in *dom-or-range*)
  OUTPUT: Similar to the above, but this returns the first subrelation of *relation* that has exactly *concept* (and not its subclasses) as domain or range

B.2 Manipulation of paths

- **elim-has-subclass (path)**
  INPUT: a path on the ontology
  OUTPUT: the same path where all sequences <*... other-arc1 cl1 has-subclass cl2 has-subclass ... has-subclass cl-lowest other-arc2 ...*> such that *other-arc1* and *other-arc2* are not *has-subclass* are replaced by <*... other-arc1 cl-lowest other-arc2 ...*>
- **elim-subclass-of (path)**
  INPUT: a path on the ontology
  OUTPUT: the same path where all sequences <*... other-arc1 cl1 subclass-of cl2 subclass-of ... subclass-of cl-uppermost other-arc ...*> such that *other-arc1* and *other-arc2* are not *subclass-of* are replaced by <*... other-arc1 cl-uppermost other-arc2 ...*>

- **elim-has-subclass-of (path)**
  INPUT: a path on the ontology
  OUTPUT: the same path where all sequences <... *other-arc1 cl1 arc1 cl2 arc2 ... arcN clN other-arc ...*> such that *arc1, arc2, ... arcN* are either *has-subclass* or *subclass-of and other-arc1* and *other-arc2* are neither *has-subclass* nor *subclass-of* are replaced by <... *other-arc1 cl-uppermost other-arc2 ...*>
- **insert-has-subclass (subcl-path)**
  INPUT: a list of class identifiers
  OUTPUT: assuming that all *id* in *subcl-path* are superclasses of the next *id* in the list, inserts between any pair of *id* the item (arc) *has-subclass*
- **rev-links (path)**
  INPUT: a path
  OUTPUT: the same path where all structural links are replaced by their duals
- **find-lowest-subclass (path)**
  INPUT: a path
  OUTPUT: starting from the end of *path* the first node not reached via a *subclass-of* link
- **make-path-link (path newlinks)**
  INPUT: a path on the ontology and a set of extension pairs <link node>, given in the form <link1 node1 link2 node2 ...>
  OUTPUT: a set of pseudo paths: if *path* is <n1 n2 n3> and *newlinks* is as above, the function returns <<n1 n2 n3 <node1 link1>> <n1 n2 n3 <node2 link2>>>
- **check-rel-restr (path)**
  INPUT: a path on the ontology
  OUTPUT (boolean): true if the path passes (in suitable ways) through two different restrictions of the same relation.
- **check-negconstr (path arc-label value negconstr)**
  INPUT: a path on the ontology, a tentative extension pair <link value> and a negative constraint
  OUTPUT (boolean): true if the negative constraint is matched (so the extension is not acceptable)
  This happens in case:
    - The first of negative constraint is an atom; in this case it is assumed that *negconstr* has length one. The value of the function is true if the first of *negconstr* is present in *path*
    - The first of negative constraint is not an atom, and *link* (i.e. the proposed extension) appears in *negconstr*. *path* is assumed to have length 1 (see § ...)
- **check-disjointness (arc-label conc-id path)**
  INPUT: a tentative extension of a path (*arc-label* + *conc*) and the path being extended
  OUTPUT (boolean): true if the path includes a concept disjoint from *conc-id*. Only the end of *path* and nodes reached from the end via a suitable sequence of *subclass-of* and/or *has-subclass* links are taken into account in the check
- **include-inverses (prevpath firstpart conc arc-label)**
  INPUT: an existing path (*prevpath*) and a first portion of path (*firstpart*) with a tentative extension of it (*arc-label* + *conc*)
  OUTPUT (boolean): true if (after eliminating all final *has-subclass* and *sublass-of* links:
    - the final elements of the old path and of the new extended path are inverse relations
    - the second elements of the old path and of the new extended path are *'other-arg'* links
    - the third element of the old path is a subclass of the new extended path or viceversa
- **member-or-subcl (path nodelist)**
  INPUT: a path on the ontology and a list of nodes
  OUTPUT (boolean): true if the first of path or any node reachable from the start of path only via *subclass-of or has-subclass links* is member of *nodelist. E.g.*
    - member-or subcl ((item1 item2 item3) (item0 item1 item2)) → true
    - member-or subcl ((item1 subclass-of item2 item3) (item0 item1 item2)) → true
    - member-or subcl ((item1 subclass-of item2 item3) (item3 item4)) → false
- **cond-add-new-path (oldpaths firstpart conc link)**
  INPUT: a set of paths on the ontology (*oldpaths*), a single path (*firstpart*), a concept or a list of concepts (*conc*), and a link label (*link*)
  OUTPUT: a set of paths. This is an extended set of path with respect to *oldpaths*, provided that there are one or more extensions of *firstpart* by adding *link* and one of the concepts in *conc* that is better than all of the paths in *oldpaths*. In case no such new path exists, *oldpaths* is returned unchanged.

- **unreasonable-path (path)**
  INPUT: a path
  OUTPUT (boolean): true if the path satisfies some criteria for unreasonableness:
  The path is longer than one and:
    - the path ends in <*subclass-of "too-general-concept"*>, where *"too-general-concept* is a set including *££entity* and some other domain-specific concepts
    - the path passes through *££dialogue-topic*, but does not end in *££dialogue*
    - the path ends in <*££description subclass-of ££reified-relation*>
    - the path ends in an instance and we are not in the *atlas* or *speak2home* domains
    - the conditions in *check-rel-restr* (see above) hold.
- **prune-rels (paths)**
  INPUT: a set of paths
  OUTPUT: a subset of the input set, after exclusion of all the paths that have, as last element, a relation that is more general than a relation that is the last element of another path in *paths*.

B.3 Top level search

- **find-shortest-path (node1 node2 &optional constraint negconstr)**
  INPUT: two ontology nodes and some possible constraints concerning nodes where the found path must pass (*constraint*) or nodes where the found pass must not pass (*negconstr*)
  OUTPUT: a set of path in the ontology linking *node1* and *node2* (see ch.3 of this report)
- **find-shortest-path-new (node1 node2 &optional constraint negconstr)**
  INPUT: as above (find-shortest-path)
  OUTPUT: as above, but skipping the test about the presence of the path in cache
- **int-find-sh-path (prevpaths node2 backup-solution &optional negconstr)**
  INPUT: a set of paths (*prevpaths*), a node to be reached (*node2)*, a possible solution already found, but not respecting all criteria of effectiveness (*backup-solution*) and a possible negative constraint (see above)
  OUTPUT: as above, but without positive constraints (nodes which the solution must pass thorugh)
- **int-2-find-sh-path (lnode1 lnode2 constraint &optional negconstr)**
  INPUT: two lists of nodes (*lnode1* and *lnode2*), a constraint, i.e. a node through which the solution path must pass, and a possible negative constraint (see above)
  OUTPUT: all paths that link any of the nodes in *lnode1* with any of the nodes in *lnode2* and that satisfy both *constraint* and *negconstr* (if present)

B.4 Miscellanea

- **replace-role (negconstr them-role-label)**
  INPUT: a negative constraint and a label marking a thematic role
  OUTPUT: replaces the first occurrence of *(themrole)* in *negconstr* with *(them-role-label)*
- **merge-compounds (vals)**
  INPUT: a description (possibly involving *union* or *someval*) of a set concept identifiers
  OUTPUT: the *flattened* set of identifiers, where the keywords *union* and *somavel* have been removed
- **add-in-cache (node1 node2 constraint result)**
  INPUT: a connection (*result*) between two nodes, satisfying *constraint*
  OUTPUT: none. The side effect is that the cache is expanded with the new path.
- **find-in-cache (node1 node2 constraint)**
  INPUT: two node identifiers and a constraint
  OUTPUT: the connection (*result*) between two nodes, satisfying *constraint*, if it is present in the cache, otherwise nil

APPENDIX C – Loading an ontology

This Appendix specifies the very simple operations required of a user to load an ontology. The ontology may contain instances, but it is also possible to have two separate files, one for the concepts (T-Box) and another one for the instances (A-Box).

The input files need be simple text files. A portion of a 'concept' file is reported in Box 6. In case the T-box and the A-Box are kept in separate files, the two files have to be loaded with separate operations.

```
; ****  CONCEPTS ****************************************************
(defconcept ££air)
(defconcept ££amount-modification)
(defconcept ££anaphoric-description)
(defconcept ££anaphoric-time-interval-description)
(defconcept ££to-affect-1)
(defconcept ££to-arrive-1)
(defconcept ££applied-function)        ; reified relation
(defconcept ££bad-weather)
(defconcept ££cardinal-direction)
; ******* THE BACKBONE: SUBCLASSES ***************
(subclass-of ££datatype ££entity)
   (subclass-of $string ££datatype)
   (subclass-of $sound ££datatype)
     (subclass-of ££speech $sound)
(subclass-of ££measureunit ££entity)
  (subclass-of ££temperature-measureunit ££measureunit)
  (subclass-of ££pressure-measureunit ££measureunit)
(subclass-of ££comparison-operator ££entity)
(subclass-of ££spatial-location ££entity)
  (subclass-of ££geographic-area ££spatial-location)
    (subclass-of ££country-polit ££geographic-area)
    (subclass-of ££it-geogr-area ££geographic-area)
      (subclass-of ££it-region ££it-geogr-area)
        (subclass-of ££it-cardinal-region ££it-region)
          (subclass-of ££it-western-region ££it-cardinal-region)
          (subclass-of ££it-southern-region ££it-cardinal-region)
   ; *********************** RELATIONS *****************************************************************
(relation &part-bigger (domain ££part-of) (range ££physical-entity) (funct 1 N))
(relation &part-smaller (domain ££part-of) (range ££physical-entity) (funct 1 N))
(relation &part-selector (domain ££part-of) (range ££entity) (funct 1 N))
(relation &arriver (domain ££to-arrive-1) (range ££entity) (funct 1 1))
(relation &arrive-origin (domain ££to-arrive-1) (range ££entity) (funct 1 1))
(relation &arrive-receiver (domain ££to-arrive-1) (range ££living) (funct 1 1))
   ; *********************** INSTANCES *****************************************************************
(instance §speaker ££person)
(instance §speaker+others ££person)
(instance §generic-ag ££living)        ; generic agents (for impersonal pronouns)
(instance £north ££cardinal-direction)
(instance £south ££cardinal-direction)
(instance £east ££cardinal-direction)
(instance £west ££cardinal-direction)
(instance £greater-than ££comparison-operator)
(instance £monday ££deictic-day-description)
(instance £tuesday ££deictic-day-description)
   ; *********************** RELATION INSTANCES ********************************************
(relinstance &has-it-southarea-bigger &geogr-part-bigger £South-It-area £Italy)
(relinstance &has-it-southarea-smaller &geogr-part-smaller £South-It-area £it-southern-area)
(relinstance &has-it-southarea-selector &geogr-part-selector £South-It-area £South)
(relinstance &has-it-south-bigger &geogr-part-bigger £South-Italy-part-of £Italy)
(relinstance &has-it-south-smaller &geogr-part-smaller £South-Italy-part-of ££it-southern-region)
(relinstance &has-it-south-selector &geogr-part-selector £South-Italy-part-of £South)
(relinstance &has-it-north-bigger &geogr-part-bigger £North-Italy-part-of £Italy)
(relinstance &has-it-north-smaller &geogr-part-smaller £North-Italy-part-of ££it-northern-region)
(relinstance &has-it-north-selector &geogr-part-selector £North-Italy-part-of £North)
   ; *********************** DISJOINTNESS *****************************************************************
(disjoint-set (££Acoustic ££Actuator ££Sensor ££PowerDelivery ££Lighting))
```

Box 6 – An extract of an ontology, as it must be given to be loaded inside the LISP runtime memory