A Computational Model for the Efficient Retrieval of Very Large Structure-Based Knowledge Bases

José F. Quesada

Centro Informático Científico de Andalucía
Avda. Reina Mercedes, s/n, (41012) Seville, Spain
Tel: +34 - 5 - 462 - 3811
Fax: +34 - 5 - 462 - 4506
josefran@cica.es

Gabriel Amores

Departamento de Lengua Inglesa
Universidad de Sevilla
Palos de la Frontera s/n, (41004) Seville, Spain
Tel: +34 - 5 - 455 - 1588
Fax: +34 - 5 - 455 - 1516
gaby@fing.us.es

Abstract:

In this paper we present a computational model aimed at the efficient manipulation of large knowledge bases which use f-structures as their theoretical backbone. We propose a novel approach to the problem of the interface between the knowledge base and the inference engine, based on a layered architecture, where each module is aimed at improving a task of the interface.

The *KNOWLEDGE WRITER* module defines the KB specification language. Its task is to provide the necessary expressive power. This is achieved through the use of semantically very powerful constructions which enhance the representational adequacy. At this level, we have defined STFS (Soft-Typed-Feature Structures). This language, from the point of view of typification, may be placed half-way between untyped formalisms and completely typed ones.

The *SEEK RECEPTOR* and *KNOWLEDGE GENERATOR* modules serve as query languages. Their immediate goal is the achievement of inferential adequacy in the knowledge constructions returned to the inference engine. The *KNOWLEDGE COMPILER* module is primarily aimed at securing an optimal logical model of representation. From this model, the *KNOWLEDGE PROSPECTOR* will be able to perform an efficient use and retrieval of the KB. The *KNOWLEDGE COMPILER*, *KNOWLEDGE REPRESENTATION* and *KNOWLEDGE PROSPECTOR* modules use two types of techniques in order to achieve usage and storage efficiency. The first consists of a tetra-dimensional logical organization of the data adapting the representational model to access probabilities. The second technique incorporates a special type of binary trees (improved binary trees with vertical cut).

Keywords: feature structure representation formalisms, computational lexicons, representation of linguistic knowledge, computational efficiency, very large knowledge bases storage and retrieval.

1.- Feature structures as the representation of linguistic knowledge

Feature structures (we will use f-structures as a cover term hereafter, independent of any specific linguistic formalism) are currently used as a model of linguistic representation in a variety of linguistic theories and grammar formalisms: GPSG (Gazdar et al. 1985), LFG (Bresnan 1982), DCG (Pereira and Warren 1980), FUG (Kay 1983), PATR-II (Shieber et al. 1983) and HPSG (Pollard y Sag 1994), just to name a few.

So much so that we could consider them a "standard" in current NLP. From a formal point of view f-structures exhibit a number of characteristics which make them similar to frame-based and object-oriented knowledge representation languages.

Some of these characteristics include expressive power, flexibility, representational adequacy and recursivity. Furthermore, f-structures share with object-oriented formalisms properties such as high level of abstraction, class definitions, multiple-inheritance mechanisms and so on. These properties make f-structures especially suitable for their use in linguistics and, particularly, in the definition of lexical and syntagmatic hierarchies.

More recently, a number of formalisms have incorporated typed or sorted f-structures Carpenter (1992). Among them are TDL (Krieger and Schäfer 1994), ALE (Carpenter and Penn 1994), CUF (Dörre and Dorna 1993) and TFS (Emele and Zajac 1990, Emele 1994). Also, within logic programming some languages have appeared which make use of sorted feature terms such as Life (Aït-Kaci & Lincoln 1989) and Oz (Smolka et al 1995).

Formally speaking, sorted feature terms have an expressive power equivalent to first-order terms in Prolog (Mellish 1992). However, the flexibility and representational adequacy of feature structures, either typed (HPSG, GPSG) or untyped (DCG, PATR-II, LFG), make them more suitable for the description of linguistic knowledge.

This has motivated the implementation of tools that translate from feature structures into Prolog terms. Examples of such systems are CLE (Alshawi 1991), ALEP (Alshawi et al 1991), ProFIT (Erbach 1995) and Schöter (1993).

Nevertheless, most of the systems and tools mentioned above (whether designed for the manipulation of Prolog terms, f-structures or typed f-structures) present some problems of computational efficiency. They have focussed on aspects related to the syntax and semantics of the models, paying less attention to the computational model itself, especially with regard to storage and retrieval. Usually, the computational model is of little importance if we are developing prototypes with a reduced number of lexical entries. The scenario changes dramatically when the goal is the implementation of a real system (Wah 1994), where the size of the lexicon may condition the feasibility of the overall system.

In this paper we present a computational model aimed at the efficient manipulation of large knowledge bases which use f-structures as their theoretical backbone.

2. LektaKb: A novel approach for the efficient storage of very large knowledge bases

Within the field of knowledge engineering, the problems that arise in NLP as a result of the growth of the lexical database may be understood as a bottleneck in the communication between the inference engine (IE) and the knowledge base (KB). Figure 1 below illustrates this situation.

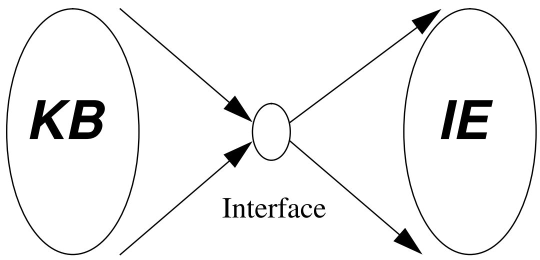

Figure 1: The bottleneck between the KB and the IE

As a starting point, it is usually required that the knowledge base specification module satisfies a series of properties such as representational adequacy, modularity, high level of abstraction and expressive power.

These requirements are necessary in order to satisfy criteria of adequacy in the interface between the system and the expert knowledge engineer. Unfortunately, it is also the case that those requirements are usually in contradiction with other properties which would allow an efficient interface between the inference engine and the KB.

Both sets of properties do not seem to be compatible with an efficient storage and use of the KB, where we have to take into account low-level computational issues pertaining to the physical architecture of the computer, its operating system and the characteristics of the compiler.

All these problems arise as a consequence of using a “direct” or “flat” model in the design of the interface between the KB and the IE. Rather, we propose a model based on a layered architecture.

In the figure below, the *KNOWLEDGE WRITER* module defines the KB specification language. Its task is to provide the necessary expressive power. This is achieved through the use of semantically very powerful constructions which enhance the representational adequacy.

The *SEEK RECEPTOR* and *KNOWLEDGE GENERATOR* modules serve as query languages. Their immediate goal is the achievement of inferential adequacy in the knowledge constructions returned to the inference engine.

The characteristics of the interface between levels 3 and 4 favor the use of a multi-protocol strategy for level 3. Thus, the system allows the definition of many specification languages aimed at different domains and, at the same time, many query languages specific to different IEs.

The interface between levels 2 and 3 consists of flat terms. Since their expressive power is equivalent to that of feature structures (Mellish 1992) they are able to convey the same type of information as in level 3. However, limitations relative to representational and inferential adequacy disfavor their use in the interface between levels 3 and 4. As a result, the *KNOWLEDGE WRITER* and *KNOWLEDGE GENERATOR* act as rewrite systems between complex structures and flat terms and viceversa.

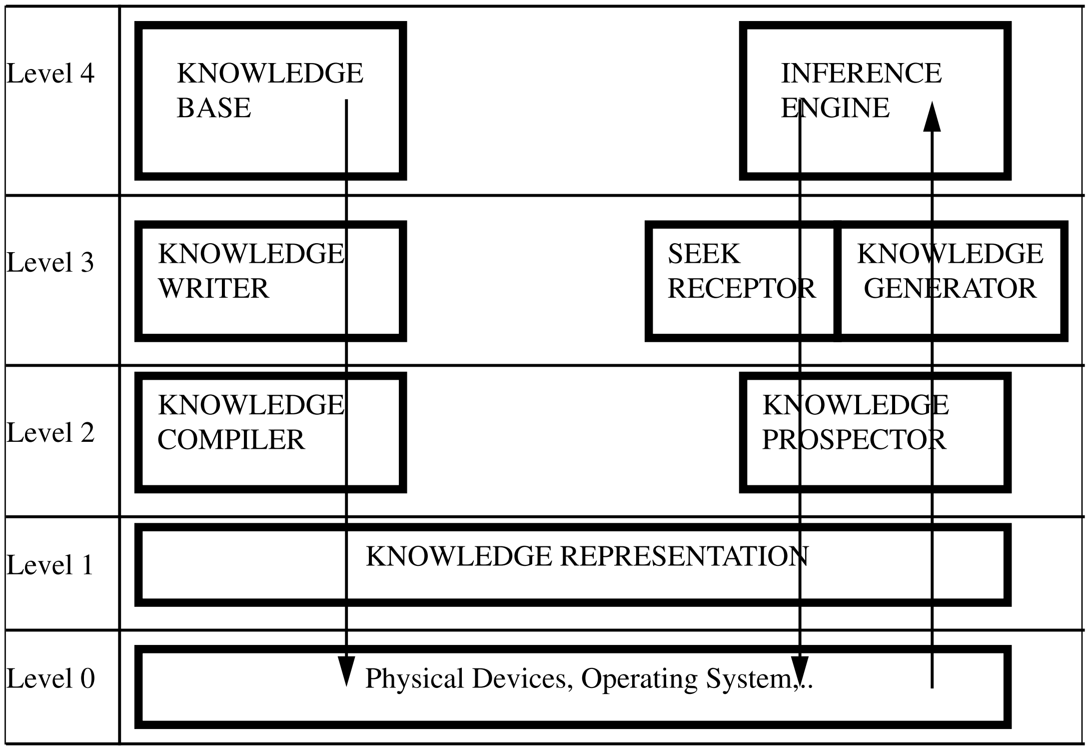

Figure 2: A layered architecture between KB and IE

The *KNOWLEDGE COMPILER* module is primarily aimed at securing an optimal logical model of representation. From this model, the *KNOWLEDGE PROSPECTOR* will be able to perform an efficient use and retrieval of the KB. In sum, level 2, as a whole, will be in charge of use and storage efficiency.

Finally, level 1 performs the actual storage and retrieval of the data. The interface between levels 1 and 2 makes use of packets with a very simple format, whose basic constituents are strings of characters. Level 1 aims at obtaining the best constituents adequacy possible between the logical model in level 2 and the physical properties of the devices (in level 0) involved in storage and retrieval.

The *KNOWLEDGE COMPILER*, *KNOWLEDGE REPRESENTATION* and *KNOWLEDGE PROSPECTOR* modules use two types of techniques in order to achieve usage and storage efficiency. The first consists of a tetra-dimensional logical organization of the data which adapts the representational model to access probabilities. The second technique incorporates a special type of binary trees (improved binary trees with vertical cut), whose motivation and advantages will be described in section 4.3 below.

3. The Specification Language: Soft-typed Feature Structures (STFS)

This section describes the implementation of a specification language intended for the description of lexical entries in an LFG-like formalism. As we pointed out above, this is just one of the many possible specification languages that could have been defined for different purposes.

From the point of view of typification, this language may be placed half-way between untyped formalisms such as PATR-II or DCG and completely typed ones such as ALE or TFS.

3.1 Shapes and Multiple Inheritance

The basic object used in STFS is that of "shapes". A shape defines the structure of a lexical entry, allowing for the definition of both typed and untyped structures. STFS also permits the definition of hierarchies of shapes organized as a partial order, including the standard mechanisms of multiple inheritance (Smolka and Aït-Kaci 1990). Following we show the syntax of the definition of shapes for a verbal entry in the lexicon.

```
DefShapes

Lex (LU)

% Auxiliary Shapes
AGR (agr:(gen,num,per))

% English verbs
Everb    (CAT:v,LU,MOR,pred,ggf,vtype,aktion,tense,
         asp:(fut,cond,part,prog,nec),
         num,form,
         subj:(role,form,semfeat,@AGR),
         obj:(role,form,semfeat,@AGR),
         obj2:(role),
         pobj:(role, pcase, semfeat),
         scomp:(role),
         acomp:(role),
         ncomp:(role),
         vcomp:(role))
```

3.2 Transformational Rules

A shape may have a number of transformational rules associated with it. The application of these rules on lexical entries which match the condition will generate new lexical entries. This is interesting in the case of LFG, since it allows us to define morphological rules and lexical redundancy rules using the same format.

A transformational rule consists of a pattern to be matched and a set of target structures. The pattern filters out the entries upon which the rule may be applied and the target structures specify the "effect" of the rule. These consist of a series of new shapes in whose definition we may use values from the original entry and a language for the manipulation of lexical information.

Below we give examples of transformational rules for the generation of -ing and -ed forms and examples of a lexical redundancy rule of "intransitivization". This rule generates an intransitive verb entry from a transitive verb which allows optional deletion of its object.

```
% ING RULES
% default, add 'ing'
% look -> looking
TRulePattern (MOR:[EING1])
TRuleTargets
{   (MOR:extract(base->MOR:,[EING1]))
    (LU:strcat(base->pred:,ing),asp:(part:pres),num:sing|plur,MOR:null())   }

% look at -> looking at
TRulePattern (MOR:[EING1c])
```

```
TRuleTargets
{   (MOR:extract(base->MOR:,[EING1c]))
    (LU:strcat(base->pred:[1],ing,' ',base->pred:[2+]),
              asp:(part:pres),num:sing|plur,MOR:null())  }

% stop -> stopping
TRulePattern (MOR:[EING2])
TRuleTargets
{   (MOR:extract(base->MOR:,[EING2]))
    (LU:strcat(base->pred:,strlast(base->pred:,1),ing),
               asp:(part:pres),num:sing|plur,MOR:null())  }

% solve -> solving
TRulePattern (MOR:[EING3])
TRuleTargets
{   (MOR:extract(base->MOR:,[EING3]))
    (LU:strtail(base->pred:,1,ing),asp:(part:pres),num:sing|plur,MOR:null()) }

% MORPHOLOGICAL RULES FOR ENGLISH REGULAR VERBS

% default add 'ed'
% look -> look, looked, looked
TRulePattern (MOR:[EED1])
TRuleTargets
{   (MOR:extract(base->MOR:,[EED1]))
    (LU:strcat(base->pred:,ed),tense:past,num:sing|plur,
                     MOR:extract(base->MOR:,[EED1]))
    (LU:strcat(base->pred:,ed),asp:(part:perf),
                     MOR:extract(base->MOR:,[EED1]))  }

% default add 'ed' for compound verbs
% look at -> look at, looked at, looked at
TRulePattern (MOR:[EED1c])
TRuleTargets
{   (MOR:extract(base->MOR:,[EED1c]))
    (LU:strcat(base->pred:[1],ed,' ',base->pred:[2+]),
         tense:past,num:sing|plur,MOR:extract(base->MOR:,[EED1c]))
    (LU:strcat(base->pred:[1],ed,' ',base->pred:[2+]),
                 asp:(part:perf),MOR:extract(base->MOR:,[EED1c]))  }

% solve -> solved
TRulePattern (MOR:[EED2])
TRuleTargets
{   (MOR:extract(base->MOR:,[EED2]))
    (LU:strcat(base->pred:,d),tense:past,num:sing|plur,
                      MOR:extract(base->MOR:,[EED2]))
    (LU:strcat(base->pred:,d),asp:(part:perf),
                      MOR:extract(base->MOR:,[EED2]))   }

% stop -> stop, stopped, stopped
TRulePattern (MOR:[EED4])
TRuleTargets
{   (MOR:extract(base->MOR:,[EED4]))
    (LU:strcat(base->pred:,strlast(base->pred:,1),ed),
        tense:past,num:sing|plur,MOR:extract(base->MOR:,[EED4]))
    (LU:strcat(base->pred:,strlast(base->pred:,1),ed),
                asp:(part:perf),MOR:extract(base->MOR:,[EED4]))  }

% LEXICAL-REDUNDANCY RULES

TRulePattern (vtype:[opt_obj])
TRuleTargets
{   (vtype:extract(base->vtype:,[opt_obj]))
    (vtype:extract(base->vtype:,[opt_obj]),ggf:[subj,0],obj:null())   }
```

3.3 Meta-Relations

Meta relations are a special type of transformational rule whose goal is the association of a shape with another virtual shape (a meta-shape) from which we have access to the first. In the case of NLP we could create a meta-shape called “lex”, which plays the role of a target of the meta-relations of other shapes. Effectively, a query on “lex” will result in a query about any of the shapes associated with it. A direct effect of this strategy is a transparent and compact solution of the problem of lexical ambiguity in NLP. An example is given below.

```
DefMetas

MetaPattern Everb ()
MetaTarget  Lex (LU:base->LU:)
```

3.4 Macros and Input Forms

The specification language permits the use of macros, whose behavior is similar to templates in PATR-II, ProFIT, etc.

```
<MPl> = (agr:(gen:masc,num:plural))
```

In the definition of large lexicons it is frequent that many lexical entries vary minimally with respect to each other, sharing many common features. The language permits the definition of Input Forms to ease the lexicographer’s job. For example, we could define different input forms using Levin’s (1993) verb classes. Following is an example:

```
DefIForms

levin_if (pred,MOR,vtype)
    Everb (pred:base->pred,MOR:base->MOR,vtype:base->vtype)
```

3.5 Basic Entries

Basic lexical entries are associated to shapes. For the definition of an entry, we may take advantage of macros, incorporating multiple inheritance mechanisms.

An entry may also be written making use of Input Forms, upon which the corresponding gene-ration models will be applied.

```
ActIForm levin_if

(bake, [EING3,EED2], [opt_obj])
(chop, [EING2,EED4], [opt_obj])
```

Once we have defined the entry (either directly or through an input form), transformational rules and meta-relations previously defined will be applied. The result obtained will constitute the know-ledge base.

4. The KNOWLEDGE COMPILER and the KNOWLEDGE PROSPECTOR

In this section we describe the *KNOWLEDGE COMPILER* and the *KNOWLEDGE PROSPECTOR*. The goal of these two modules is to obtain a logical model which permits a highly efficient storage and retrieval of information from very large knowledge bases.

Basically, the implementation makes use of two main strategies:
- Binary trees with vertical cut
- Tetra-dimensional organization of logical records

4.1 From standard to improved binary trees

Our search and storage algorithms utilize binary trees as the basic structure. If over a secondary storage device (usually a magnetic disk) we store *N* records (each with their corresponding full feature structure), the number of accesses needed to find a record (called A_{sbt}), using a standard binary search algorithm, is determined by (1) below:

$$A_{sbt} = ceil(\log(N)) \qquad (1)$$

where *ceil* is a function which returns the least integer which is not less than its argument. However, the retrieval of information from secondary devices uses physical sectors and not logical records as its basic unit.

If we take *R* to be the size of a logical record, and *T* to be the size of the physical sector, the blocking factor (*BF*) will be determined by (2).

$$BF = ceil\left(\frac{T}{R}\right) \qquad (2)$$

Using standard binary trees, each of the accesses of (1) above is generating an overhead (O_{sbt}) of wasted retrieved information proportional to the formula (3) below:

$$O_{sbt} = ceil(BF) - 1 \qquad (3)$$

In order to improve the efficiency of a binary search algorithm, a primary goal will be the reduction of the overhead obtained in (3). This may be achieved increasing the locality of reference of the information, which, in turn, will be achieved storing complete subtrees in each sector.

For a given *BF*, in each physical sector we may store a binary subtree of *ST* nodes, where *ST* is defined by (4).

$$ST = floorexp2(BF) \qquad (4)$$

where *floorexp2* returns the highest integer which is not higher than its argument (*floor*) and is also a power of 2 (*exp2*). Thus the reading overhead (O_{ibt}) for this kind of improved binary trees will be reduced to (5).

$$O_{ibt} = ceil(BF) - ceil(\log(BF)) \qquad (5)$$

and the number of accesses (A_{ibt}) to the device will in turn be reduced from the expression obtained in (1) to the one obtained by (6).

$$A_{ibt} = ceil\left(\frac{A_{sbt}}{\log(ST)}\right) \qquad (6)$$

As we can see, the efficiency of the model increases in logarithmic proportion to the number of nodes (actually the number of indexes of logical records) in the subtree stored in a physical sector.

An exhaustive implementation of this model will result in the utilization of wasted disk space. Namely, the size of wasted space will be determined by (7)

$$T - (ST \times R) \qquad (7)$$

The figure resulting from (7) in addition to the exponential pattern in which *ST* grows may result, in the worst case, in a waste of more than half of the necessary disk space.

In order to solve this problem, subtrees will not be stored in independent physical sectors. Rather, we allow for a subtree to be stored in one or more sectors as long as the sectors are immediately adjacent to each other. Logical records from more than one subtree are also allowed to appear in the same sector.

Thus we have achieved a complete use of the disk space and more flexibility to obtain the optimal value of *ST*.

Although this modification in the model should also result (at least theoretically) in a modification of the values obtained in (5) and (6) for O_{ibt} and A_{ibt}, the tests performed to compare both models show a level of real efficiency which is completely equivalent. We may therefore conclude that formulae (5) and (6) are in fact correct for the modification of the original model.

4.2 Tetra-dimensional organization of logical records

One of the consequences which follows from the utilization of a model of binary trees such as the one presented in the previous section is a tetra-dimensional organization of the logical records, as illustrated in figure 3 below.

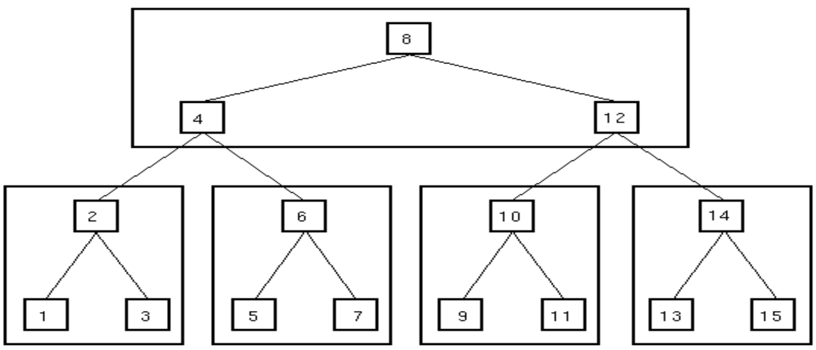

Figure 3: Tetra-dimensional Organization

The general binary tree, which in our case takes a height of order 4, is in turn divided into blocks (horizontal divisions of the complete tree).

Each block contains a number of sectors. In the example above, the first block contains one sector and the second block four sectors. Each sector contains a series of levels (the number of levels is the height of the sector). Within the same block, all sectors have the same number of levels, although sectors of different blocks may have different heights. In the example above, all sectors have two le-vels. Within each level, positions are numbered from left to right. In any of the sectors of the example, the first level contains one position and the second level two positions.

4.3 Improved binary trees with vertical cut (*ibtvc*)

Standard binary search algorithms usually use the average value as the root node of the tree. Using this value generates trees with a horizontal growing pattern. We show this pattern in figure 4, where the numbers associated with each node indicate the growing sequence.

In node 1 we will always have the exact average value (for an odd number of nodes) or the default average value (for an even number of nodes)[1].

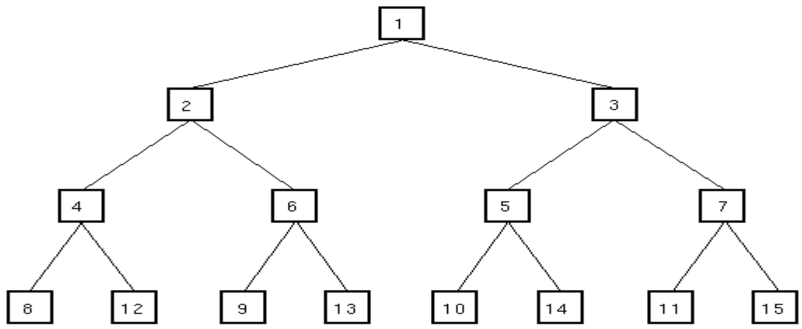

Figure 4: Binary Trees with Horizontal Cut (bthc)

In any case, this type of tree generates a horizontal growing pattern. Using this sort of trees in the tetra-dimensional organizational model described above will pose important problems in the last growing layer. This is so because the model requires sectors of a fixed size for each block. This results in an overhead for standard binary trees with horizontal cut (O_{sbthc}) of wasted disk space which, for the worst case, is determined by (8).

$$O_{sbthc} = \left(\frac{N}{2} - 1\right) \times R \qquad (8)$$

In order to avoid this problem while maintaining the advantages of binary search over the tetra-dimensional organization described above we define binary trees with a vertical cut as those whose root node is the value corresponding to the highest power of 2 in the order of the nodes.

As figure 5 illustrates, the growing pattern shows a perfect vertical cut. The figures associated with the nodes indicate the order in which the nodes will enter the tree. As can be seen, this order coincides with the relative order of the node in the tree, which did not happen in the model of figure 4.

1. If we use the average value in excess, the growing pattern will be the mirror image of the tree in figure 4.

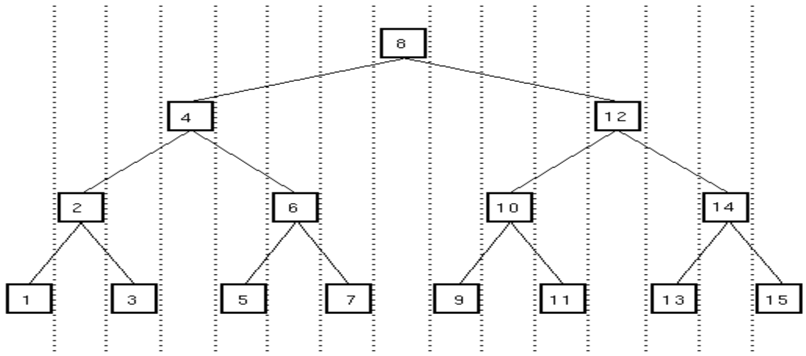

Figure 5: Binary Trees with Vertical Cut (btvc)

The formula in (9) defines the overhead of wasted disk space generated when we use binary trees with a vertical cut in conjunction with the tetra-dimensional organizational model described above. Let us assume that we have made a division into *B* blocks, each of which has a height of sector or number of block levels indicated by L_i ($1 \le i \le B$). Then, O_{ibtvc} (the overhead for improved binary trees with vertical cut) will be determined by (9):

$$O_{ibtvc} = \sum_{i=1}^{B} \left(2^{(L_i - 1)} - 1\right) \qquad (9)$$

For example, if we have a tree whose total height is 20, that is, a tree containing between 2^{19} (524,289) and 2^{20} (1,048,576) nodes, the O_{sbthc} varies between 524,287 and 0, with an average of 262,143. However, the O_{ibtvc}, assuming an organization in 3 blocks with 6 levels each, varies between 93 and 0, with an average of 46.

5. Experimental Measures of Efficiency

In this section we show some experimental results which are representative of the real efficiency achieved with the model outlined above.

5.1 Efficiency of the KNOWLEDGE COMPILER

This module uses as input the flat terms generated by the specification language and it must compile them following the ideas presented in section 4 above. In order to achieve more efficiency and modularity, the overall process has been divided into four stages:

• KbCompile-1: Performs the indexation. It generates the query indices for the structures (or flat-terms). Indexation by multiple attributes is permitted. We have implemented an indexation algorithm by subsets and consolidation. The real efficiency achieved is directly proportional to the number of records.

• KbCompile-2: Extracts repetitions, performing a modification in the representation of duplicates in the entries of the indexes. This results in a reduction of the size of the tree and compactation of the index.

• KbCompile-3: It obtains the tetra-dimensional model of organization. Basically, it decides the height (number of levels) of each block.

• KbCompile-4: Performs the actual storage. It transfers to the *KNOWLEDGE REPRESENTATION* module the packets containing subtrees which conform each logical sector for their subsequent storage.

The efficiency of this model may be demonstrated by the results shown in figure 6 below. These results were obtained under a DEC 3000 Model 400 AXP platform, over different knowledge bases with identical structure but different number of records. The times shown correspond to compilation time and index generation.

(1)	(2)	(3)
162	0.083	0.00051
1249	0.550	0.00044
4801	2.067	0.00043
19999	8.966	0.00044
57121	26.716	0.00046
101250	48.581	0.00047

(1) Number of records
(2) Total compilation time
(3) Average time per record

Figure 6: Experimental Results with the K. Compiler module

5.2 Efficiency of the KNOWLEDGE PROSPECTOR

This module receives search queries generated by IE and, if the search was successful, returns through the *KNOWLEDGE GENERATOR* the information retrieved.

Most of the ideas presented in the description of the compiling module were geared to improve the performance of the search module. Therefore, the efficiency of the *KNOWLEDGE PROSPECTOR* is a direct consequence of the model generated by the compiler.

In figure 7 below we show the times obtained by this module in finding information in different knowledge bases. We performed these tests under the same platform and knowledge bases described in 5.1 above.

Number of records	Search time
162	0.001
1249	0.001
4801	0.001
19999	0.002
57121	0.002
101250	0.002

NOTE: Times expressed in seconds

Figure 7: Experimental Results with the K. Prospector module

As can be seen, the practical results (real efficiency) obtained coincide with the computational efficiency predicted by the model. Namely, for the knowledge bases used in figures 6 and 7, the tetra-dimensional organization has generated blocks whose number of levels is close to 6. In figure 8 we show the resulting organization in blocks, sectors, levels and positions generated by the model.

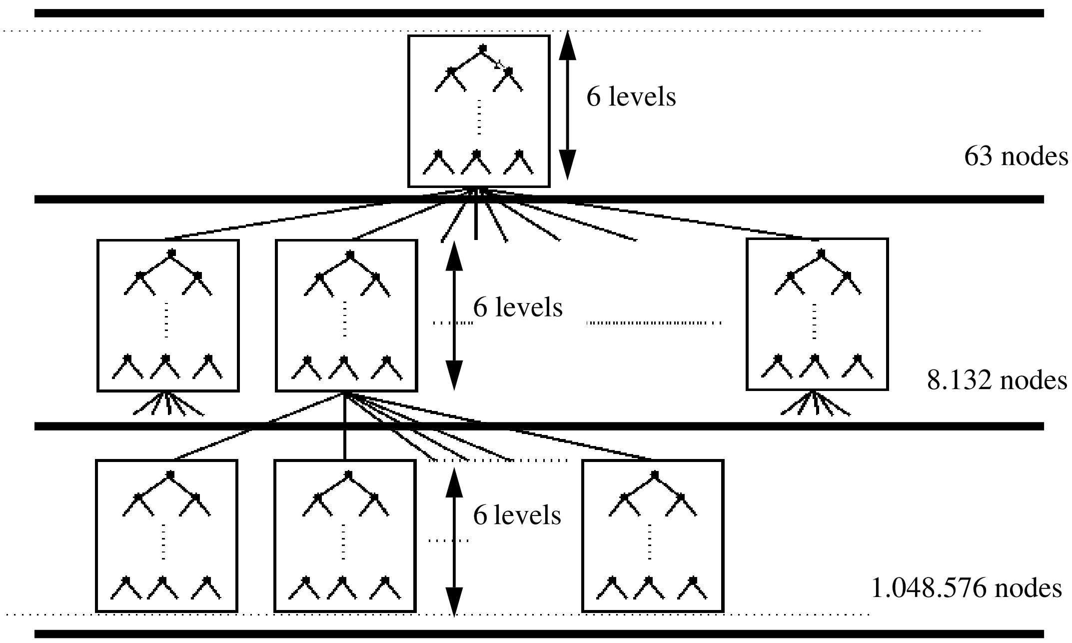

Figure 8: A sample organization

A single block may contain up to 2^6 (63) nodes. Between 2^6+1 (64) and 2^{13} (8,192) would require two blocks, and between 2^{13}+1 (8,193) and 2^{20} (1,048,576) nodes would need three blocks. As a consequence, the query time with a database of 100,000 records is only twice (0.002 sec) the time required over 100 records (0.001 sec in this case).

Finally, although the resources available did not allow us to test it practically, the model predicts that a database generating between $2^{20}+1$ (1,048,577) and 2^{27} (134,217,728) records would require four blocks. This implies that a search in a database of this size would take 0.004 sec using the same computational environment as that described above.

Conclusions

In this paper we have described a novel approach for the efficient storage and retrieval of very large knowledge bases.

We haved shown that a layered architecture at the interface between the knowledge base and the interface greatly incomes the computational efficiency while maintaining the desired expressive power of the specification language and inferencial adequacy of the query language.

In addition, we have also demonstrated how standard binary techniques can be improved in order to decrease wasted disk space and the number of accesses needed to find a record. This new model is based on binary trees with vertical cut and a tetra-dimensional organization ot the knowledge base.

Finally, we have demonstrated that the experimental results match the mathematical expectations of the model.

References

Aït-Kaci, H. and Lincoln, P. 1989. Life, a natural language for natural language. *T. A. Informations*, 30 (1-2): 37-67.

Alshawi, H. ed. 1991. *The Core Language Engine*. MIT Press.

Alshawi, H., Arnold, D. J., Backofen, R., Carter, D. M., Lindop, J., Netter, K., Tsujii, J. and Uszkoreit, H. 1991. Eurotra 6/1: Rule formalism and virtual machine design study - Final report. Technical report, SRI International, Cambridge.

Bresnan, J. ed. 1982. *The Mental Representation of Grammatical Relations*. Cambridge, Massachusetts: MIT Press.

Carpenter, B. 1992. *The logic of typed feature structures.* Cambridge Tracts in Theoretical Computer Science. Cambridge: Cambridge University Press.

Carpenter, B. and Penn, G. 1994. *ALE. The Attribute Logic Engine Version 2.0.1. User's Guide*. University of Pittsburgh.

Dörre, J. and Dorna, M. 1993. CUF - A formalism for linguistic knowledge representation. In Dörre, J. ed. *Computational Aspects of Constraint-Based Linguistic Description. Deliverable R1.2.A*. DYANA-2. ESPRIT Basic Research Project 6852.

Emele, M. and Zajac, R. 1990. Typed unification grammars. In *Proceedings of the 13th International Conference on Computational Linguistics, COLING-90*. Helsinki.

Emele, M. 1994. TFS - The Typed Feature Structure Representation Formalism. In *Proceedings of the International Workshop on Sharable Natural Language Resources*. Ikoma, Nara, Japan: Nara Institute of Science and Technology.

Erbach, G. 1995. ProFIT: Prolog with Features, Inheritance and Templates. CMP-LG e-print archive: cmp-lg/9502003

Gazdar, G., Klein, E., Pullum, G. and Sag, I. 1985. *Generalized Phrase Structure Grammar.* Cambridge, Massachusetts: Harvard University Press.

Kay, M. 1983. *Unification Grammar.* Technical Report. Palo Alto, California: Xerox Palo Alto Research Center.

Krieger, H. and Schäfer, U. 1994. TDL-a type description language for constraint-based grammars. In *Proceedings of the 15th International Conference on Computational Linguistics, COLING-94*, Kyoto, Japan.

Levin, B. 1993. *Verb classes and alternations. A Preliminary Study.* Chicago: The University of Chicago Press.

Mellish, C. S. 1992. Term-encodable descriptions spaces. In Brough, D. R. ed. *Logic Programming: New Frontiers*, pp. 189-207. Oxford: Intellect.

Pereira, F. C. N. and Warren, D. H. D. 1980. Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 1980, 13, 231-78.

Pollard, C. J., and Sag, I. A. 1994. *Head-driven Phrase Structure Grammar.* Chicago: Chicago University Press.

Shieber, S. M., Uszkoreit, H., Pereira, F. C. N., Robinson, J., and Tyson, M. 1983. The formalism and implementation of PATR-II. In *Research on Interactive Acquisition and Use of Knowledge*. Volume 1894 of SRI Final Report, SRI International, Menlo Park, California.

Schöter, A. P. 1993. Compiling feature structures into terms: A case study in Prolog. Technical Report RP-55, University of Edinburgh, Centre for Cognitive Science.

Smolka, G., Henz, M. and Würtz, J. 1995. Object-oriented concurrent constraint programming in Oz. In van Hentenryck, P. and Saraswat, V. eds. *Principles and Practice of Constraint Programming*. The MIT Press.

Smolka, G., and Aït-Kaci, H. 1990. Inheritance Hierarchies: Semantics and Unification. In Kirchner, c. ed. *Unification*. San Diego, California: Academic Press Inc.

Wah, B. W. ed. 1994. Report on Workshop on High Performance Computing and Communications for Grand Challenge Applications: Computer Vision, Speech and Natural Language Processing, and Artificial Intelligence. *IEEE Transactions on Knowledge and Data Engineering*, 5 (1), Feb. 1994.